Algoritmi Paralleli Appunti

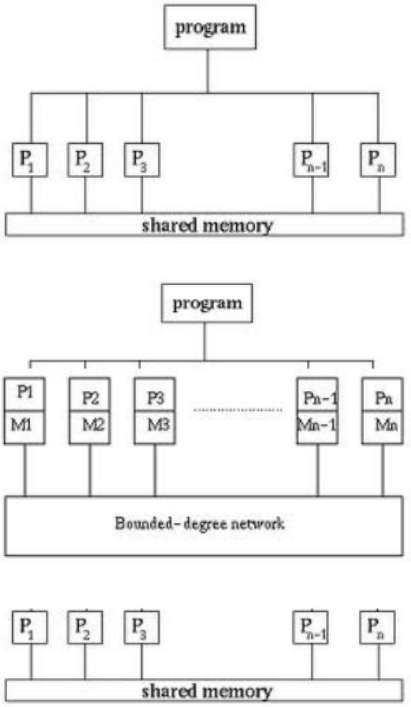
Introduzione

Algoritmo sequenziale e parallelo, nel primo caso una sola istruzione dell'algoritmo è eseguita per volta, nel secondo caso più istruzioni dello stesso algoritmo sono eseguite contemporaneamente. Un calcolatore parallelo ha più di 1 processore che possono essere di due tipi: sincroni o asincroni e avere due tipi di memoria: globale o locale.

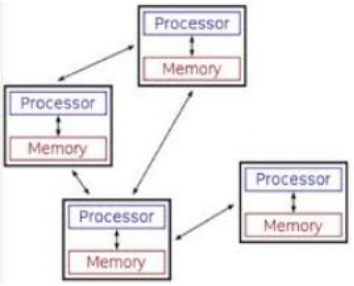
Tipologie di processori:
Processori sincroni: tutti i processori eseguono la stessa identica istruzione ma su dati diversi, tutti i processori quindi hanno lo stesso clock.
Processori asincroni: i processori eseguono simultaneamente diverse istruzioni su diversi dati, ogni processore ha quindi il proprio clock, il quale non è necessariamente sincronizzato con gli altri.

Tipologie di memoria:
Memoria globale: memoria unica condivisa tra tutti i processori, usata da essi per comunicare tra loro.
Memoria locale distribuita: ogni processore ha la propria memoria riservata, e comunicano attraverso una rete, leggendo dalle altre memorie o richiedendo i dati.

Modelli paralleli fondamentali:
- **PRAM (Parallel Random Access Machine)**: processori sincroni + memoria globale. Ogni singolo programma è lo stesso per tutti i processori, ogni istruzione è eseguita da tutti i processori, ma ogni processore può eseguire questa istruzione su dati diversi, ad esempio in un array di n elementi n processori elaborano una singola posizione.
- **Rete a grado limitato**: processori sincroni + memoria locale. Ogni processore ha la propria memoria ma tutti eseguono la stessa istruzione nello stesso momento. Per connettere le memorie dei processori si utilizza una rete a grafo, in cui i nodi sono i processori e gli archi la possibilità di scambiarsi dati tra loro. Lo scambio dei messaggi richiede più tempo. NOTA: PRAM può essere vista come una specializzazione di questo modello in cui il grafo è completo: tutti i nodi tra loro.
- **Modelli concorrenti**: processori asincroni + memoria globale. Ogni processore esegue istruzioni differenti nello stesso momento, avendo anche un clock differente, ma condividono la stessa memoria che gli permette di comunicare. Vi possono essere problemi di sincronizzazione.



- **Reti distribuite di calcolatori**: processori asincroni + memoria locale. Mentre i tre modelli precedenti erano implementabili su un solo computer, questo può essere implementato su calcolatori differenti. Anche qui è presente un grafo in cui i nodi comunicano tra loro attraverso una rete e rappresentano processori con una memoria locale che eseguono istruzioni diverse in tempi diversi.



Algoritmi PRAM
(processori sincroni - memoria globale)

Un costrutto parallelo è indicabile nel seguente modo:
for all i **where** inf ≤ i ≤ sup **do in parallel** operazione$_i$

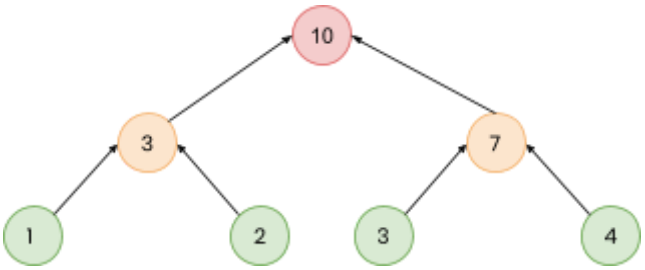
Ogni operazione è eseguita simultaneamente per ogni i compreso tra inf e sup su dati che dipendono da i. L'istruzione seguente non è iniziata finché tutti i processori non hanno terminato l'esecuzione.

Varianti del modello PRAM:
- **EREW** (Exclusive Read, Exclusive Write): processori distinti *non possono mai* accedere contemporaneamente alla stessa locazione di memoria né in lettura né in scrittura.
- **CREW** (Concurrent Read, Exclusive Write): processori distinti *possono* accedere contemporaneamente alla stessa locazione di memoria *solo in lettura* ma non in scrittura.
- **CRCW** (Concurrent Read, Concurrent Write): processori distinti *possono* accedere contemporaneamente alla stessa locazione di memoria sia in *lettura* che in *scrittura*.

Costi algoritmo:
P - Processori: ordine di grandezza del numero di processori richiesti per l'esecuzione dell'algoritmo in parallelo.
T - Tempo: ordine di grandezza del tempo impiegato per l'esecuzione dell'algoritmo in parallelo.
L - Lavoro: prodotto Tempo per Processori (L = PT), indica il tempo che si otterrebbe sequenzializzando l'algoritmo parallelo, cioè usando un solo processore che esegue un'istruzione per volta. NOTA: il lavoro dovrebbe essere uguale al tempo del miglior algoritmo sequenziale.

Sommatoria
Si assuma che n sia una potenza di 2 ($n = 2^h$) e si immagini di costruire un albero heap di 2n-1 nodi (così da contenere tutti i valori dell'albero). I numeri da sommare andranno nelle foglie dell'albero, ovvero nella seconda metà dell'heap (i nodi da n a 2n-1), man mano che si risale l'albero il padre di due nodi conterrà la loro somma: in generale quindi *un nodo i avrà la somma dei nodi figli 2i e 2i+1*. Ad esempio se i numeri da sommare sono 1, 2, 3, 4 si avrà:

num	10	3	7	1	2	3	4
pos	1	2	3	4	5	6	7

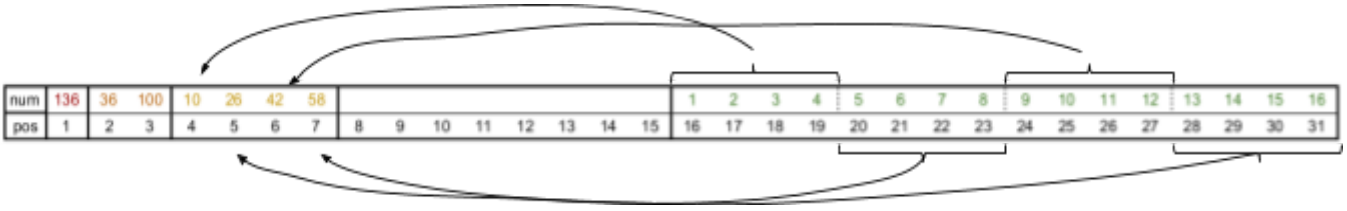



```
procedure SOMMATORIA(a, n):
  begin
    for k := log(n) - 1 downto 0 do
      for all i where 2^k ≤ i ≤ 2^(k+1) - 1 do in parallel
        a_i := a_2i + a_2i+1;
    return(a1);
  end
```

Il primo for viene eseguito log(n)-1 volte, ovvero per ogni livello dell'albero, partendo dal penultimo. Il for interno esegue in parallelo per tutti i nodi di ogni livello la somma dei figli.

Il *tempo totale* è *O(log n)* e il *numero di processori O(n)*. Il lavoro, che corrisponde al tempo richiesto dall'algoritmo nel caso in cui non fosse parallelo ma sequenziale, è L = T*P = O(log n) * O(n) = O(n log n).

Essendo il lavoro maggiore di un normale algoritmo sequenziale che somma in O(n) è possibile migliorarlo ripartendo gli n elementi iniziali in gruppi da log n elementi ciascuno, formando n/(log n) gruppi. Si fa poi sommare ad ogni processore gli elementi di ogni gruppo sequenzialmente.



n = 16 e log n = 4, quindi n/log n = 4, allora si avranno 4 processori per l'ultimo livello. In breve nel primo livello che contiene n/log n nodi vengono inserite le somme di ogni gruppetto di log n elementi, poi si risale l'albero come prima, sommando nei padri i valori dei figli.
Infatti il tempo è stato ridotto a O(log(n / log n)) = O(log n) (4), usando O(n / log n) processori (4). Infine il lavoro diventa L = T*P = O(log n) * O(n / log n) = O(n).

Somme prefisse

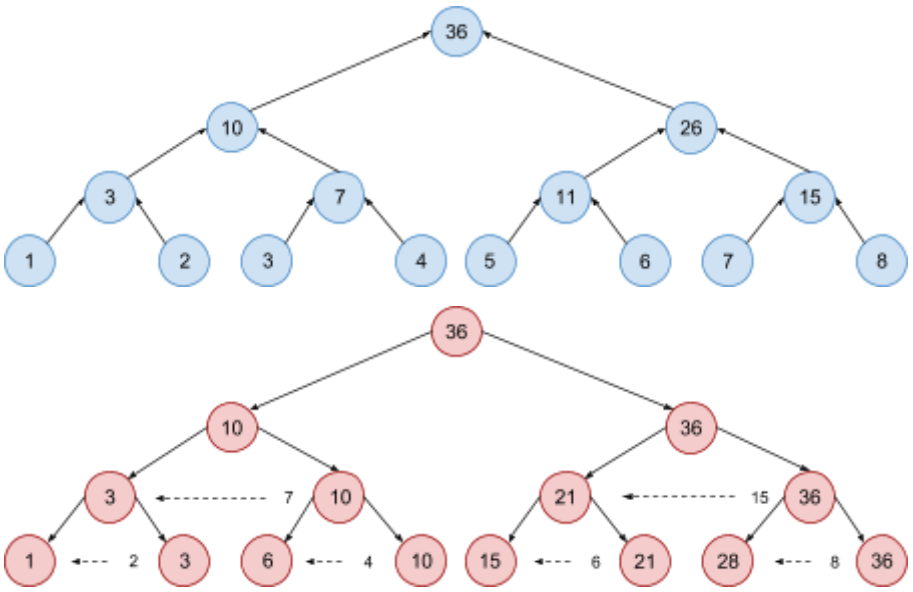
Dati n numeri interi $a_1, a_2, \ldots, a_n$, calcolare tutte le somme parziali $b_i = \sum_{1 \le j \le i} a_j$ per i = 1, 2, ..., n. Si assume ancora che n sia una potenza di 2 e che i numeri siano indicati con $a_{n+1}, \ldots, a_{2n-1}$ e che siano contenuti nella seconda metà di un albero heap di 2n - 1 nodi, ovvero risiedono nelle foglie. Al termine il risultato sarà contenuto nell'array b.

```
procedure SOMMEPREFISSE(a, b, n);
  begin
    b1 := SOMMATORIA(a, n);
    for k := 1 to log n do
      for all i where 2^k ≤ i ≤ 2^(k+1)–1 do in parallel
        if odd(i) then                          ← è un figlio destro
          bi := b[i/2]
        else                                    ← è un figlio sinistro
          bi := b[i/2] - ai+1
  end
```

Si risolve in due fasi, prima si genera l'albero a della sommatoria e si mette il risultato di a_1 in b_1 che è il nodo radice, poi per ogni livello dell'albero, partendo dalla radice, in parallelo
- se b_i è un *figlio destro* gli sarà assegnato il valore del padre.
- se b_i è un *figlio sinistro* gli sarà assegnato il valore del padre meno il valore a del fratello (cioè a_{i+1}).

Così b_i è uguale alla somma degli a_j dei nodi che si trovano allo stesso livello e che hanno indice $j \le i$. Quindi $b_{n+i} = a_n + \ldots + a_{n+i}$ per i = 0, 1, ..., n - 1.

b	36	10	36	3	10	21	36	1	3	6	10	15	21	18	36
a	36	10	26	3	7	11	15	1	2	3	4	5	6	7	8
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



La complessità della procedura è O(log n) utilizzando O(n) processori, che però possono essere ridotti a O(n / log n) con l'accorgimento spiegato per la SOMMATORIA, ottenendo un lavoro di O(n).

Ordinamento: Torneo

Data una sequenza $a_1, \ldots, a_n$ di n numeri, trovarne una permutazione tale che $a_1 \leq a_2 \leq \ldots \leq a_n$. Assumendo che i numeri siano tutti distinti tra loro, l'algoritmo del Torneo mette a confronto coppie di numeri facendo "passare" solo chi risulta maggiore. Il numero di perdenti dà la posizione in cui deve stare a_i all'interno della sequenza ordinata.

```
procedure TORNEO(a, n)
  begin
    for all i, j where 1 ≤ i, j ≤ n do in parallel        ← creazione matrice con gli "scontri"
      if a_i ≤ a_j then V_i,j := 1 else V_i,j := 0
    for all j where 1 ≤ j ≤ n do in parallel              ← somma delle vittorie di tutti
      P_j := SOMMATORIA(j-esima colonna di V, n);
    for all j where 1 ≤ j ≤ n do in parallel              ← #vittorie indica la nuova pos. in a'
      a_P_j := a_j
  end
```

V (j,i)	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	1	1	1	1	1	1	1	1
2	0	1	0	0	0	0	0	0
3	0	1	1	0	1	1	0	1
4	0	1	1	1	1	1	1	1
5	0	1	0	0	1	0	0	0
6	0	1	0	0	1	1	0	1
7	0	1	1	0	1	1	1	1
8	0	1	0	0	1	0	0	1
P	1	8	4	2	7	5	3	6

a	**10**	**80**	**48**	**26**	**79**	**51**	**35**	**63**
a'	**10**	**26**	**35**	**48**	**51**	**63**	**79**	**80**
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Questa procedura richiede O(log n) tempo e $O(n^2)$ processori, in quanto nel primo for vi sono due variabili $1 \leq i, j \leq n$ e perché vi sia un parallelismo sono necessari n^2 processori. Viene inoltre utilizzato il modello CREW perché nel confronto $a_i \leq a_j$ avviene un conflitto di lettura.

Per utilizzare EREW occorre evitare letture nella stessa cella. Questo può essere eliminato effettuando n copie di ciascuna a_i ed n copie di ciascuna a_j in tempo O(log n), in modo che tutti gli n^2 processori possano accedere in lettura in tempo O(1) ad un'unica coppia dei valori di a_i e a_j riservatagli. Per effettuare questa replica si può utilizzare un algoritmo come segue:

```
procedure REPLICA(copia, d, n)
  begin
    copia[1] := d
    for k := 0 to log n - 1 do
      for all i where 1 ≤ i ≤ 2^k do in parallel
        copia[i + 2^k] := copia[i]
  end
```

Copia il numero 5 per 8 volte:

5							
5	5						
5	5	5	5				
5	5	5	5	5	5	5	5
1	2	3	4	5	6	7	8

Questa procedura richiede O(log n) di tempo poiché ad ogni iterazione il numero di copie di d è raddoppiato, ed O(n) processori. Richiamandola in parallelo 2n volte creando due matrici R ed S, una la trasposta dell'altra, si possono effettuare n copie di ogni a_i ed a_j in tempo O(log n) utilizzando $O(n^2)$ e lavoro $O(n^2 \log n)$ processori con modello EREW.

a_1	a_1	a_1	a_1	a_1
a_i	a_i	a_i	a_i	a_i
a_n	a_n	a_n	a_n	a_n

$R_{i,j} = a_i$

a_1		a_j		a_n
a_1		a_j		a_n
a_1		a_j		a_n
a_1		a_j		a_n
a_1		a_j		a_n

$S_{i,j} = a_j$

Nel modello EREW va quindi inserito il confronto tra matrici invece che tra valori di a:

```
for all i, j where 1 ≤ i, j ≤ n do in parallel        ← creazione matrice con gli "scontri"
    if R_{i,j} ≤ S_{i,j} then V_{i,j} := 1 else V_{i,j} := 0
```

Nella procedura Torneo è possibile abbassare il numero di processori utilizzati se ogni processore calcola $O(\log n)$ elementi di $V_{i,j}$ riducendo il numero di processori a $O(n^2 / \log n)$ e ad un lavoro finale di $O(n^2)$ che resta comunque peggiore del miglior tempo sequenziale, ad esempio mergesort: $O(n \log n)$.

NOTA: Algoritmo Torneo che preveda la presenza anche di numeri uguali:

```
procedure TORNEO(a, n)
  begin
    for all i, j where 1 ≤ i, j ≤ n do in parallel
      if (a_i < a_j || i = j || (a_i = a_j && j > i)) then V_{i,j} := 1 else V_{i,j} := 0
    for all j where 1 ≤ j ≤ n do in parallel
      P_j := SOMMATORIA(j-esima colonna di V, n);
    for all j where 1 ≤ j ≤ n do in parallel
      a_{Pj} := a_j
  end
```

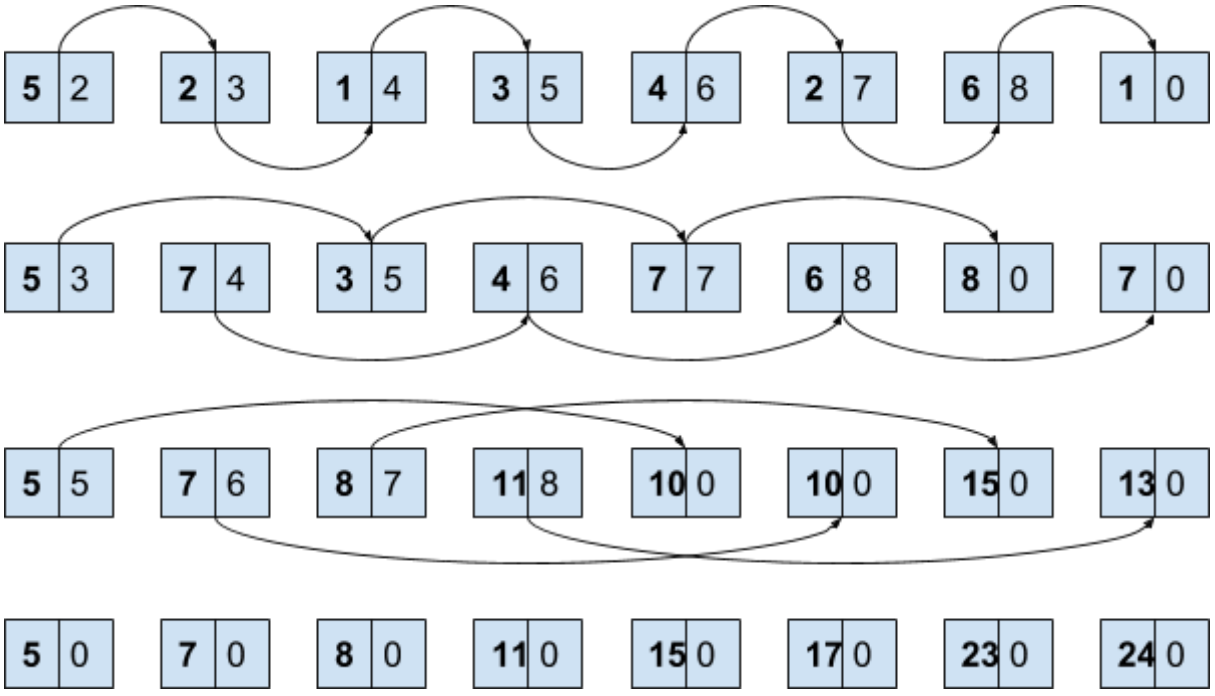
Pointer jumping: somme prefisse

Consideriamo il problema delle somme prefisse, dove però gli n elementi dati sono stavolta memorizzati in una lista monodirezionale. Si assuma che la lista sia in realtà rappresentata da due sequenze: $a_1, \ldots, a_n$ e $succ_1, \ldots, succ_n$. La prima contiene i valori degli elementi della lista e la seconda quella dei cursori i cui valori sono interpretati come indici che individuano la posizione degli elementi nella prima sequenza. Vi sono anche n processori ognuno responsabile per uno e un solo elemento della lista.

```
procedure SOMMEPREFISSE(a, b, succ, n)
  begin
    for all i where 1 ≤ i ≤ n do in parallel        ← copia l'array a in b
      b_i := a_i
    for k := 1 to [log n] do
      for all i where 1 ≤ i ≤ n do in parallel
        if succ_i ≠ 0 then begin
          b_{succ_i} := b_{succ_i} + b_i
          succ_i := succ_{succ_i}
        end
  end
```

Il lato sinistro conta il valore di k

-	**a**	**5**	2	**2**	3	**1**	4	**3**	5	**4**	6	**2**	7	**6**	8	**1**	0
	b	5		2		1		3		4		2		6		1	
1	**a**	**5**	3	**2**	4	**1**	5	**3**	6	**4**	7	**2**	8	**6**	0	**1**	0
	b	5		7		3		4		7		6		8		7	
2	**a**	**5**	5	**2**	6	**1**	7	**3**	8	**4**	0	**2**	0	**6**	0	**1**	0
	b	5		7		8		11		10		10		15		13	
3	**a**	**5**	0	**2**	0	**1**	0	**3**	0	**4**	0	**2**	0	**6**	0	**1**	0
	b	5		7		8		11		15		17		23		24	
		1		2		3		4		5		6		7		8	

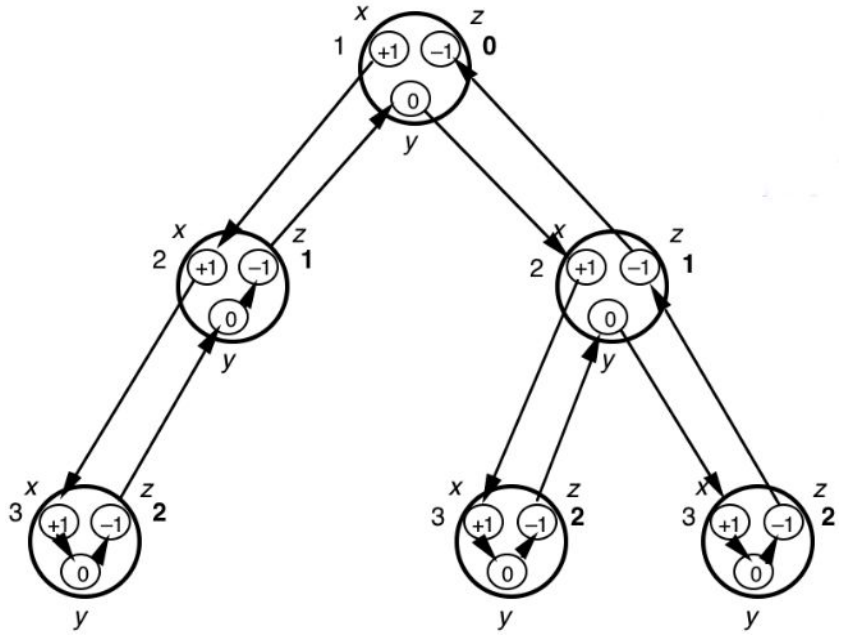


Nel for all più interno è utilizzata una tecnica chiamata pointer jumping, che permette non vi siano mai conflitti da parte di più processori per accedere alla medesima locazione di memoria, poiché tutti i valori di succ diversi da 0 sono distinti, il modello utilizzato è quindi EREW.
La procedura richiede O(log n) di tempo ed O(n) processori, poiché si parte con un'unica lista e ad ogni iterazione si ha un numero doppio di liste disgiunte, fino ad ottenere n liste di un solo elemento ciascuna. Il lavoro di questa procedura è O(n log n).

Pointer Jumping: cammino di Eulero

Un'altro esempio di utilizzo del pointer jumping è il calcolo dei livelli dei nodi di un albero binario. Dato un albero binario di n nodi, realizzato con puntatori, calcolare il livello di ciascun nodo, dove il livello della radice è 0, mentre il livello di ogni nodo è pari al livello di suo padre aumentato di 1. Per far ciò si utilizza un'altra tecnica, detta "ciclo Euleriano", un ciclo Euleriano di un grafo G è un ciclo che include ogni arco una e una sola volta e G contiene un ciclo Euleriano se e solo se è connesso.

Dato un albero binario B realizzato con puntatori, calcolare il livello dei nodi.
1. ogni nodo u di B è sostituito con tre elementi (puntatori) x, y, e z di una lista L, rappresentanti rispettivamente figlio sinistro, figlio destro e padre del nodo.
2. I 3n elementi di L sono gestiti ciascuno da uno ed un solo processore.
3. L'elemento x del nodo u vale +1 e punta all'elemento x del figlio sinistro di u, se esiste, altrimenti all'elemento y di u stesso.
4. L'elemento y del nodo u vale 0 e punta all'elemento x del figlio destro di u, se esiste, altrimenti all'elemento z di u stesso.
5. L'elemento z del nodo u vale -1 e punta all'elemento y del padre di u, se u è un figlio sinistro, oppure all'elemento z del padre di u, se u è un figlio destro, oppure a 0, se u è la radice.
Si calcolano le somme prefisse di L e i valori delle z danno i livelli, questo con un tempo O(log n) e numero di processori O(n), con modello EREW.



Tesi della computazione parallela

Si dice che un algoritmo è in **tempo polinomiale** se il suo tempo di esecuzione è limitato superiormente da un ordine di $O(n^k)$ per una qualche costante k.
Ogni problema risolubile in tempo polinomiale con un algoritmo parallelo può essere risolto in spazio polinomiale con un algoritmo sequenziale, e viceversa.

Si considerano “trattabili” in parallelo i problemi risolubili con algoritmi paralleli tali che $P = O(n^k)$ e $T = O(\log^h n)$,con n dimensione del problema e k ed h costanti. Una funzione $O(\log^h n)$, con h costante è detta “polilogaritmica”.

I problemi **NC** sono i problemi efficientemente parallelizzabili cioè che possono essere risolti in tempo polilogaritmico avendo a disposizione una quantità di hardware polinomiale rispetto alla dimensione dell'input.

La classe **P** contiene tutti i problemi decisionali che possono essere risolti da una macchina di Turing deterministica usando una quantità polinomiale di tempo di computazione, o tempo polinomiale. Non si sa se NC sia propriamente contenuta in P o se le due classi coincidono.

Per la Tesi della Computazione Parallela, si considerano “intrattabili” in parallelo i problemi in P non risolubili con algoritmi sequenziali in spazio $O(\log^h n)$.
Un problema in P è **LOG-SPAZIO-completo** se ogni altro problema in P si riduce ad esso con una trasformazione che richiede tempo polinomiale e spazio polilogaritmico. Tuttavia non è noto alcun algoritmo sequenziale con tempo polinomiale e spazio polilogaritmico per nessun problema log-SPAZIO-completo, e quindi neanche alcun algoritmo parallelo con $P = O(n^k)$ e $T = O(\log^h n)$.
Tali problemi sono considerati inerentemente sequenziali (a meno che P = NC)!

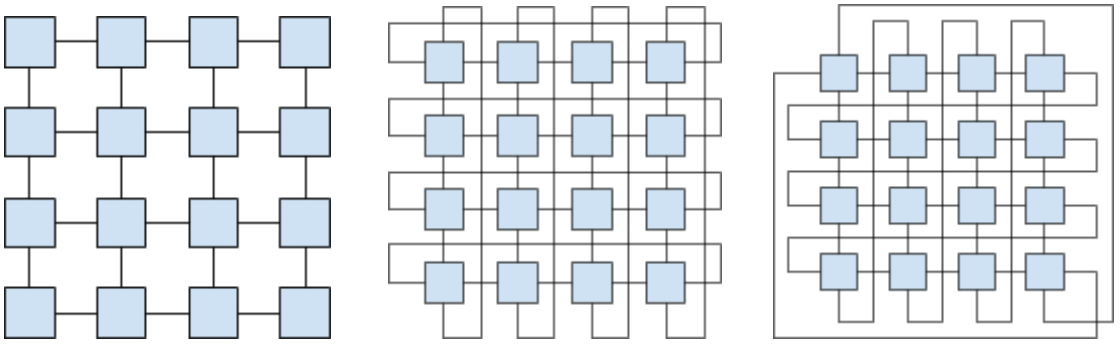
Algoritmi per Reti a Grado Limitato
(processori sincroni - memoria locale)

Quando si parla di rete di interconnessione si intende una struttura a grafo solitamente non orientato. I processori vengono indicati come i nodi del grafo, e gli archi rappresentano le comunicazioni dirette tra processori adiacenti. Un processore ha la propria memoria e può accedere in sola lettura alla memoria di un processore con cui è collegato.

Il **grado massimo g**, è il numero massimo di vicini di un nodo, mentre il **diametro d**, è il numero di archi nel cammino minimo tra i nodi più lontani, ovvero il tempo massimo per scambiare informazioni tra due nodi del grafo, il **grado** di un nodo è il numero di nodi connessi ad un certo nodo. Perché una rete di interconnessione sia fisicamente costruibile, non può essere un grafo completo (come ad esempio il modello PRAM), ma deve avere un grado massimo limitato superiormente da una costante o al massimo dal logaritmo del numero dei processori.

Mesh
In una mesh quadrata gli n processori con $n = p^2$ per un p intero positivo sono disposti in una matrice quadrata di dimensione $\sqrt{n} \times \sqrt{n}$. Il grado massimo g di ogni nodo è 4, mentre il diametro d è $O(\sqrt{n})$. Il generico processore $P_{i,j}$ con $1 \leq i, j \leq \sqrt{n}$, è connesso con i processori $P_{i,j\pm1}$ e $P_{i\pm1,j}$ (colonna prima e colonna dopo, e riga sopra e riga sotto) se esistono.

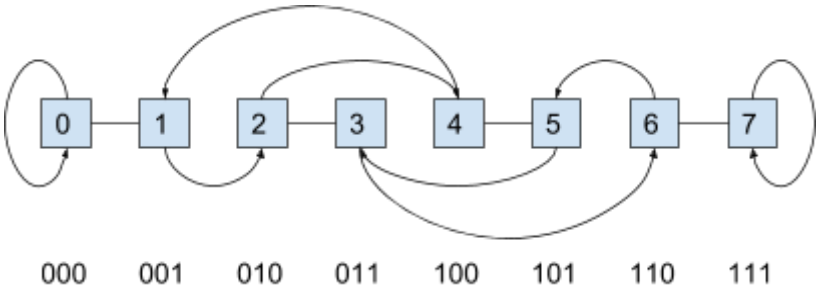
Le varianti comprendono la mesh ciclica, in cui il primo e l'ultimo processore di ogni riga e colonna sono connessi, e quella toroidale, dove l'ultimo processore di ogni riga e colonna è connesso con il primo processore della riga e colonna successiva.



Mesh quadrata | Mesh ciclica | Mesh toroidale

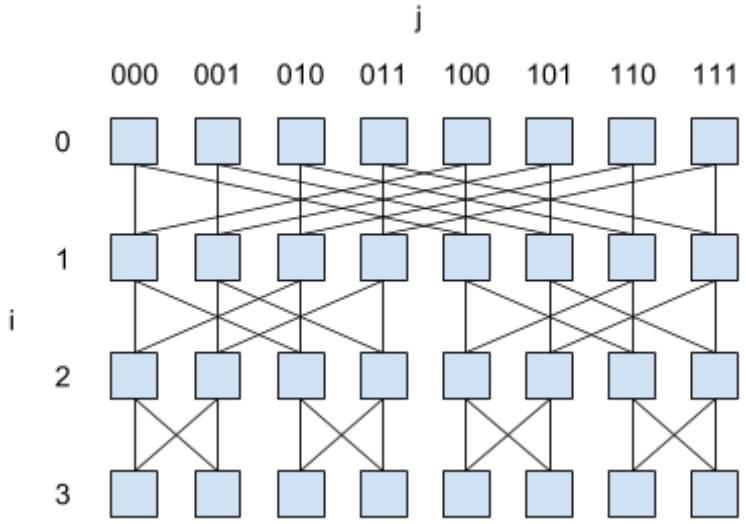
Shuffle

Gli n processori con $n = 2^p$ sono numerati in binario con indici di p bit. Hanno grado 3, ogni nodo è infatti connesso a 3 elementi e il diametro, ovvero il grado massimo per raggiungere il nodo più lontano, è O(log n). Vi sono due tipi di connessioni: quelle non orientate di scambio e quelle orientate di mescolamento. Le prime, non orientate di scambio (*exchange*), sono tra i processori P_i e P_{i+1}, per ogni i pari, in binario tra le coppie il cui indice differisce del bit meno significativo (quindi a coppie pari-dispari). Le seconde, orientate di mescolamento (shuffle), partono da un processore P_i con $0 \leq i \leq n - 2$ ed arrivano al processore P_j con $\mathbf{j = 2i \bmod (n - 1)}$, e infine una che esce ed entra in P_{n-1}.



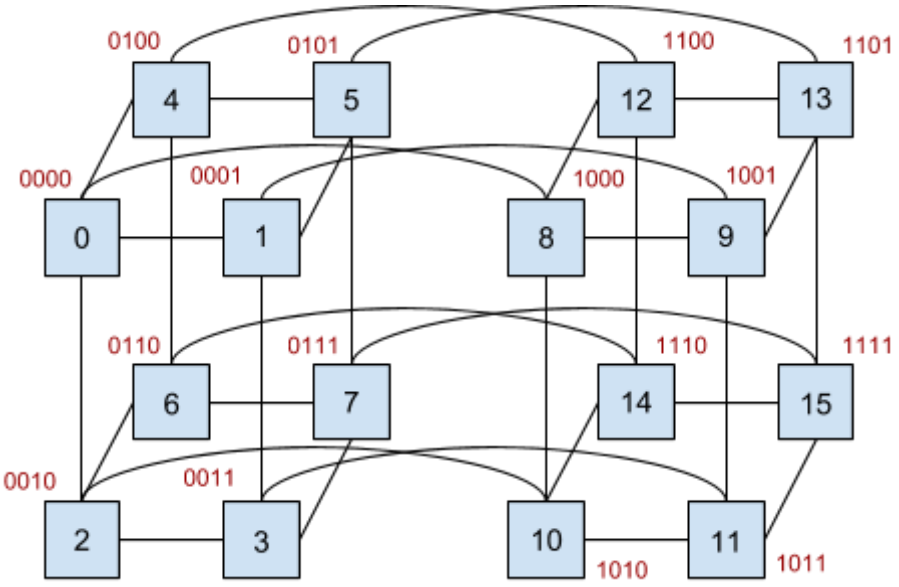
Butterfly

In una butterfly di grado k, gli n processori, con $n = (k + 1)2^k$ per un k intero positivo, sono disposti in k + 1 righe e 2^k colonne. Il generico processore $P_{i,j}$ con $1 \leq i \leq k$, $0 \leq j \leq 2^k - 1$, è connesso con i processori $P_{i-1,j}$ e $P_{i-1,m}$ dove *m è l'intero ottenuto complementando l'i-esimo bit più significativo della rappresentazione binaria di j*. Il grado massimo g di ogni nodo è 4 e il diametro d è O(k) = O(log n).



Ipercubo

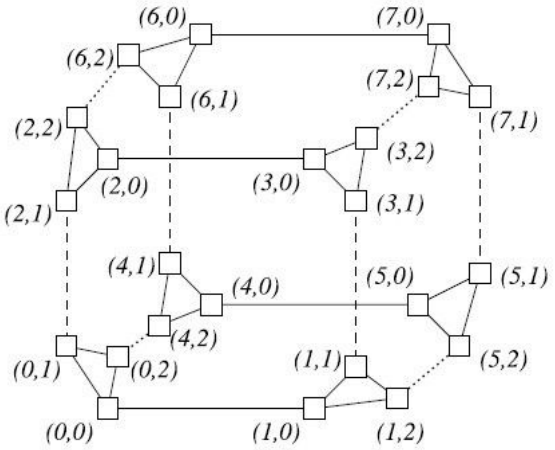
In un ipercubo di dimensione k, gli n = 2^k processori sono numerati da 0 a 2^k - 1 e il generico processore **P_i** è connesso con tutti i processori **P_j** tali che le rappresentazioni binarie di i e j differiscano per il valore di un singolo bit. Il grado massimo g e il diametro d sono entrambi uguali a O(k) = O(log n).



Cube Connected Cycles

Il problema dell'ipercubo è che il grado g non è una costante, ma bensì log n, con questa variante si vuole far si che il grado della struttura sia 3. Consiste in un ipercubo di dimensione k, dove però ognuno dei 2^k vertici dell'ipercubo è sostituito con un circuito di k processori, per un totale di k2^k processori. Si indichi con $P_{i,j}$ l'i-esimo processore del circuito corrispondente al vertice j-esimo dell'ipercubo, con 1 ≤ i ≤ k e 0 ≤ k ≤ 2^k - 1. Il generico processore $P_{i,j}$ è connesso con **$P_{i,m}$** dove *m è il numero intero ottenuto complementando l'i-esimo bit più significativo della*

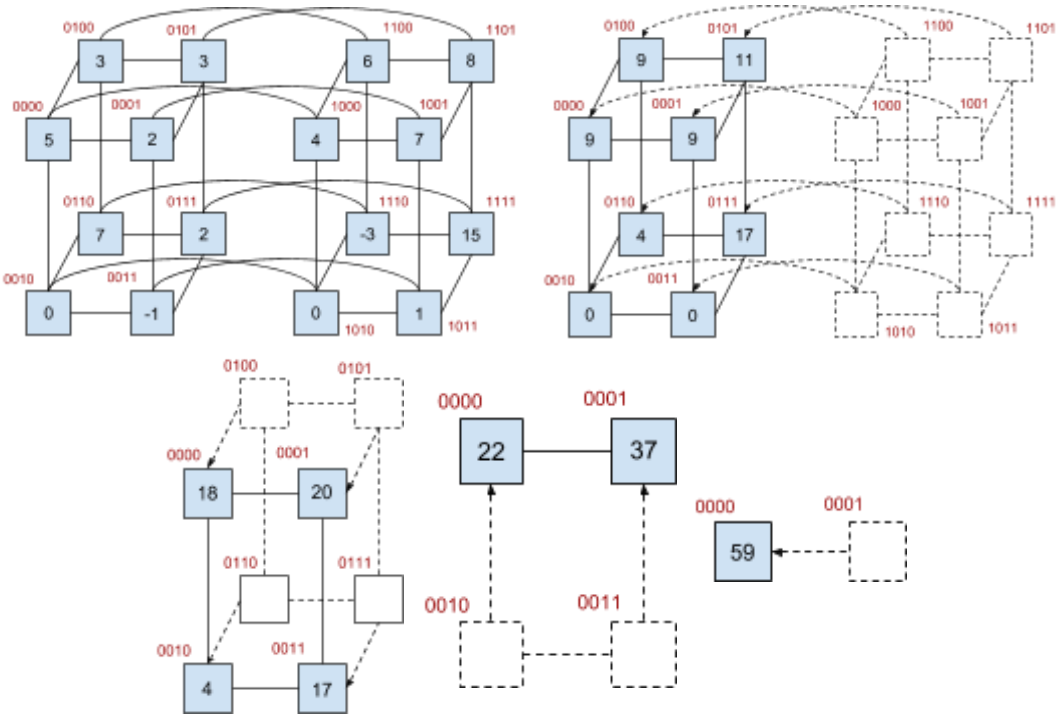
rappresentazione binaria di j. Così facendo il grado massimo g del CCC diventa 3. Si noti l'analogia con la butterfly.



Sommatoria: ipercubo

Sia $n = 2^k$ il numero dei processori e si consideri un ipercubo di dimensione k, in cui il dato a_i sia contenuto nel processore P_i. Concettualmente si considerano prima gli n/2 processori con indici minori, ovvero quelli che hanno il bit più significativo a 0, i quali formano un ipercubo di dimensione k - 1. Ognuno somma in parallelo il suo dato con quello contenuto nel corrispondente processore dell'altro ipercubo, ovvero un processore con il bit più significativo a 0 somma il dato con il processore che ha il bit più significativo a 1. Il procedimento viene ripetuto dimezzando ancora, il processore con i bit più significativi a 00 sommano i processori con i bit più significativi a 01, e così via fino a 0000 che somma 0001. Così facendo dopo k = log n iterazioni nel processore P_0 si avrà la somma di tutti gli elementi.

```
procedure SOMMATORIA-IPERCUBO(a0, a1, . . . , an-1);
  begin
    for k := log n - 1 downto 0 do begin
      for all j where 0 ≤ j ≤ 2^k - 1 do in parallel begin
        bj ← aj+2^k        ← richiedo il valore al processore j + 2^k
        aj := aj + bj
      end;
  end;
end;
```

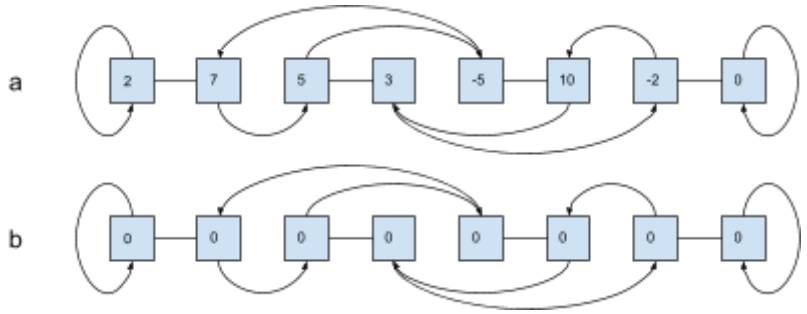


Il tempo richiesto è O(log n) utilizzando O(n) processori con lavoro O(n log n). Per ottenere lavoro ottimo vanno apportati alcuni aggiustamenti, mentre prima si avevano n valori, 1 per processore, bisogna invece avere O(n log n) valori da sommare, divisi in O(log n) valori per ogni processore. Ognuno dei quali somma i propri O(log n) valori sequenzialmente, ma in parallelo con gli altri processori, riducendo la dimensione dell'input ad n. Infine si esegue SOMMATORIA-IPERCUBO, utilizzando sempre lavoro O(n log n) ma con una sommatoria di O(n log n) numeri invece che di solamente n, quindi il lavoro finale risulta essere O(n).

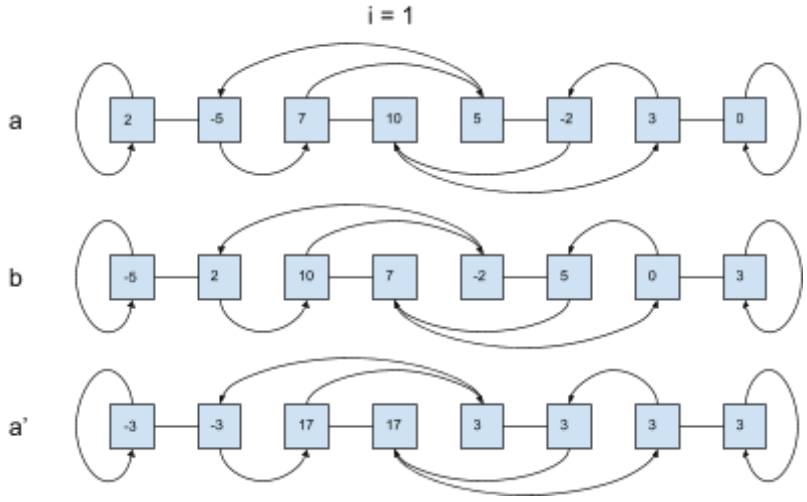
Sommatoria: shuffle

Considerando sempre $n = 2^k$ processori, si introducono due funzioni per la shuffle:
- *SCAMBIA*(a_i): che inverte in tempo O(1) i valori nei processori di indice pari con quelli dei processori di indice dispari, in accordo con le connessioni di scambio di a_i. Ad esempio SCAMBIA(a_2) fa sì che il processore P_2 ottenga il valore a_3 dal processore P_3 e viceversa che a_3 ottenga il valore di a_2 dal processore P_2.
- *MESCOLA*(b_j): che inverte in tempo O(1) i b_j in accordo alle connessioni di mescolamento. Ad esempio *MESCOLA*(b_2) fa sì che il processore P_2 ottenga il valore b_1 dal processore P_1 e viceversa che b_1 ottenga il valore b_2 dal processore P_2.

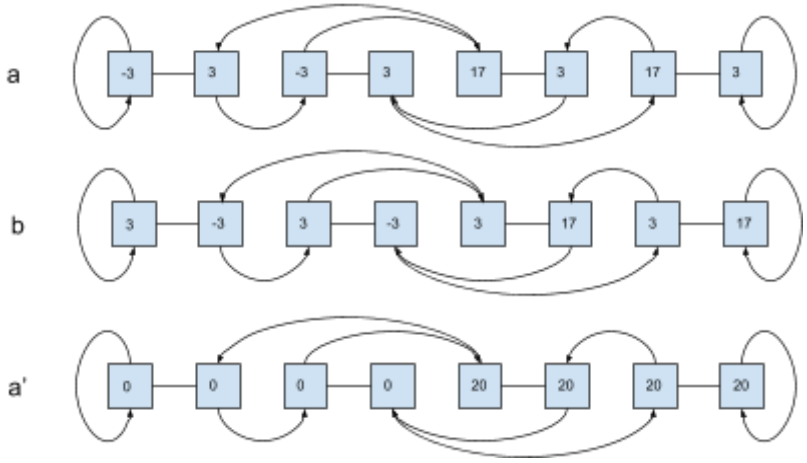
```
procedure SOMMATORIA-SHUFFLE(a_0, a_1, ..., a_{n-1});
  begin
    for i := 1 to log n do
      for all j where 0 ≤ j ≤ n - 1 do in parallel begin
        MESCOLA(a_j);
        b_j := a_j;
        SCAMBIA(b_j);
        a_j := a_j + b_j
      end
  end;
```

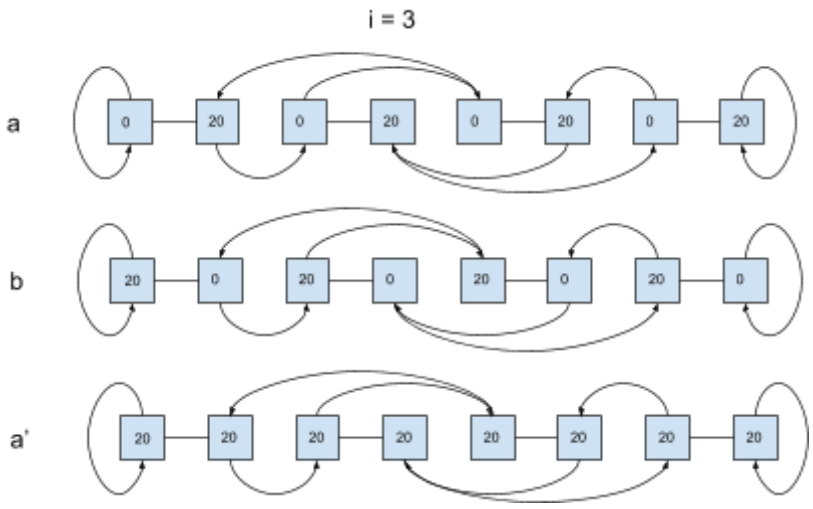


i = 1



i = 2

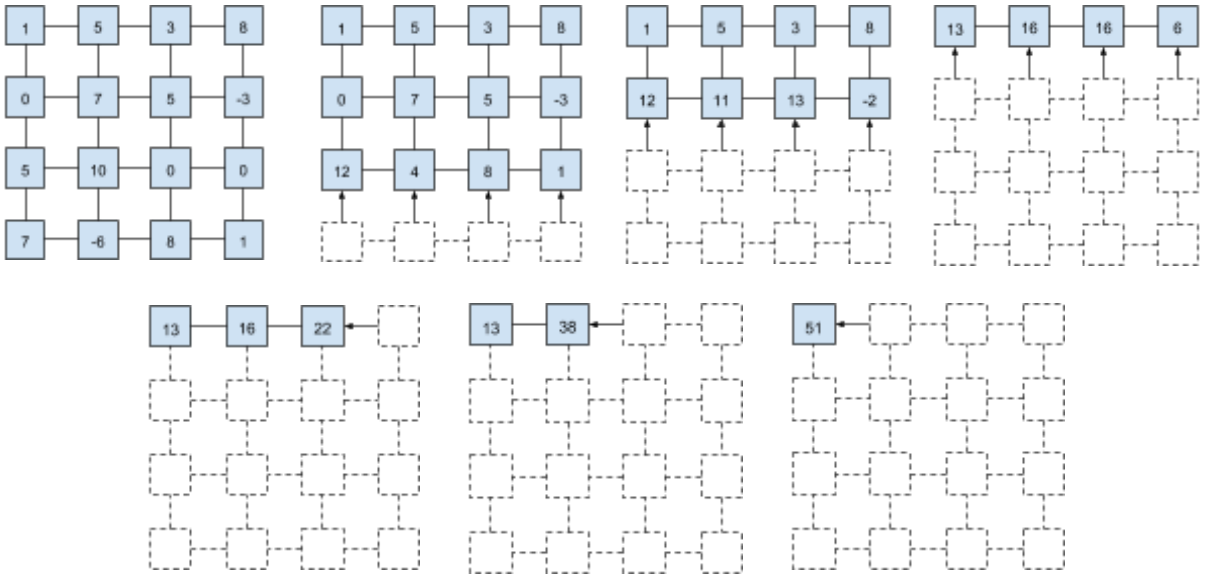




Sommatoria: mesh

L'algoritmo consiste nel sommare gli elementi dell'ultima riga a quelli della penultima riga, poi sommare i risultati agli elementi della riga sopra e così via risalendo fino alla prima riga. Infine i risultati parziali sono sommati da destra verso sinistra, ottenendo il risultato in $P_{1,1}$. Il tempo è $O(\sqrt{n})$ usando n processori.

procedure SOMMATORIA-MESH($a_{1,1}$, $a_{1,2}$, ..., $a_{p,p}$);
  begin
    for i := p - 1 **downto** 1 **do** ← fase 1: somma dal basso verso l'alto
      for all i, j **where** 1 ≤ j ≤ p **do in parallel**
        $b_{i,j} \leftarrow a_{i+1,j}$
        $a_{i,j} := a_{i,j} + b_{i,j}$
    for j := p - 1 **downto** 1 **do** ← fase 2: somma da destra verso sinistra
        $b_{1,j} \leftarrow a_{1,j+1}$
        $a_{1,j} := a_{1,j} + b_{1,j}$
  end;



Per ottenere lavoro ottimo si può usare la tecnica vista precedentemente, si considera un input di dimensione $O(n\sqrt{n})$ con tutti i valori da sommare, si mettono $O(\sqrt{n})$ valori in ogni processore, che verranno sommati sequenzialmente, e infine si applica SOMMATORIA-MESH ottenendo così un lavoro di

Ordinamento: Mergesort bitonico

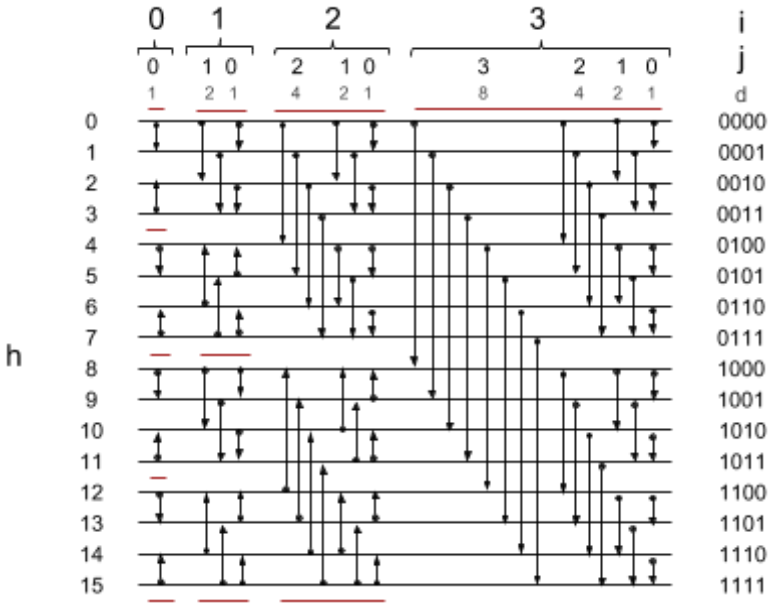
Prima di tutto va introdotto il concetto di **sequenza unimodale**: una sequenza $a_0, \ldots, a_{n-1}$ è detta *unimodale* se esiste un indice i compreso tra 0 ed n-1 tale per cui gli elementi hanno la forma $a_0 \leq a_1 \leq \ldots \leq a_i \geq a_{i+1} \geq \ldots \geq a_{n-1}$.

Da qui una sequenza è detta bitonica se è unimodale, oppure è ottenibile da una sequenza unimodale tramite una traslazione circolare. Es. 2, 5, 7, 8, 10, 3, 1, 0 è una sequenza unimodale e quindi anche bitonica. Una proprietà delle sequenze bitoniche è che: data la sequenza bitonica $A = a_0, a_1, \ldots, a_{2m-1}$ allora le due sequenze (in cui m è la metà della sequenza):

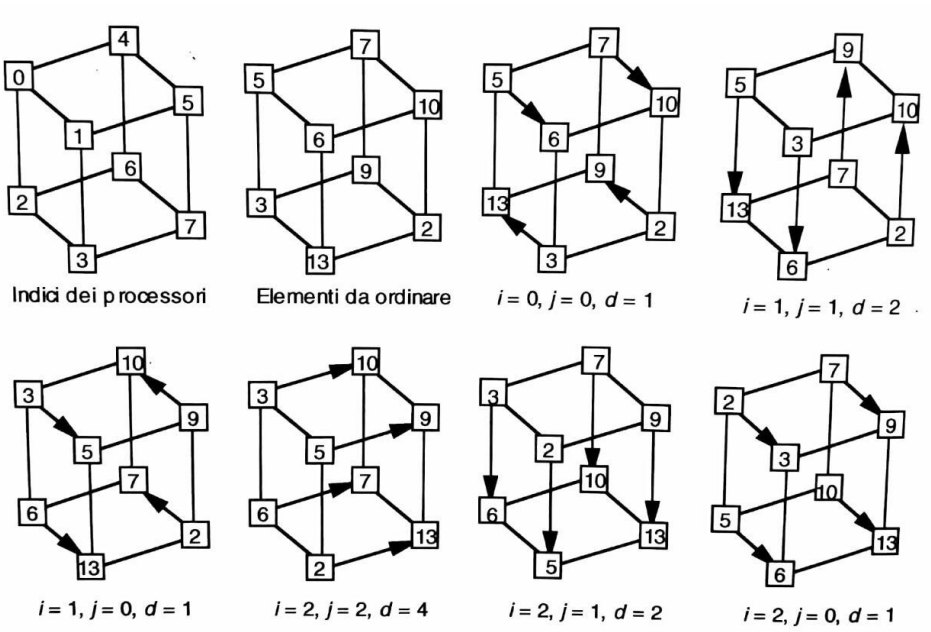
$$A_{MIN} = \min\{a_0, a_m\}, \min\{a_1, a_{m+1}\}, \ldots, \min\{a_{m-1}, a_{2m-1}\}$$
$$A_{MAX} = \max\{a_0, a_m\}, \max\{a_1, a_{m+1}\}, \ldots, \max\{a_{m-1}, a_{2m-1}\}$$

sono entrambe bitoniche, in cui ogni elemento di A_{MIN} è $\leq$ di ogni elemento di A_{MAX}. Questa proprietà può essere utilizzata per un algoritmo divide-et-impera, ordinando una sequenza A lunga $n = 2^k$ dividendola ogni volta nelle due sequenze A_{MIN} e A_{MAX} e applicando il procedimento in parallelo ad esse. Il concetto è che una sequenza A di n elementi può essere vista come n/2 sequenze bitoniche di lunghezza 2, o n/4 sequenze bitoniche di lunghezza 4 e così via finché non si ordina una sequenza bitonica di lunghezza n.

L'algoritmo ha k iterazioni (con $k = \log n$), e per ogni iterazione i per $1 \leq i \leq k$, richiede i passi per ordinare le $n/2^i$ sequenze bitoniche lunghe 2^i. Al j-esimo passo, $1 \leq j \leq i$ sono confrontati e scambiati tra loro elementi le cui posizioni differiscono del j-esimo bit meno significativo. Questo suggerisce un ipercubo di dimensione k. Il diagramma di ordinamento è il seguente:

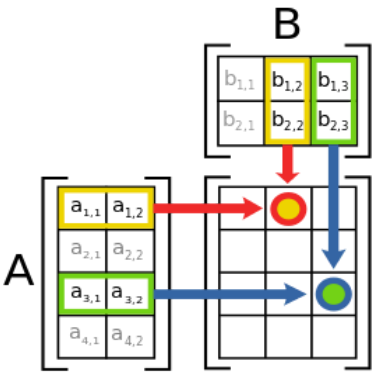


```
procedure BITONIC-MERGESORT-IPERCUBO(a0, a1, ..., an-1);
  begin
    for i := 0 to log(n) - 1 do                          ← log(n) fasi i
      for j := i downto 0 do begin                       ← le sotto-fasi j partono da i fino a 0
        d := 2^j;                                        ← dimensione sequenze bitoniche
        for all h where 0 ≤ h ≤ n - 1 do in parallel begin ← per ogni elemento della sequenza
          if h mod 2d < d then begin                     ← primi d elementi
            t_h ← a_{h+d};                               ← corrispettivo nei secondi d elem.
            if h mod 2^{i+2} < 2^{i+1} then              ← decide il verso del confronto
              b_h := max{a_h, t_h};
              a_h := min{a_h, t_h};
            else
              b_h := min{a_h, t_h};
              a_h := max{a_h, t_h};
            end
          end;
          if h mod 2d ≥ d then                           ← secondi d elementi
            a_h ← b_{h-d}
        end
  end;
```



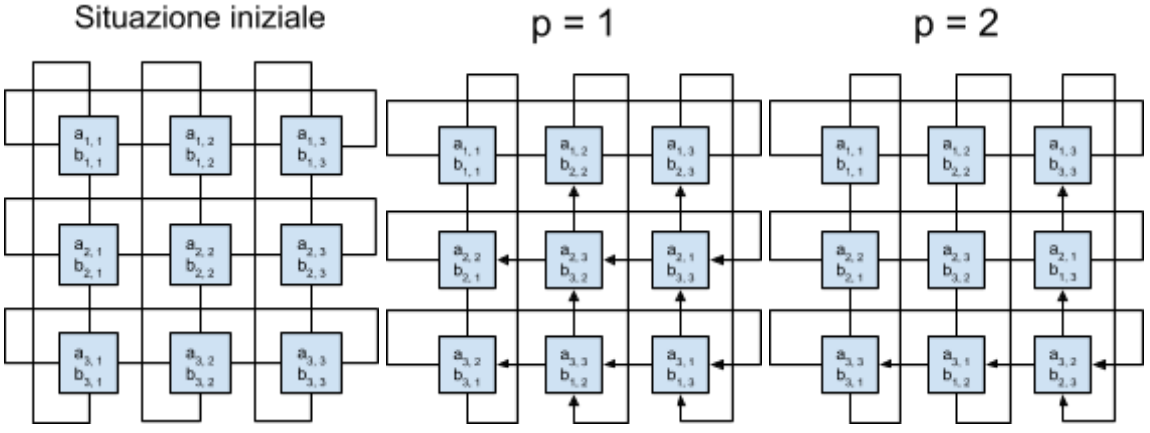
Si noti che k = log n; il controllo ***h mod 2d < d*** serve a decidere quale dei due processori interessati in un confronto prende in carico il lavoro, mentre l'altro processore ***h mod 2d ≥ d*** si limita a prendere il valore opportunamente impostato dal primo. Secondo la scelta impostata è il processore di indice minore a svolgere i calcoli.
Il controllo ***h mod $2^{i+2} < 2^{i+1}$*** serve invece a stabilire il verso del confronto, ovvero dove posizionare il valore minore ed il valore maggiore.
Il tempo è $O(\log^2 n)$ e i processori utilizzati $O(n)$, lavoro $O(n \log^2 n)$.

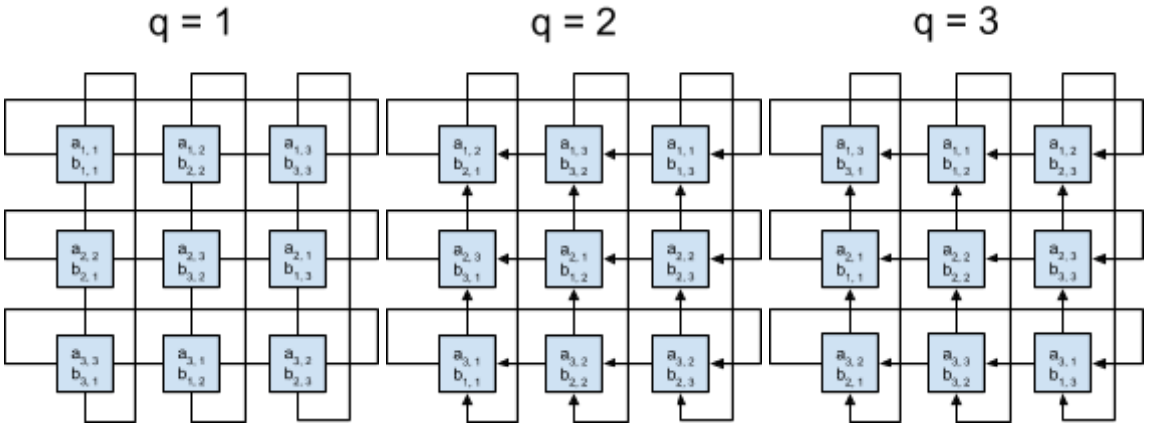
Moltiplicazione di matrici: ***mesh***



La moltiplicazione tra matrici consiste nell'ottenere un valore $c_{i\,j} = \sum_{1 \le k \le n} a_{i\,k}\, b_{k\,j}$.
Si assuma che le due matrici siano memorizzate nella mesh ciclica in modo che il processore $P_{i,j}$ contenga gli elementi $a_{i\,j}$ e $b_{i\,j}$ e al termine della computazione contenga $c_{i\,j}$.
L'algoritmo ha due fasi, nella prima si prepara la matrice: gli elementi della riga i-esima di A sono traslati ciclicamente a sinistra di i - 1 posizioni con $1 \le i \le n$. Mentre gli elementi della colonna j-esima di B sono traslati ciclicamente verso l'alto di j - 1 posizioni con $1 \le j \le n$. In questo modo nella seconda fase gli elementi $a_{i\,k}$ e $b_{k\,j}$ si incontrano per la moltiplicazione. Nella seconda fase invece avvengono le vere e proprie somme dei prodotti.

```
procedure MOLTMATRICI-MESH(A, B, n);
  begin
    for p := 1 to n - 1 do
      for all i, j where 1 ≤ i, j ≤ n do in parallel begin      ← Fase 1
        if i > p then a_{i,j} ← a_{i, j mod n + 1};             ← traslazione della riga i di i - 1 pos.
        if j > p then b_{i,j} ← b_{i mod n + 1, j}              ← traslazione della colonna j di j - 1 pos.
      end;
    for all i, j where 1 ≤ i, j ≤ n do in parallel
      c_{i,j} := 0;
    for q := 1 to n do                                          ← Fase 2
      for all i, j where 1 ≤ i, j ≤ n do in parallel begin
        c_{i,j} := c_{i,j} + a_{i,j} * b_{i,j};
        a_{i,j} ← a_{i,j+1};
        b_{i,j} ← b_{i+1,j}
      end
  end;
```





La procedura richiede tempo O(n) e O(n^2) processori, ottenendo un lavoro di O(n^3).

Moltiplicazione di matrici: ***ipercubo***

$n = 2^2 = 4$
$n^3 = 64$

Gli indici hanno valori compresi tra 0 e $2^{2*3} - 1 = 63$

1 1	1 1	1 1
k	i	j

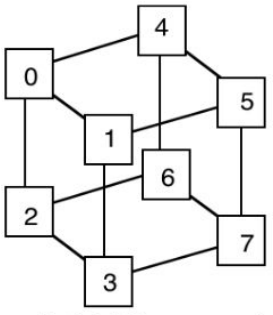
Assumendo sempre una matrice n x n con $n = 2^h$ valori presi in input, si crea un ipercubo di n^3 nodi i cui indici dei processori avranno valori compresi tra 0 e $2^{3h} - 1$ (es. matrice 2x2 si crea un ipercubo di 8 nodi). La loro rappresentazione è costituita da 3h bit, che possono essere visti come 3 blocchi di h bit. Ogni blocco di h bit ha un valore compreso tra 0 e $n - 1 = 2^h - 1$. Quindi è possibile memorizzare distintamente in ognuno dei 3 blocchi di bit tutti i valori presi in input. Con il blocco a sinistra che rappresenta un indice k, quello centrale un indice i, e quello più a destra indice j. Un processore può essere individuato con un unico indice decimale $m = k \cdot 2^{2h} + i \cdot 2^h + j$ in forma (k, i, j). Ciascun processore P_m ha 5 variabili locali: A_m, B_m, C_m, D_m, E_m. Le matrici A e B sono memorizzate sull'ipercubo in modo che il processore $P_m = P_{i\ 2\text{^}h + j} = P_{0, i, j}$ (la prima metà degli elementi dell'ipercubo) contenga l'elemento $a_{i\ j}$ in A_m e $b_{i\ j}$ in B_m e al termine della computazione contenga $c_{i\ j}$ in C_m.

L'algoritmo ha 3 fasi, nella **prima fase** i dati sono distribuiti in tempo O(log n) su tutti i processori, al termine gli elementi $a_{i\ j}$ e $b_{i\ j}$ risultano distribuiti su tutti gli A_m e B_m dei processori $P_m = P_{k\ 2\text{^}(2h) + i\ 2\text{^}h + j} = P_{k, i, j}$ con k tra 0 e n - 1, ugualmente $a_{i\ k}$ è distribuito su tutti gli A_m dei $P_m = P_{k, i, j}$ tali che $0 \le j \le n-1$ e $b_{k\ j}$ è distribuito su tutti i B_m dei $P_m = P_{k, i, j}$ tali che $0 \le i \le n-1$. Quindi P_m contiene $a_{i\ k}$ e $b_{k\ j}$ proprio gli elementi che vanno moltiplicati tra loro. Nella **seconda fase** in tempo O(1) sono effettuati tutti i prodotti, nella terza e **ultima fase** invece sono sommati i prodotti sui processori $P_{k, i, j}$ per k compreso tra 0 e n - 1.

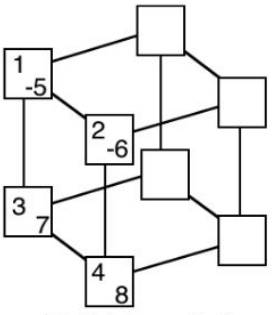
Vengono introdotte 2 funzioni:

- BIT(m, p): che restituisce il (p + 1)-esimo bit meno significativo della rappresentazione binaria dell'intero m.
- COMPLEMENTA(m, p): che restituisce l'intero ottenuto complementando il (p + 1)-esimo bit meno significativo della rappresentazione binaria di m.

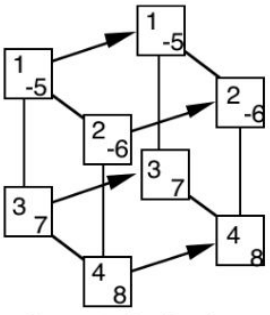
```
procedure MOLTMATRICI-IPERCUBO(A, B, n);
  begin
    for p := 3h - 1 downto 2h do                          ← Fase 1
      for all m where BIT(m, p) = 1 do in parallel        ← Distribuiti su tutti i k, ovvero
        begin                                               copiati nel resto
          D_m := COMPLEMENTA(m, p);                         dell'ipercubo,
          A_m ← A_D_m;                                      complementando un bit
          B_m ← B_D_m;                                      di k alla volta
        end;
    for p := 2h - 1 downto h do                           ← Distribuiti su tutti gli i copi-
      for all m where BIT(m, p) ≠ BIT(m, h + p) do in parallel  andoli sul nodo sopra/sotto
        begin
          D_m := COMPLEMENTA(m, p);
          B_m ← B_D_m;
        end;
    for p := h - 1 downto 0 do                            ← Distribuiti su tutti i j
      for all m where BIT(m, p) ≠ BIT(m, 2h + p) do in parallel  copiandoli sul nodo a lato
        begin
          D_m := COMPLEMENTA(m, p);
          A_m ← A_D_m;
        end;
    for all m where 0 ≤ m ≤ 2^3h - 1 do in parallel       ← Fase 2
      C_m := A_m * B_m;
    for p := 2h to 3h - 1 do                              ← Fase 3
      for all m where 0 ≤ m ≤ 2^3h - 1 do in parallel
        begin
          E_m := COMPLEMENTA(m, p);
          D_m ← C_E_m;
          C_m := C_m + D_m;                               ← somma delle due metà
        end;
  end;
```



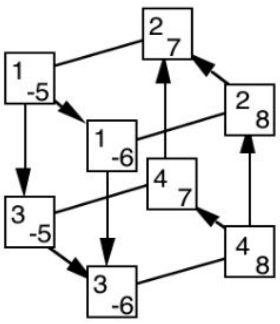
Indici dei processori



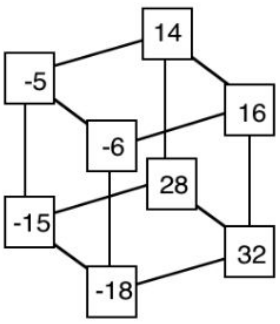
Matrici da moltiplicare



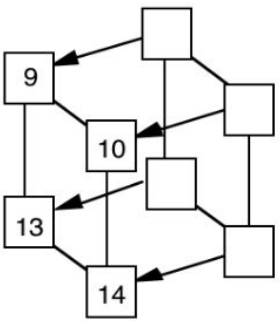
Fase 1: distribuzione su *k*



Fase 1: distribuzione su *i* e *j*



Fase 2: moltiplicazioni



Fase 3: addizioni

Il tempo per questo algoritmo è O(log n), mentre il numero di processori $O(n^3)$ producendo un lavoro di $O(n^3 \log n)$.

Cammini minimi

Per quanto riguarda il problema di trovare un cammino il più corto possibile all'interno di un grafo che va da un nodo ad un altro, è possibile utilizzare l'algoritmo visto in precedenza con piccole modifiche. Un cammino minimo da i a j che comprende al più 2k archi può essere ottenuto da un cammino minimo tra i ed un nodo intermedio h che comprende al più k archi, seguito da un altro cammino che comprende al più k archi. Sia $d_{ij}^{(k)}$ la lunghezza di un cammino minimo da i a j che contiene al più k archi. Vale la seguente relazione:

$$d_{ij}^{(2k)} = \min_{1 \le h \le n}\{d_{ih}^{(k)} + d_{hj}^{(k)}\},$$

dati $1 \le i, j \le n$. In sostanza la lunghezza di un cammino minimo di 2k archi da un nodo i ad un nodo j, altro non è che il cammino minimo tra tutti i cammini che partono da i ed arrivano a un certo nodo intermedio in k passi, più i cammini che da questo nodo poi terminano in j. Questo può essere visto ricorsivamente. Si crea una matrice con condizioni iniziali:

$$d_{ij}^{(1)} = \begin{cases} c_{ij} & \text{se } (i, j) \in A, \\ 0 & \text{se } i = j, \\ \infty & \text{altrimenti.} \end{cases}$$

Per risolvere il problema è sufficiente calcolare $d_{ij}^{(k)}$ iterando per k che va da $2^0=1$ a $2^{\log(n-1)} = n - 1$, cioè raddoppiando ogni volta k finché non raggiunge o supera n - 1. Il calcolo $d_{ij}^{(2k)}$ è analogo a quello di una moltiplicazione di una matrice quadrata $D = [d_{ij}]$ per se stessa dove $d_{ij} = d_{ij}^{(k)}$, k è solo un indice di iterazione e si può ignorare.

Trasformata discreta di Fourier

Partiamo con la definizione di *radice n-esima complessa dell'unità*, ovvero un qualunque numero complesso ω tale che $\omega^n = 1$ (tutti i valori complessi la cui n-esima potenza è pari ad 1). Si tenga a mente che $i = \sqrt{-1}$ è l'unità immaginaria. Di queste radici n-esime complesse dell'unità ve ne sono n, rappresentate per $0 \le k \le n - 1$, come:

- $e^{\frac{2\pi i}{n} \cdot k}$ in forma esponenziale,
- $(1, \frac{2\pi}{n} \cdot k)$ in forma polare, e
- $cos(\frac{2\pi}{n} \cdot k) + i \cdot sin(\frac{2\pi}{n} \cdot k)$ in forma trigonometrica.

La *radice primitiva n-esima dell'unità* è rappresentata da quella radice che genera la radice n-esima dell'unità:

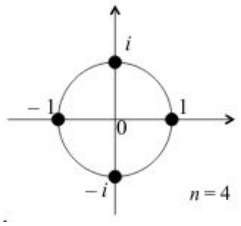
$$\omega_n = e^{\frac{2\pi \imath}{n}} = \cos(\frac{2\pi}{n}) + \imath \sin(\frac{2\pi}{n}) = (1, \frac{2\pi}{n})$$

Da questa si possono ricavare le altre radici n-esime complesse dell'unità come potenza della radice primitiva: $\omega_n^0, \ldots, \omega_n^{n-1}$.

Il *lemma di cancellazione* indica che per qualsiasi intero $n \geq 0$, $k \geq 0$ e $d \geq 0$ si ha che: $\omega_{dn}^{dk} = \omega_n^k$, questo è evidente in quanto n è sempre al denominatore e k sempre al numeratore, quindi un ipotetico d moltiplicato ad entrambi verrebbe semplificato.

Inoltre per qualsiasi n pari e maggiore di 0 si ha che $\omega_n^{n/2} = \omega_2 = -1$.
Il *lemma di dimezzamento* indica che se $n > 0$ e pari allora i quadrati delle n radici complesse dell'unità sono le n/2 radici n/2-esime complesse dell'unità, ovvero $(\omega_n^k)^2\ \omega_n^{2k} = \omega_{n/2}^k$.

Infine una proprietà è $\omega^j \omega^k = \omega^{j+k} = \omega^{(j+k) \bmod n}$, in quanto vi è come un moto circolare, una volta completato il cerchio si ritorna al principio.
A destra un esempio di radici quarte dell'unità (n = 4).



<u>*Trasformata*</u>
Per comodità la dicitura ω_n può essere abbreviata con ω e basta. La trasformata di Fourier prevede che venga fornito un vettore reale $a = (a_0, \ldots, a_{n-1})$, la radice primitiva n-esima dell'unità ω e la funzione $F = [\omega^{ij}]$ per i, j compresi tra 0 e n - 1. Restituisce un vettore complesso b, in cui ogni elemento $\mathbf{b_i = \Sigma_{0 \leq j \leq n-1} a_j (\omega^i)^j}$ con i compreso tra 0 e n - 1. Avendo la funzione:

$$a(x) = \sum_{k=0}^{n-1} a_k x^k$$

Dato il polinomio di grado n - 1 avente come coefficienti le n componenti del vettore a, le componenti del vettore b si ottengono valutando il polinomio a(x) negli n punti $\omega^0, \ldots, \omega^{n-1}$ cioè nelle radici n-esime dell'unità.

<u>Algoritmo</u>
L'algoritmo utilizza un escamotage: esprime il polinomio a(x) di n coefficienti e grado n-1 in due diversi polinomi $a^P(x)$ e $a^D(x)$ che opportunamente combinati formano nuovamente a(x). *$a^P(x)$* contiene i *coefficienti in posizioni pari*, mentre *$a^D(x)$* contiene quelli in *posizioni dispari*. Formalmente sono divisi in:

$$a^P(x) = \sum_{k=0}^{\frac{n}{2}-1} a_{2k} x^k \qquad a^D(x) = \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1} x^k$$

Così facendo è possibile ricostruire $\mathbf{a(x) = a^P(x^2) + x a^D(x^2)}$. Quindi per valutare a(x) nelle n radici n-esime dell'unità $\omega^0, \ldots, \omega^{n-1}$ si riduce a valutare $a^P(x)$ e $a^D(x)$ di grado n/2 nei punti $(\omega^0)^2, \ldots, (\omega^{n-1})^2$ per poi combinare i risultati secondo la forma: $a(x) = a^P(x^2) + x a^D(x^2)$. Per il lemma del dimezzamento gli n valori $(\omega^0)^2, \ldots, (\omega^{n-1})^2$

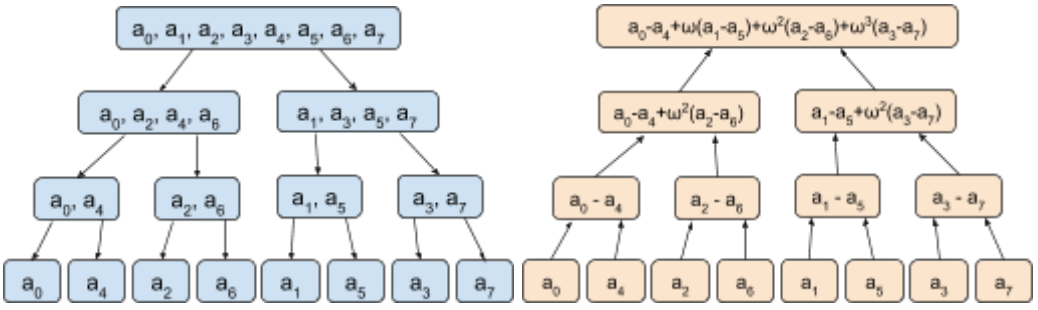
non sono distinti, ma soltanto le n/2 radici n/2-esime dell'unità, quindi i polinomi $a^P(x)$ e $a^D(x)$ sono da valutare ricorsivamente nelle n/2 radici n/2-esime dell'unità.

Esempio:
Sia $a = (a_0, a_1, a_2, a_3)$ quindi n = 4 e $\omega = (1, \pi/2)$. Ricordiamo che $b_i = \sum_{0 \le j \le n-1} a_j(\omega^i)^j$:
- $\mathbf{b_0} = a_0\omega^{0*0} + a_1\omega^{0*1} + a_2\omega^{0*2} + a_3\omega^{0*3} = \mathbf{a_0 + a_1 + a_2 + a_3}$
- $\mathbf{b_1} = a_0\omega^{1*0} + a_1\omega^{1*1} + a_2\omega^{1*2} + a_3\omega^{1*3} = \mathbf{a_0 + a_1\omega - a_2 - a_3\omega}$
- $\mathbf{b_2} = a_0\omega^{2*0} + a_1\omega^{2*1} + a_2\omega^{2*2} + a_3\omega^{2*3} = \mathbf{a_0 - a_1 + a_2 - a_3}$
- $\mathbf{b_3} = a_0\omega^{3*0} + a_1\omega^{3*1} + a_2\omega^{3*2} + a_3\omega^{3*3} = \mathbf{a_0 - a_1\omega - a_2\omega + a_3\omega}$

Sequenzialmente l'algoritmo risulta come segue:

```
procedure FFT(a_0, a_1, a_2, ..., a_{n-1})
  begin
    if n = 1 then
      b_0 := a_0;
    else begin
      a^P := (a_0, a_2, a_4, ..., a_{n-2});
      a^D := (a_1, a_3, a_5, ..., a_{n-1});
      FFT(a^P);
      FFT(a^D);
      ω := cos(2π / #a) + i*sin(2π / #a)        ← radice primitiva n-esima dell'unità
      v := 1;
      for h := 0 to [n / 2] - 1 do begin        ← ricava b da b^P e b^D usando ω^{h+n/2} = -ω^h
        b_h := b_h^P + vb_h^D;
        b_{h+n/2} := b_h^P - vb_h^D;
        v := ωv;
      end;
    end;
  end;
end;
```

Di seguito il calcolo del valore $\mathbf{b_1}$ del vettore complesso b:



Il costo computazionale è facilmente calcolabile dato che viene dimezzato ogni volta l'input: $2T(n/2) + O(n)$, il tempo risulta $O(n \log n)$.

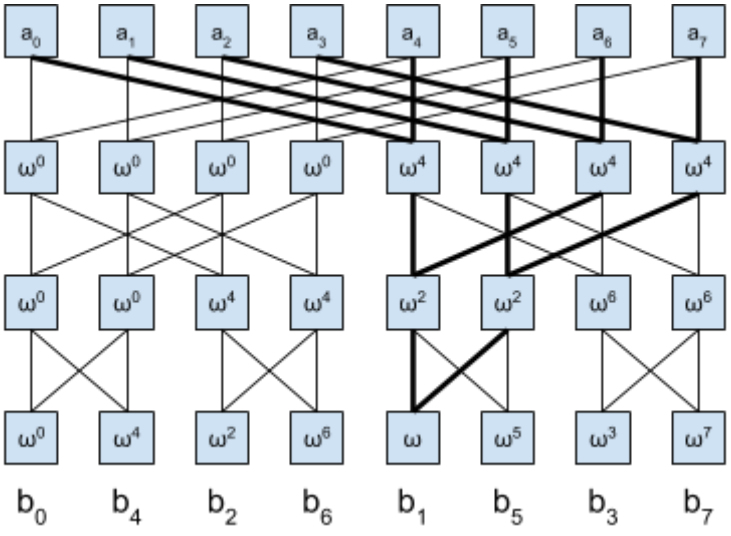
Per calcolare l'intera trasformata di Fourier, si osservi il processo di ricombinazione nella figura soprastante. Se si potessero accoppiare gli elementi da sottrarre (a_0, a_4), (a_2, a_6), (a_1, a_5), (a_3, a_7) allora si potrebbero calcolare direttamente le quattro trasformate. Se poi vengono accoppiate opportunamente questi quattro valori si possono calcolare le due trasformate al livello 1 dell'albero e da questi l'intera trasformata. La computazione può procedere quindi dal basso verso l'alto iterativamente, uno schema che implementa bene questa struttura è la butterfly.

Trasformata di Fourier: Butterfly

```
procedure FFT-BUTTERFLY(a, b, n);
  begin
    k := log n;
    for all j where 0 ≤ j ≤ n - 1 do in parallel        ← popolo la prima riga della butterfly
      r_{0, j} := a[j];
    for i := 0 to k - 1 do                              ← per ogni riga della butterfly
      for all j where 0 ≤ j ≤ n - 1 do in parallel      ← per ogni elemento di ogni riga
        r_{i + 1, j} := r_{i, BIT0(j, i)} + r_{i, BIT1(j, i)} * OMEGA(i, j, k);   ← calcolo la trasformata di 2 elem
    for all j where 0 ≤ j ≤ n - 1 do in parallel        ← ottengo il b parziale
      b_j := r_{k, INVERTI(j, k)}
  end;
```

Le seguenti funzioni sono usate:

- BIT0(j, i): restituisce l'intero ottenuto ponendo a 0 l'i-esimo bit più significativo della rappresentazione binaria di j.
- BIT1(j, i): restituisce l'intero ottenuto ponendo a 1 l'i-esimo bit più significativo della rappresentazione binaria di j.
- OMEGA(i, j, k): restituisce ω^m, dove m è ottenuto a partire dalla rappresentazione binaria a k bit di j, invertendo l'ordine degli i + 1 bit più significativi e ponendo a 0 i restanti bit (k - i - 1).
- INVERTI(j, k): restituisce l'intero ottenuto invertendo l'ordine di tutti i k bit della rappresentazione binaria di j.



Il tempo è O(log n) usando O(n log n) processori, con lavoro O(n $\log^2$ n). In grassetto il processo di calcolo per b_1 come negli schemi precedenti. La computazione procede dall'alto verso il basso, i processori dalla riga 1 in poi ricevono in parallelo due dati, *x* e *y* dai processori della riga precedente, ed effettuano l'operazione $x + \omega^h y$, per un'opportuna potenza di ω.

Trasformata di Fourier: Ipercubo

Si mettono i valori del vettore reale a nell'ipercubo, dal processore 0 al processore n - 1, in questo modo ogni processore si trova già collegato a quelli da cui ricevere i valori opportuni.

```
procedure FFT-IPERCUBO(a);
  begin
    k := log n;
    for all j where 0 ≤ j ≤ n - 1 do in parallel        ← b array d'appoggio per a
      b_j := a_j;
    for i := 0 to k - 1 do
      for all j where 0 ≤ j ≤ n - 1 do in parallel begin        ← per tutti gli n elementi
        x_j := b_j;
        d_j := COMPLEMENTA(j, k - 1 - i);
        if d_j = BIT0(j, k - 1 - i)
           then x_j ← b_d_j
          else x_j := b_j
        if d_j = BIT1(j, k - 1 - i)
           then y_j ← b_d_j
          else y_j := b_j
        b_j := x_j + OMEGA(i, j, k) * y_j;
      end;
  end;
```

Il tempo impiegato è O(log n) con un numero di processori O(n) quindi il lavoro risulta essere O(n log n).

Moltiplicazione di polinomi

Dati due polinomi:
$p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1x + p_0$
$q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \ldots + q_1x + q_0$
ciascuno avente n coefficienti e di grado n - 1, calcolare i polinomio prodotto r(x) = p(x)q(x) e 2n coefficienti e di grado 2n - 1. Per abbassare la complessità è necessario rappresentare i polinomi non per mezzo dei loro coefficienti, ma tramite i valori che assumono quando sono valutati in n punti distinti qualsiasi, ad esempio dato un polinomio $a(x) = x^3 - 3x^2 + 2x - 7$ di grado 3 si calcolano 4 punti a scelta: a(1) = -7, a(2) = -7, a(3) = -10, a(4) = 1. L'operazione inversa alla **valutazione** è chiamata **interpolazione** e date n coppie di valori $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ si ricavano i coefficienti del polinomio a(x) tali che $y_k = a(x_k)$.

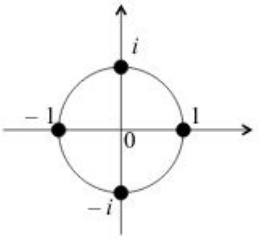
Date le rappresentazioni per punti $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ per p(x) e $(x_0, v_0), (x_1, v_1), \ldots, (x_{n-1}, v_{n-1})$ per q(x), allora $(x_0, y_0 + v_0), (x_1, y_1 + v_1), \ldots, (x_{n-1}, y_{n-1} + v_{n-1})$ è una rappresentazione per punti di s(x) = p(x) + q(x). Allo stesso modo, utilizzando ovviamente 2n punti invece di n, date le rappresentazioni per punti di $(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})$ e $(x_0, v_0), (x_1, v_1), \ldots, (x_{2n-1}, v_{2n-1})$ di p(x) e q(x) allora $(x_0, y_0v_0), (x_1, y_1v_1), \ldots, (x_{2n-1}, y_{2n-1}v_{2n-1})$ è la rappresentazione per punti di r(x) = p(x)q(x).

Quindi per valutare p(x) e q(x) *occorrono 2n punti*, e in linea di principio andrebbero bene qualsiasi 2n punti distinti, ma *se questi fossero le 2n radici complesse 2n-esime dell'unità, allora la valutazione di p(x) e q(x) può essere svolta contemporaneamente in tutti i 2n punti utilizzando la trasformata di Fourier*.

Una volta trovati i punti che rappresentano r(x), l'operazione di interpolazione che permette di ricavare la rappresentazione con coefficienti di r(x) è l'**antitrasformata di Fourier**. Il procedimento è:
1. Si *valutano* p(x) e q(x) nelle 2n radici complesse 2n-esime dell'unità, calcolando le trasformate di Fourier dei vettori espansi $(p_0, \ldots, p_{2n-1})$ e $(q_0, \ldots, q_{2n-1})$.
2. Si eseguono i prodotti $p(\omega^j)q(\omega^j) = r(\omega^j)$ per j compreso tra 0 e 2n - 1.
3. Si calcola l'antitrasformata di Fourier per *interpolare* e ricavare i coefficienti $(r_0, r_1, \ldots, r_{2n-1})$ del polinomio prodotto partendo dai punti di r(x) calcolati.
Il tempo è ancora O(log n) se si utilizza un ipercubo, e sempre O(n) processori portando ad un lavoro di O(n log n).

Antitrasformata di Fourier
Il problema consiste, dato un vettore complesso $b = (b_0, b_1, \ldots, b_{n-1})$, calcolare il vettore reale $a = (a_0, a_1, \ldots, a_{n-1})$ tale che b sia la trasformata di a.
La formula è $\mathbf{a_i = 1/n \sum_{0 \le j \le n-1} b_j (\omega^{-i})^j}$. L'algoritmo è simile a quello della trasformata per quanto riguarda le prestazioni.

Esempio
Siano $\mathbf{p(x) = 3x - 2}$ e $\mathbf{q(x) = 4x + 1}$;
quindi n = 2 e 2n = 4.
Le radici quarte dell'unità sono:
$\omega^0 = 1$ $\omega^1 = i$ $\omega^2 = -1$ $\omega^3 = -i$



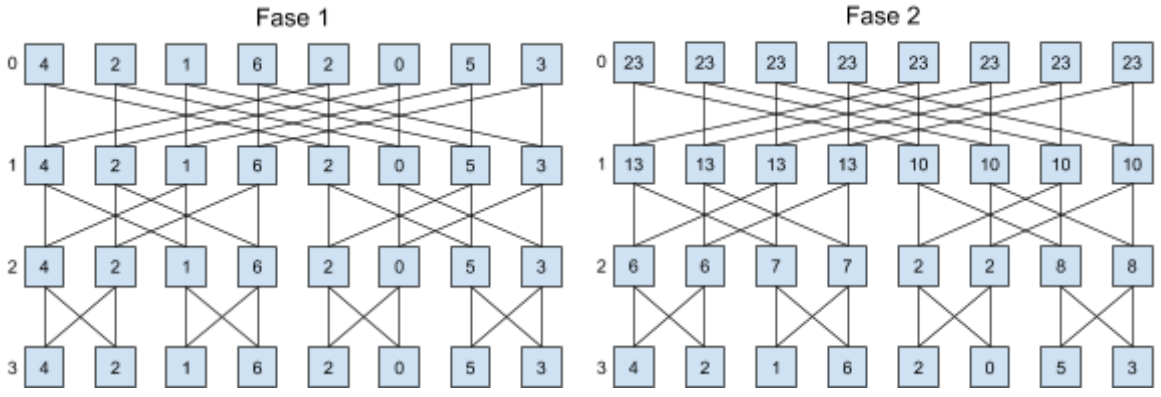
1. La valutazione dei polinomi tramite trasformata di Fourier è:
p(1) = 1 q(1) = 5
p(i) = -2 + 3i q(i) = 1 + 4i
p(-1) = -5 q(-1) = -3
p(-i) = -2 -3i q(-i) = 1 - 4i

2. Quindi i prodotti dei valori dei polinomi nelle radici dell'unità sono:
p(1)q(1) = 5
p(i)q(i) = - 14 - 5i
p(-1)q(-1) = 15
p(-i)q(-i) = -14 + 5i

3. Interpolando si ricava l'antitrasformata di Fourier, con la formula:
$r_k = ¼ [5\omega^0 + (-14 - 5i)\omega^{-k} + 15\omega^{-2k} + (-14 + 5i)\omega^{-3k}]$ per $0 \le k \le 3$
$r_0 = -2$ $r_1 = -5$ $r_2 = 12$ $r_3 = 0$
ovvero $r(x) = 12x^2 - 5x - 2$.

Esercizio: sommatoria su butterfly
Siano dati $n = 2^k$ dati da sommare, l'algoritmo si divide in due fasi, la prima copia gli elementi posizionati sulla linea 0 seguendo le colonne, la seconda fase somma gli elementi dal basso verso l'alto seguendo le connessioni della butterfly, portando sulla riga 0 il risultato.

procedure SOMMATORIA-BUTTERFLY($a_{0,0}, a_{0,1}, \ldots, a_{k,n-1}$)
begin
for i := 1 **to** k **do**
for all j **where** $0 \le j \le n-1$ **do in parallel begin** ← copia tutti i numeri nelle celle
$b_{i,j} \leftarrow a_{i-1,j}$;
$a_{i,j} := a_{i,j} + b_{i,j}$
end
for i := k− 1 **downto** 0 **do**
for all j **where** $0 \le j \le n-1$ **do in parallel begin** ← somma in base alle connessioni
$b_{i,j} \leftarrow a_{i+1,j}$;
$c_{i,j} \leftarrow a_{i+1,\ \textbf{BITCOMPLEMENTA}(i+1,j)}$;
$a_{i,j} := b_{i,j} + c_{i,j}$
end
end



Il tempo è O(k) ovvero O(log n), mentre il numero di processori O(n log n) con un tempo totale di $O(n \log^2 n)$. Il lavoro ottimo è ottenibile utilizzando $O(nk^2)$ dati da sommare, facendone gestire O(k) ad ogni processore.

Esercizio: matrice trasposta su shuffle

Sia data una matrice $A = [a_{ij}]$ di dimensione nxn, in cui $n = 2^q$, e sia P un array di dimensione n^2. L'elemento a_{ij} è memorizzato in P_k dove $k = 2^q(i - 1) + j - 1$, trasponendo la matrice a_{ij} va in P_h dove $h = 2^q(j - 1) + i - 1$.

```
procedure TRASPOSTA-SHUFFLE(a_0, a_1, ..., a_{n^2-1})
  begin
    for all k where 0 ≤ k ≤ n^2 – 1 do in parallel        ← copia a in b
      b_k := a_k
    for l := 1 to q do
      for all k where 0 ≤ k ≤ n^2 – 1 do in parallel     ← mescolamento elementi b per q volte
        MESCOLA(b_k)
  end
```

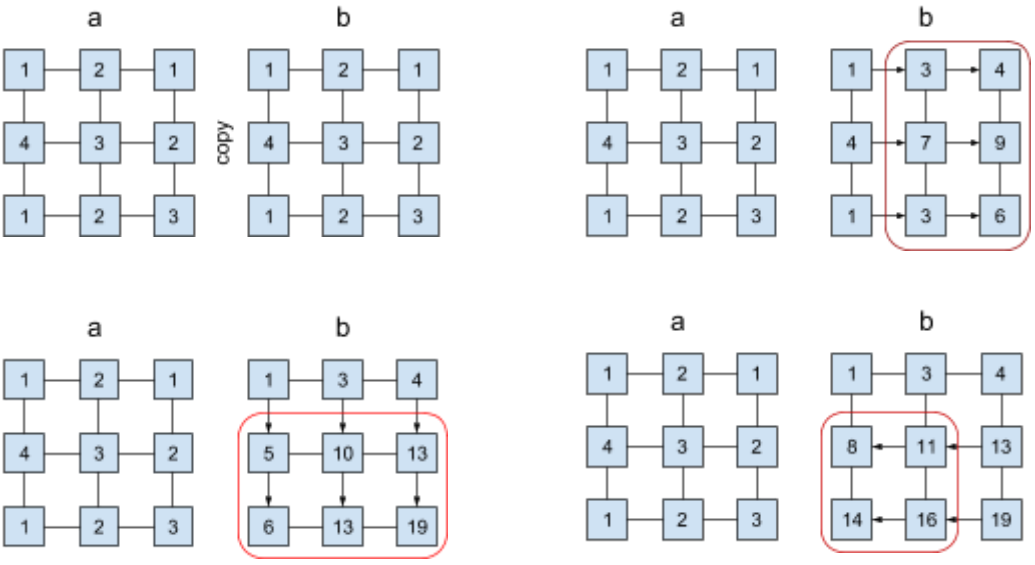
Il tempo impiegato è $O(\log n)$ e il numero di processori $O(n^2)$ fornendo un lavoro di $O(n^2 \log n)$.

Esercizio: somme prefisse su mesh

Il problema delle somme prefisse su una mesh. L'algoritmo procede in 3 fasi dopo aver copiato la matrice a in una di appoggio b. Prima vengono memorizzati gli elementi in ogni cella di b come la somma tra il valore alla sinistra più il valore originario in a. Poi vengono memorizzati gli elementi in ogni cella di b come la somma tra il valore sopra più il valore in b. Ed infine in ogni cella di b viene inserita la differenza tra il b di sinistra e l'a di sinistra.

```
procedure SOMME-PREFISSE-MESH(a_{1,1}, a_{1,2}, ..., a_{n,n})
  begin
    for all i, j where 1 ≤ i ≤ n, j = 1 do in parallel      ← copia a in b
      b_{i,j} :=a_{i,j}
    for j := 2 to n do
      for all i where 1 ≤ i ≤ n do in parallel begin        ← somma elem riga vicini
        d_{i,j} ← b_{i,j–1};
        b_{i,j} := d_{i,j} + a_{i,j}
      end
    for i := 2 to n do
      for all j where j = n do in parallel begin            ← somma elem colonna vicini
        d_{i,j} ← b_{i–1,j};
        b_{i,j} := d_{i,j} + b_{i,j}
      end
    for j := n – 1 down to 1 do
      for all i where 2 ≤ i ≤ n do in parallel begin        ← sottrai valore originario a b
        d_{i,j} ← b_{i,j+1};
        e_{i,j} ← a_{i,j+1};
        b_{i,j} := d_{i,j} – e_{i,j}
      end
```

end



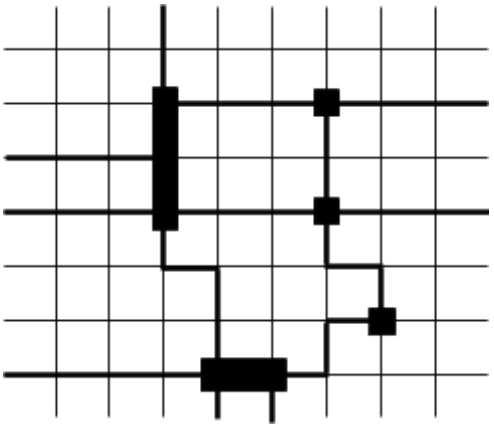
Il tempo è O(n) mentre il numero di processori $O(n^2)$ portando ad un lavoro di $O(n^3)$.

Algoritmi VLSI

(processori sincroni - memoria locale)

La tecnologia Very-Large-Scale-Integration (VLSI) consente di realizzare circuiti molto grandi e complessi su piastrine estremamente piccole. Grazie a questa tecnologia è possibile realizzare circuiti specializzati nella risoluzione di un singolo problema computazionale, gli algoritmi sono sia descritti che realizzati direttamente con circuiti hardware. Prima di tutto questi algoritmi operano su rappresentazioni binarie (e non decimali) dei dati, poi il **tempo T** richiesto dalla computazione, tende ad essere dominato più dalla necessità di distribuire i dati all'interno del circuito che dall'esecuzione di istruzioni aritmetico/logiche. Fondamentale è anche considerare l'**area A** richiesta dal circuito, in quanto il costo del circuito cresce proporzionalmente all'area. Una misura che lega insieme l'area A ed il tempo T è data dal prodotto $\mathbf{AT^2}$, in quanto risulta proporzionale all'**energia dissipata** dal circuito durante la computazione, l'equivalente del "lavoro" per algoritmi PRAM e a grado limitato.
Quando si progetta un algoritmo VLSI se ne definisce un grafo, chiamato "grafo della computazione", dove i nodi corrispondono ad unità di elaborazione e gli archi a connessioni. Ogni nodo mantiene una memoria minima (flip-flop, salva lo stato su un singolo bit e registri, insiemi di flip-flop che possono quindi memorizzare più bit) per ogni unità di elaborazione. In questo grafo i dati entrano da certi nodi (porte d'ingresso), transitano sugli archi attraversando le unità di elaborazione finché giungono ad altri nodi (porte d'uscita) che trasmettono il risultato all'esterno. Un aspetto fondamentale è trovare il layout di area minima per il grafo della computazione. Tipici grafi della computazione sono alberi binari, mesh, e in genere tutti i grafi a grado limitato da una costante.

Nel modello a griglia si considera un circuito VLSI su una griglia rettangolare sulla quale scorrono dei fili. Si assume la griglia abbia solo due livelli: su uno scorrono i fili verticali e sull'altro quelli orizzontali. *Due fili distinti possono scorrere su due linee di griglia che si incrociano, ma non possono sovrapporsi sulla stessa linea di griglia*. I processori sono unità di elaborazione specializzate per calcolare semplici operazioni aritmetico/logiche e si trovano agli incroci delle linee di griglia. L'*area A* del circuito è data dal prodotto tra il numero di linee di griglia verticali (base) ed il numero di linee di griglia orizzontali (altezza). Il circuito è sincrono e il *tempo T* di computazione, misurato in unità è uguale al numero di unità di tempo che intercorrono tra l'invio del primo segnale di ingresso e la produzione dell'ultimo segnale d'uscita. Il tempo necessario per trasmettere un segnale tra i due estremi di un filo è O(1), e il numero minimo di unità di tempo necessarie ad un segnale per raggiungere n unità di elaborazione è $\Omega(\log n)$, raggiungibile con un circuito ad albero binario.



Solido spazio/tempo

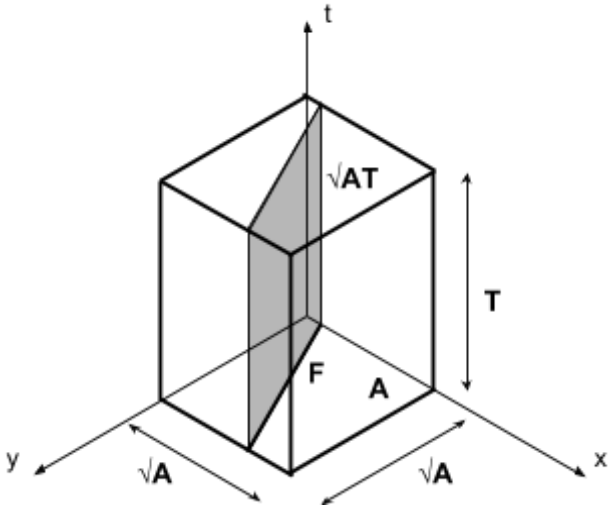
Questo solido è utile a comprendere l'evolversi nel tempo della computazione nel circuito.

- Intersecando il solido con un piano orizzontale, con valore della terza coordinata pari a t, si ottiene un rettangolo corrispondente allo stato del circuito al tempo t. L'area A del circuito non può essere inferiore al massimo numero di bit che il circuito deve ricordare all'istante di tempo t, con t compreso tra 1 e T allora $\mathbf{A \geq \max_{1 \leq t \leq T}}$. Ovvero se un circuito ha al massimo 4 elementi da portare in contemporanea ad un certo tempo t, chiaramente vi dev'essere un'area del circuito di almeno 4 (es. 4x1 o 2x2), per far si che vi sia almeno un bit in ogni segmento.

- Ad ogni istante di tempo t, il circuito può leggere un numero di bit al massimo uguale alla propria area. Il volume V del solido non può essere inferiore al numero di bit che il circuito deve ricevere complessivamente in input. Poiché V = AT, se il circuito deve leggere n bit di input (o emetterne n in output), allora $\mathbf{V = AT \geq n}$.

- Intersecando il solido con un piano verticale si ottiene un rettangolo di area √A * T che partiziona il circuito in due parti, divise da una frontiera F. Le due parti del circuito dovranno scambiarsi dati attraverso F per calcolare il risultato del problema. La "quantità di informazione" che passa complessivamente attraverso la frontiera F non può essere maggiore di √A * T. Se la quantità di informazione vale almeno I, allora $\mathbf{AT^2 \geq I^2}$.

- Infine il tempo totale di computazione T sarà sempre maggiore o uguale al logaritmo del numero dei bit di input dai quali dipende un bit di output, infatti il modo più veloce per attraversare il sistema è logaritmico, quindi se b è il numero dei bit di input dai quali dipende un bit di output, allora **T ≥ log b**.



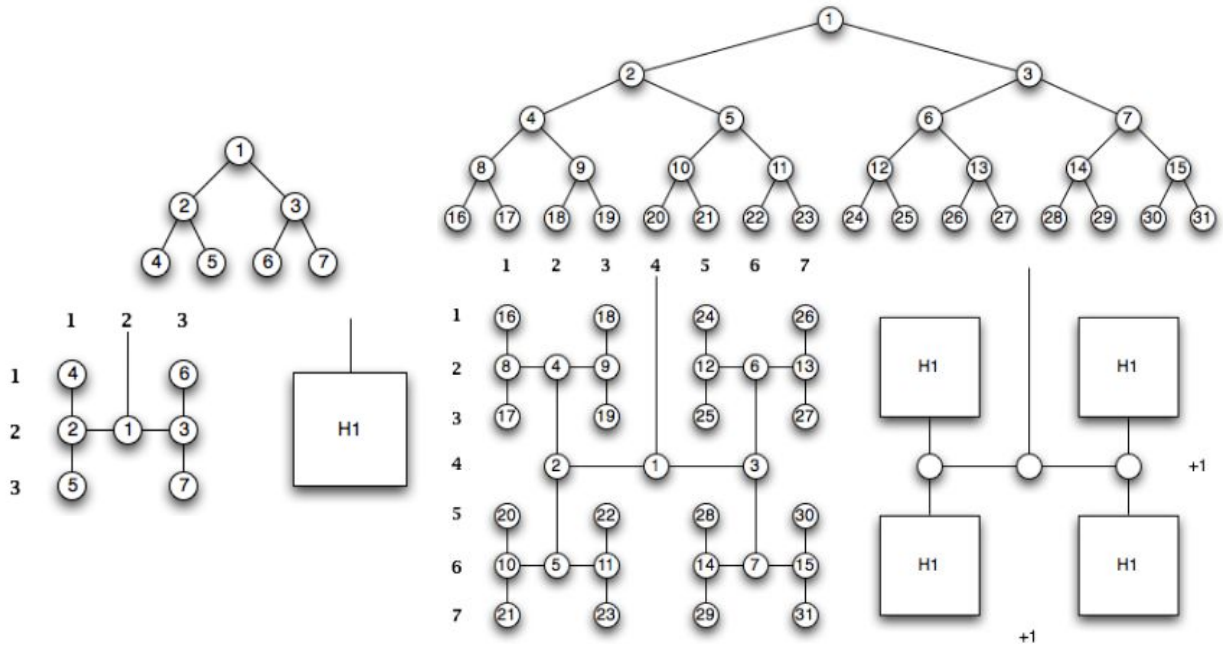
Il circuito è *sincrono* ed il tempo T di computazione è uguale al numero di unità di tempo che intercorrono tra l'invio del primo segnale di ingresso e la produzione dell'ultimo segnale di uscita. Il tempo di trasmissione di un segnale tra i due estremi di un filo è O(1) indipendentemente dalla lunghezza del filo, ma il numero di unità di elaborazione che un segnale può attraversare in un'unità di tempo è 1.

Layout: Alberi

I layout per alberi binari sono fondamentalmente due: uno presenta tutte le foglie dell'albero lungo i bordi ed ha un'area A pari a O(n log n), utile in quanto spesso le foglie sono collegate con l'esterno e vanno quindi posizionate sui bordi; l'altro è il cosiddetto "layout ad H" per la disposizione dei nodi, ed ha un'area A pari a O(n).

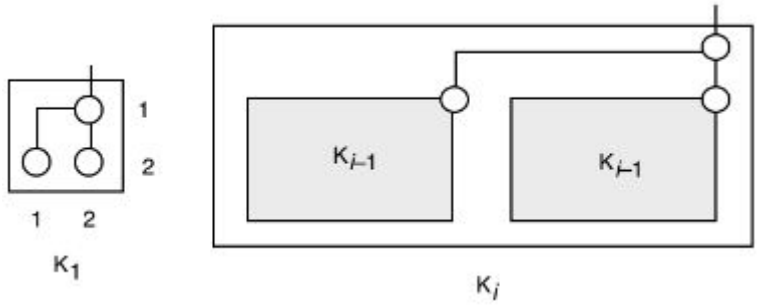
<u>Layout ad H</u>

Il layout ad H prevede che il numero delle foglie sia una potenza pari di 2, ovvero $n = 2^{2i}$. Ad esempio, per un albero con al massimo 4 foglie il layout è quello a sinistra. Il layout H è per natura ricorsivo, infatti per ottenere il layout H di dimensione successiva, ovvero con 16 foglie ($2^{2*\mathbf{1}} = 4$; $2^{2*\mathbf{2}} = 16$; $2^{2*\mathbf{3}} = 64$; . . .), si compongono opportunamente 4 layout del livello precedente H_1, ottenendo il layout a destra:



Viene aggiunta una riga per memorizzare la radice e i suoi figli, ed una colonna per collegare la radice con l'esterno. Sia L(i) il lato di H_i le relazioni di ricorrenza sono:
L(n) = 3 se n = 4
L(n) = 2L(n/4) + 1 se n > 4
Risolvendo con il master theorem si ottiene, α = log 2 / log 4 = ½ che è maggiore di β = 0, quindi O(√n). Sia n il numero di foglie dell'albero ed essendo l'area il quadrato del lato si ha che $A = L^2(n) = O(n)$.

<u>Layout foglie sul bordo</u>
Questo layout è comodo quando si necessita che tutte le foglie siano sul bordo del layout. Anche questo ha natura ricorsiva, sia $n = 2^j$ il numero di foglie dell'albero, si può vedere il layout a sinistra nel caso in cui n sia uguale a 2, nell'immagine a destra in cui sia maggiore:



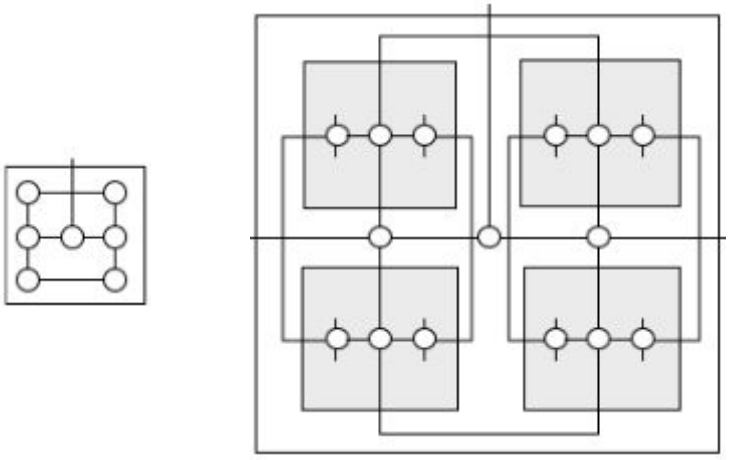
Viene aggiunta una riga per la memorizzazione della radice mentre il numero di colonne rimane invariato. Sia H(i) l'altezza di K_j, mentre W(i) la larghezza, le relazioni di ricorrenza sono:

$H(n) = 2$ se $n = 2$
$H(n) = H(n/2) + 1$ se $n > 2$
Master Th.: $\alpha = \log 1 / \log 2 = 0$; $\beta = 0$; $\alpha = \beta$ quindi $O(n^0 \log n) \rightarrow H(n) = O(\log n)$;

$W(n) = 2$ se $n = 2$
$W(n) = 2W(n/2)$ se $n > 2$
Master Th.: $\alpha = \log 2 / \log 2 = 1$; $\beta = 0$; $\alpha > \beta$ quindi $O(n^1) \rightarrow W(n) = O(n)$;

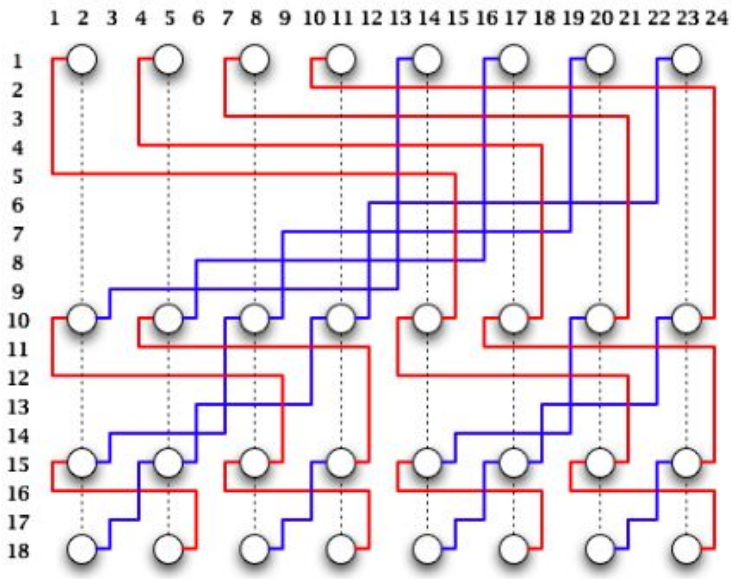
L'area risulta essere quindi: $A = W(n)H(n) = O(n)O(\log n) = O(n \log n)$.

<u>Layout ad H fault tolerant</u>
Al layout ad H può essere apportata una modifica per renderlo tollerante ai guasti: è possibile inserire collegamenti ridondanti per aggirare processori guasti, ad esempio collegando coppie di cugini (entrambi dx o sx). Di seguito i layout:



Le relazioni di ricorrenza diventano quindi:
$L(n) = 3$ se $n = 4$
$L(n) = 2L(n/4) + 5$ se $n > 4$
Master Th.: $\alpha = \log 2 / \log 4 = ½$; $\beta = 0$; $\alpha > \beta$ quindi $O(n^{1/2}) \rightarrow L(n) = O(\sqrt{n})$; E quindi l'area $A = L^2(n) = O(n)$.

Layout: Butterfly
Il circuito per una butterfly di grado k è ottenuto da due butterfly di rando k - 1 con l'aggiunta di opportune connessioni, quindi la struttura è ricorsiva. La convenzione adottata è la seguente:



Le righe tratteggiate, le linee rosse e quelle blu sono tutte connessioni. Per disegnarla conviene partire dal basso disegnando una sotto-butterfly di rango 1 e poi risalendo fino al rango scelto. Per ogni butterfly vanno connessi i nodi inferiori a quelli superiori a partire da sinistra, costruendo a mano a mano gradini di altezza crescente verso destra fino a metà e gradini di altezza crescente a sinistra dalla metà in poi, nel verso opposto. Le relazioni di ricorrenza diventano quindi:

$W(n) = 3n$ se $n = 2^k$ (3 colonne per ogni singolo nodo)

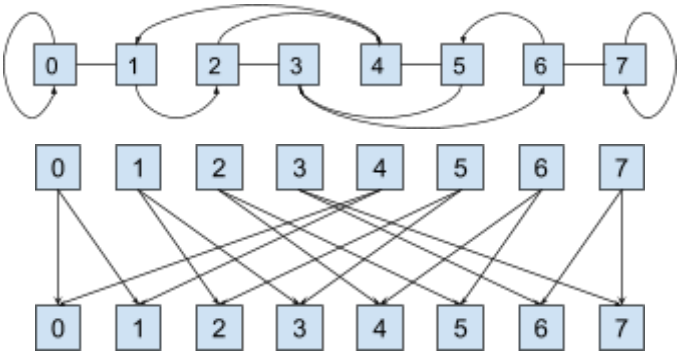
$H(n) = 4$ se $n = 2$

$H(n) = H(n/2) + n + 1$ se $n > 2$

Master Th.: $\alpha = \log 1 / \log 2 = 0$; $\beta = 1$; $\alpha < \beta$ quindi $O(n^1) \rightarrow H(n) = O(n)$; E quindi l'area $A = W(n)H(n) = O(n)O(n) = O(n^2)$.

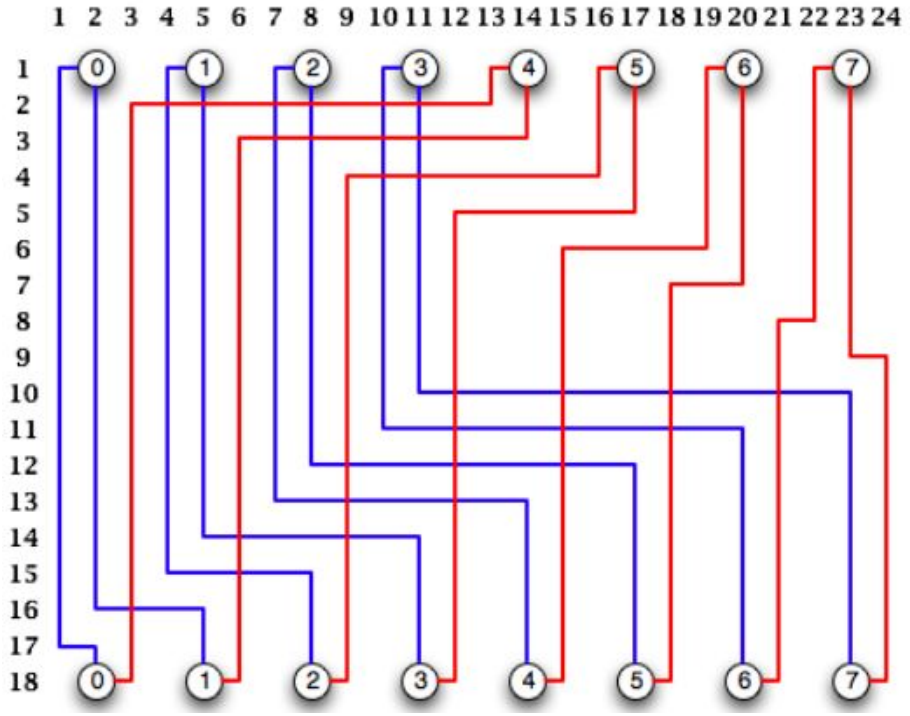
Layout: Shuffle

Si ponga sempre $n = 2^i$ per creare la shuffle è necessario ridefinirla per poterla realizzare su un circuito. Si considerino quindi 2 file di n nodi ciascuna con i nodi numerati da 0 ad n-1, si connettono i nodi di una fila con quelli dell'altra. In questa ridefinizione le connessioni di scambio non esistono e per ciascun nodo vengono calcolati i nodi raggiunti eseguendo *in successione una mossa shuffle ed una exchange*.



Per la realizzazione all'interno di un circuito è adottata la seguente convenzione per i collegamenti tra i nodi superiori ed inferiori:

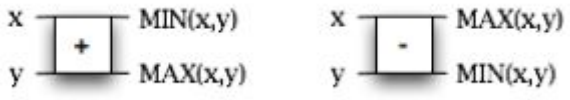
Per ricordarsi la composizione del circuito bisogna rispettare il vincolo sopra, e la prima metà dei nodi forma una L, mentre l'altra riempie gli spazi vuoti rimanenti:



Per determinare l'area del circuito è opportuno calcolare separatamente il numero di righe e colonne. Il numero di colonne è calcolabile osservando che ogni nodo occupa 3 colonne, quindi in totale sono 3n; per il numero di righe si osservi che ogni nodo occupa uno scalino rosso ed uno blu, quindi le righe intermedie sono 2n, alle quali vanno sommate le due contenenti i nodi, totale 2n+2 righe. L'area A è quindi $3n(2n + 2) = O(n^2)$.

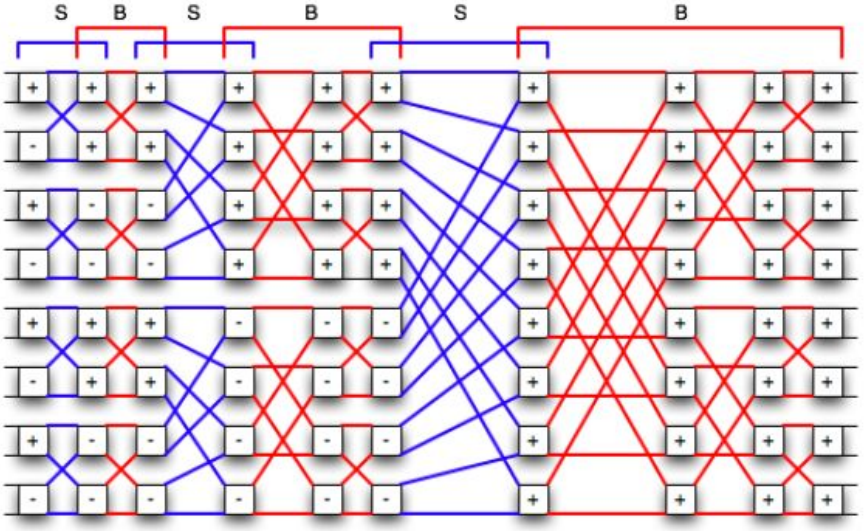
Ordinamento: Bitonic mergesort

Viene presentato l'algoritmo mergesort bitonico in un circuito VLSI. Si utilizzi la seguente convenzione per rappresentare gli elementi:



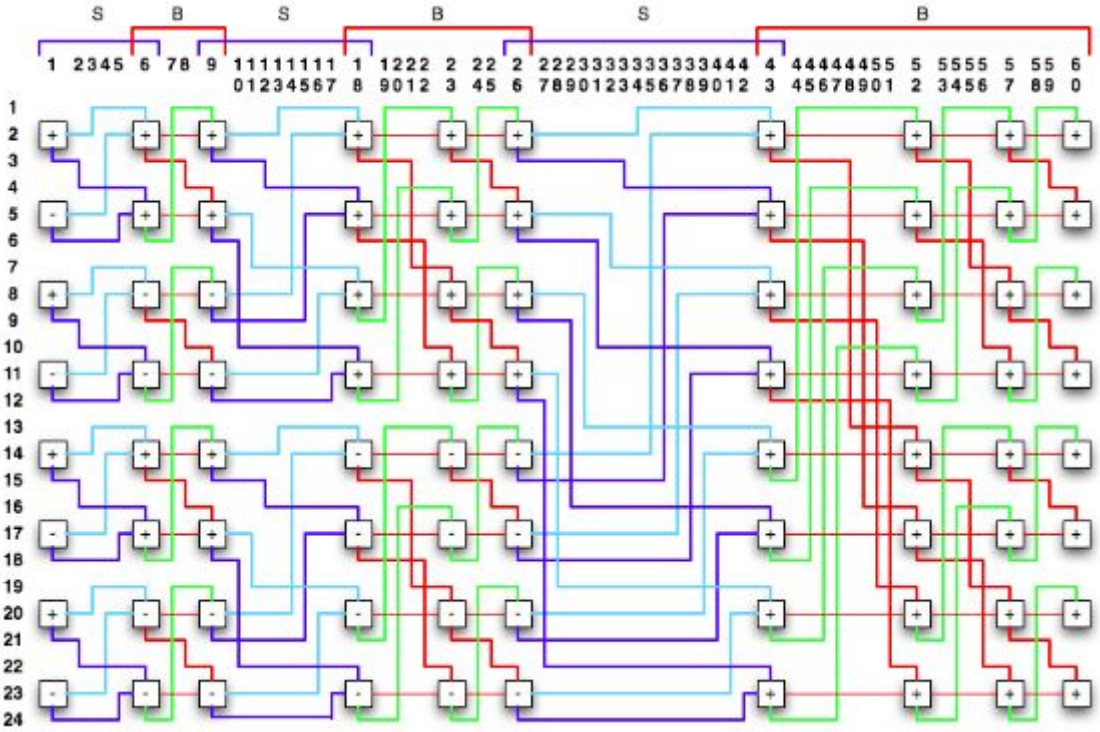
Si può disegnare il grafo di Batcher per ordinare una sequenza di lunghezza pari a $n = 2^{h+1}$, con $h \geq 1$ costruendo ricorsivamente un layout a partire dal caso base costituito da una shuffle di 2 nodi e una butterfly di rango 1, entrambe ruotate di

90° in senso antiorario. Di seguito il layout di un grafo di Batch di rango h = 3 in grado di ordinare una sequenza di n = 2^4 = 16 elementi. Le connessioni delle shuffle sono in blu, mentre quelle butterfly in rosso.



Gli elementi sono:
4 shuffle di 2 nodi, 4 butterfly di rango 1.
2 shuffle di 4 nodi, 2 butterfly di rango 2.
1 shuffle di 8 nodi, 1 butterfly di rango 3.

Il circuito è il seguente:



Per il calcolo di righe e colonne va tenuto a mente i valori visti per shuffle e butterfly in precedenza, ma invertiti in quanto ruotati di 90°. Quindi una shuffle di p nodi ha 3p righe e 2p+2 colonne, mentre una butterfly di rango k ha 3*2^k righe e $\sum_{1 \leq i \leq}$

$\sum_{k}(2^i) + k + 1$ colonne. Facendo vari calcoli viene che sia il numero di righe che quello di colonne è O(n) quindi l'area A risulta essere $O(n^2)$. Il tempo richiesto per l'ordinamento si può considerare che ciascun confrontatore richiede O(1) di area e O(1) di tempo, ed ogni filo ha ampiezza O(1), poiché trasporta un numero costante di bit. È quindi sufficiente calcolare quante colonne di confrontatori vi sono: per fare ciò si può osservare come per ogni butterfly il numero di colonne di confrontato aumenti di 1 all'aumentare del rango. Inoltre va sommata la prima colonna, non contata in nessuna butterfly. Dato il rango h del grafo di Batcher, si hanno h butterfly, ciascuna delle quali ha h+1 livelli (quindi colonne nel grafo di Batcher): tempo $T = O(\log^2 n)$, e quindi $AT^2 = O(n^2) * O(\log^2 n)^2 = O(n^2 \log^4 n)$.
NOTA: utilizzando i confronti come spiegati sotto, i quali dati due elementi di k bit hanno tempo T = O(log k) e area A = O(k log k) in quanto è un normale albero con foglie sul bordo, il risultato nel mergesort bitonico con coppie shuffle-butterfly e i confronti ad albero, ha tempo $T = O(\log^2 n \log k)$ ed area $A = O(n^2 k \log k)$ per un'energia $E = AT^2 = O(n^2 k \log^3 k \log^4 n)$.

Confronto tra due elementi
Dato un albero con le foglie sul bordo, un circuito per confrontare due elementi X e Y è eseguito nel modo seguente: la generica foglia i confronta i bit x_i ed y_i e trasmette al padre il valore binario:

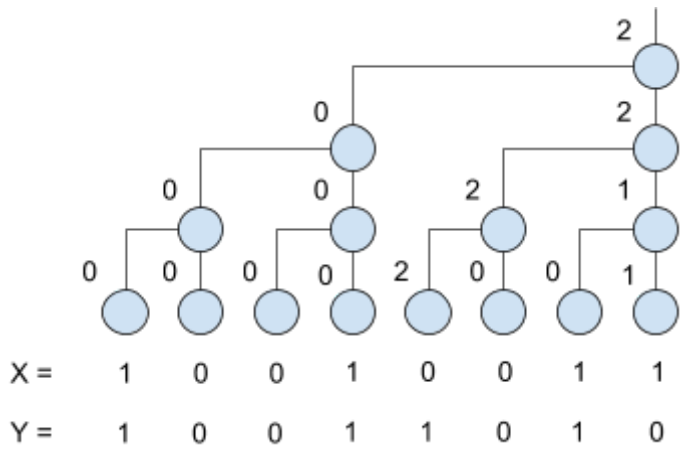
2 (1 0) se $x_i < y_i$
0 (00) se $x_i = y_i$
1 (0 1) se $x_i > y_i$

Da questo punto fino in cima ogni nodo padre avrà:
- il valore proveniente dal figlio **destro** → se il figlio sinistro riceve 0
- il valore proveniente dal figlio **sinistro** → altrimenti (figlio sinistro riceve 1 o 2)

Il risultato del confronto risiede nella radice, è:

2 se X < Y
0 se X = Y
1 se X > Y

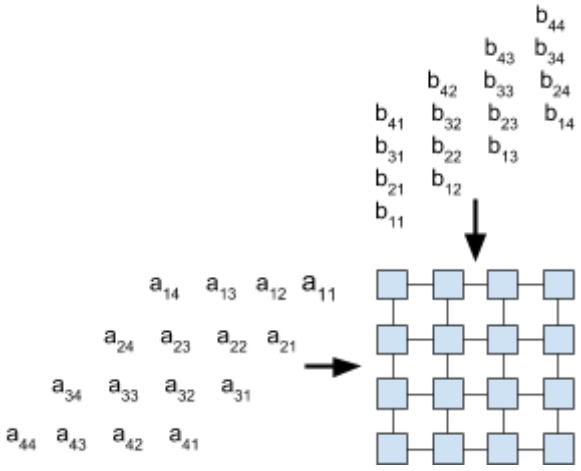
In base al risultato gli elementi X e Y sono emessi sugli opportuni fili d'uscita.



Il tempo T = O(log k) mentre l'area A = O(k log k), per un'energia $AT^2 = O(k \log^2 k)$, esattamente come un albero con foglie sul bordo.

Moltiplicazione di matrici: Moltiplicatore sistolico

Il problema che si vuole affrontare è moltiplicare due matrici $A = [a_{ij}]$ e $B = [b_{ij}]$ di dimensioni n×n. Il grafo della computazione consiste in una mesh quadrata n×n dove gli elementi vengono introdotti in modo sincrono dall'alto e da sinistra opportunamente traslati in modo tale che ogni unità di elaborazione calcoli il prodotto dei due elementi che giungono da sinistra e dall'alto, sommandolo in un accumulatore interno. In tempo $T = O(n)$ tutti i valori hanno attraversato la mesh e i risultati vengono emessi. Riassumendo: $T = O(n)$, $A = O(n^2)$, $E = AT^2 = O(n^4)$, l'algoritmo non è efficiente in quanto $T = O(n)$, ma AT^2 è ottimo.

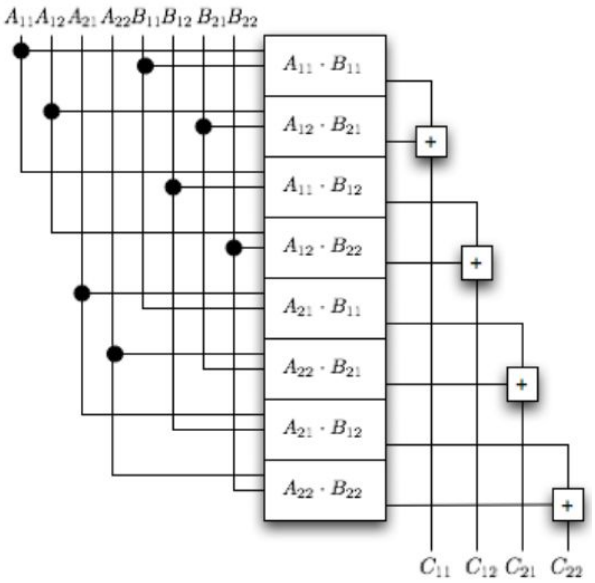


Moltiplicazione di matrici: Moltiplicatore ricorsivo

Partendo dalle stesse ipotesi del moltiplicatore sistolico, imponendo $n = 2^k$, è possibile considerare le matrici di input A e B e quella di output C come scomposte ciascuna in quattro blocchi di dimensione n/2 × n/2. In tal modo è possibile determinare la matrice C come:

$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j}$ con $1 \le i, j \le 2$

In tal modo per moltiplicare tra loro due matrici n×n sono necessarie 8 moltiplicazioni e 4 addizioni di matrici n/2 × n/2. Si ottiene dunque un layout rettangolare ricorsivo (vedi la figura seguente) per la moltiplicazione a blocchi che consta di 8 moltiplicatori per matrici di dimensioni dimezzate, ogni filo rappresenta un fascio di $(n/2)^2$ fili, uno per ciascun elemento di un blocco n/2 × n/2 (ciascuno dei quali ha lo stesso layout rettangolare di quello che li contiene, a meno di una rotazione di 90° in senso orario ed un ribaltamento orizzontale), più contiene anche 4 moduli addizionatori di matrici n/2 × n/2 che schematizzano $(n/2)^2$ addizionatori tra coppie di elementi delle matrici.



Definiti H(n) e W(n) l'altezza e la base di un moltiplicatore n × n, è possibile impostare le seguenti relazioni di ricorrenza "incrociate" sfruttando la ricorsività del layout:

$$H(n) = \begin{cases} 1 & n = 1 \\ 8W(\frac{n}{2}) & n > 1 \end{cases}$$

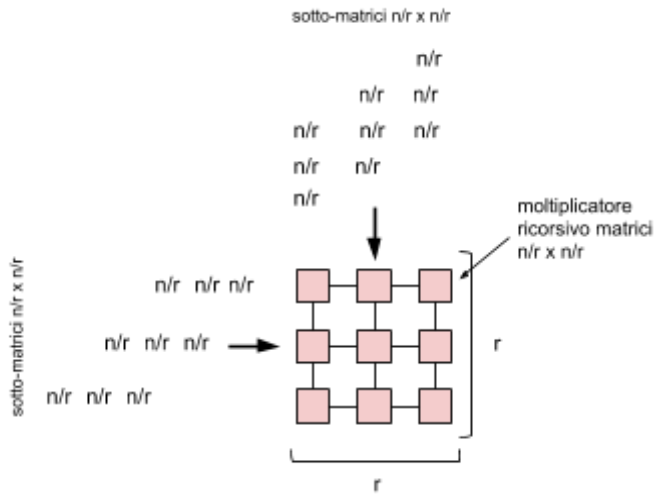
$$W(n) = \begin{cases} 1 & n = 1 \\ H(\frac{n}{2}) + 12(\frac{n}{2})^2 & n > 1 \end{cases}$$

Sostituendo e risolvendo in entrambi i casi si ottiene $O(n^2)$, quindi l'area $A = O(n^4)$. Il tempo di computazione è pari a $T = O(\log n)$ in quanto si può osservare come i livelli di elaboratori siano pari a $2[\log(n)] + 1$. Quindi $T = O(\log n)$, $A = O(n^4)$ ed $E = AT^2 = O(n^4 \log^2 n)$.

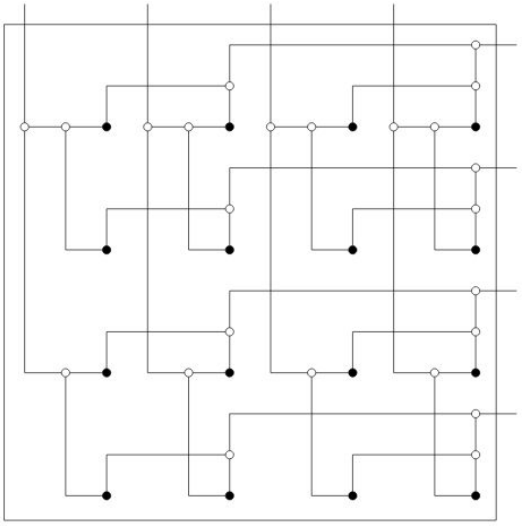
Moltiplicazione di matrici: algoritmo di Preparata-Vuillemin

Per migliorare i valori di area, tempo ed energia è possibile utilizzare una combinazione dell'algoritmo sistolico, dell'algoritmo ricorsivo e di una pipeline. Una matrice A di dimensione n × n può essere scomposta in r^2 sottomatrici n/r × n/r, pertanto si può considerare un moltiplicatore sistolico formato da una mesh r × r, dove ognuno degli r^2 nodi è un moltiplicatore ricorsivo di matrici n/r × n/r moltiplicando in pipeline r coppie di matrici n/r x n/r e accumulandone la sommatoria. Quindi $C_{i,j} = \sum_{1 \le k \le r} A_{i,k} B_{k,j}$.

La mesh ha r^2 nodi ed ogni nodo ha un area di $O(n^4 / r^4)$, quindi $A = O(r^2 * n^4 / r^4) = O(n^4 / r^2)$. Il tempo necessario ad ogni sottomatrice per attraversare la mesh è pari ad O(r) ed ogni nodo della mesh, ovvero ogni moltiplicatore ricorsivo, deve effettuare r moltiplicazioni di matrici n/r × n/r che giungono in pipeline ad istanti di tempo consecutivi. Sfruttare l'effetto pipeline richiede dunque un tempo $O(r + \log(n/r))$, che risulta pari a O(r) con $r \in [\log(n), n]$. Quindi $T = O(r)$, $A = (n^4 / r^2)$ ed $E = AT^2 = (n^4 / r^2 * r^2) = O(n^4)$ che risulta ottimo.



Esercizio: layout mesh di alberi



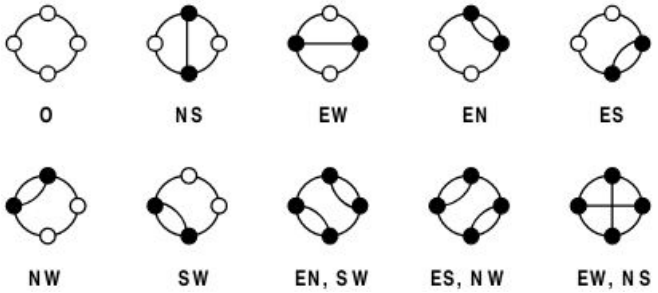
Si ha una griglia di n^2 nodi con $n = 2^k$. I nodi di ogni linea sono le foglie di un albero binario. Ha diametro O(log n) e grado 3.

L(n) = O(n log n), quindi area $A = L^2(n) = O(n^2 \log^2 n)$.

Algoritmi per mesh configurabili

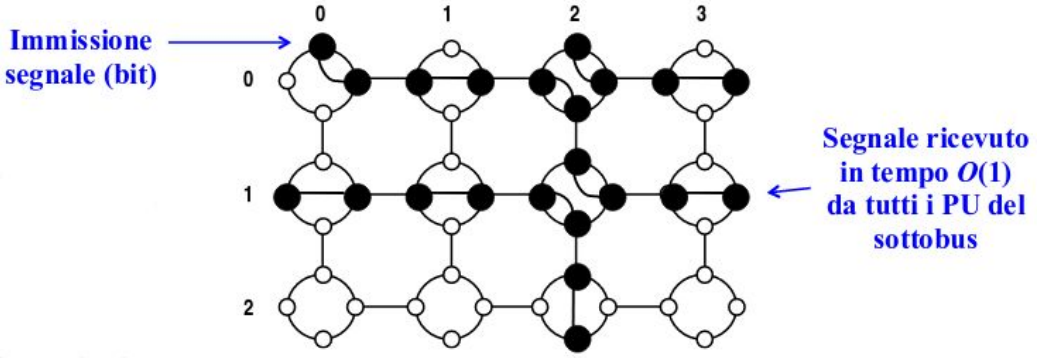
(processori sincroni - memoria locale)

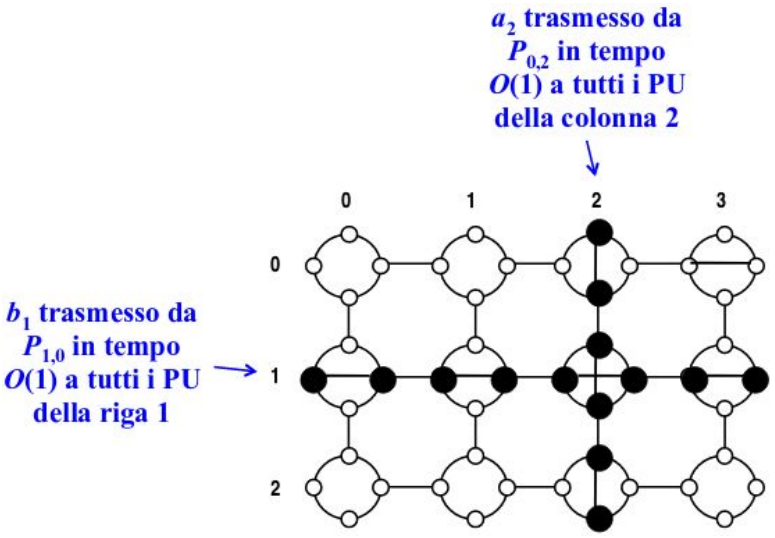
Una mesh riconfigurabile è una usuale mesh dove ogni nodo è un'unità di elaborazione che possiede una memoria locale ed uno "switch" con capacità di riconfigurazione "dinamica". Su questi vengono implementati algoritmi per calcolatori ottici, in cui gli switch operano come specchi che deviano il segnale luminoso. Lo switch consiste in 4 porte (N, S, W, E) sulle quali sono incidenti gli archi provenienti dagli altri nodi. Le possibili configurazioni sono:



È un modello sincrono, dove ogni nodo (PU - process unit) della mesh ha uno switch riconfigurabile con 4 porte. Le Interconnessioni possono essere:
- fisse tra nodi (PU) della mesh;
- dinamiche tra le porte degli switch di ogni nodo.
Un segnale che viaggia su un filo (sotto-bus) non è ritardato dai nodi intermedi e può raggiungere in tempo O(1) qualsiasi nodo del filo, indipendentemente dalla lunghezza del sottobus e può essere letto da qualsiasi unità di elaborazione presente nel sotto-bus.

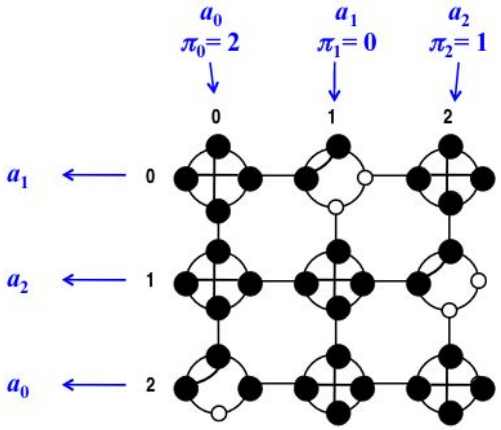
Ogni passo di una computazione parallela della mesh riconfigurabile richiede tempo O(1) ed è suddiviso nelle seguenti fasi:
1. ogni unità di elaborazione della mesh effettua una computazione locale che richiede tempo O(1);
2. ogni unità di elaborazione riconfigura autonomamente il proprio switch in tempo O(1);
3. al più un'unità di elaborazione per ciascun sotto-bus trasmette un segnale (un bit) sul proprio sotto-bus in tempo O(1);
4. una o più unità (anche tutte) di elaborazione connesse allo stesso sotto-bus ricevono il segnale dopo un tempo O(1).





Permutazione

Si prenda il caso di una permutazione di numeri dato un vettore π in cui ogni elemento π_j contiene la posizione di "arrivo" dell'elemento j-esimo di un altro vettore a (permutando quindi a), di seguito lo schema:

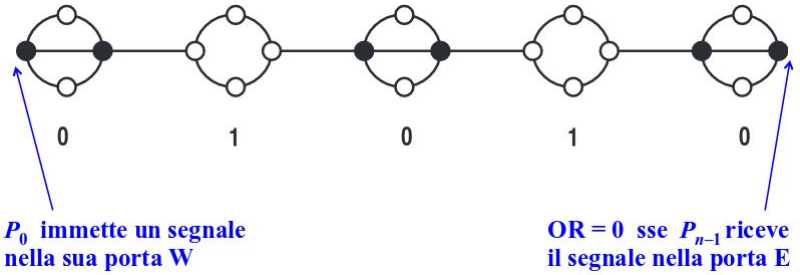


L'elemento $P_{i,j}$ viene configurato NW se $\pi_j = i$, altrimenti NS-EW. Il tempo impiegato per la manovra è $T = O(1)$, mentre il numero di processori $P = O(n^2)$.

OR logico

P_i memorizza il bit b_i e si configura O se $b_i = 1$, oppure EW se $b_i = 0$. Il nodo P_0 emette un bit a 1 sulla porta W, così è facile intuire che l'ultimo nodo (P_{n-1}) riceverà sulla propria porta E il bit 1 se e solo se tutti i nodi hanno configurazione EW, quindi hanno tutti i bit 0 e comunicano tra loro. Se la porta E del nodo P_{n-1} riceve il segnale 1 non è presente nemmeno un valore 1 (che bloccherebbe il flusso con la sua configurazione ad O) e il risultato sarà 0, se il segnale non è ricevuto, significa che vi è almeno un bit impostato ad 1 e il segnale emesso sarà quindi 1.

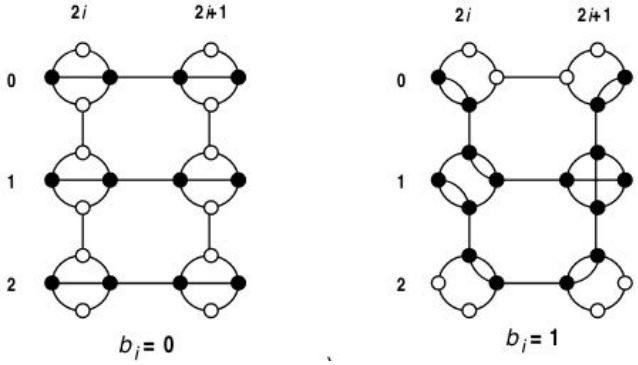
Concettualmente l'OR restituisce 1 se vi è almeno un elemento 1, altrimenti 0. Per far ciò era necessario trovare un modo di far avvenire un cambiamento al flusso *solo se* vi è un 1 presente, e l'unico modo è ragionare in modo "inverso", ovvero il flusso viene stoppato solo se vi è un 1, altrimenti scorre fino in fondo; e se il flusso viene stoppato significa che l'OR avrà risultato positivo e si considera l'esito ad 1.



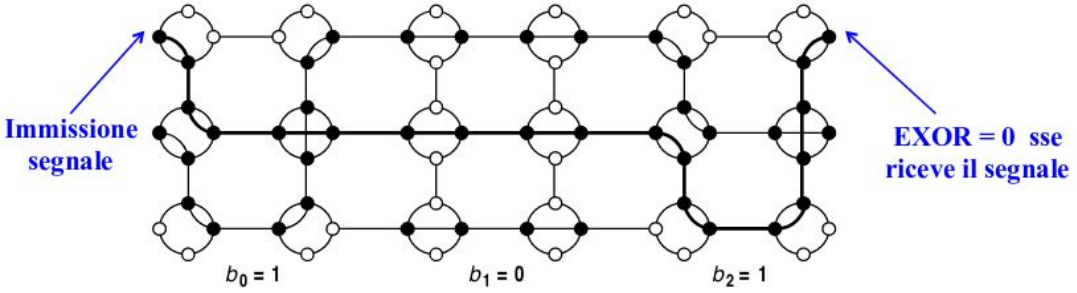
Il tempo è O(1), mentre il numero di PU è O(n).

XOR logico (somma modulo 2) di n bit

Concettualmente lo XOR ha come risultato 1 se il numero dei bit con valore 1 è dispari, altrimenti il risultato è 0. P_i memorizza il bit b_j e viene configurato come segue in base al suo valore:



Supponendo come esempio il numero 5 = 101, verrà configurato come di seguito:

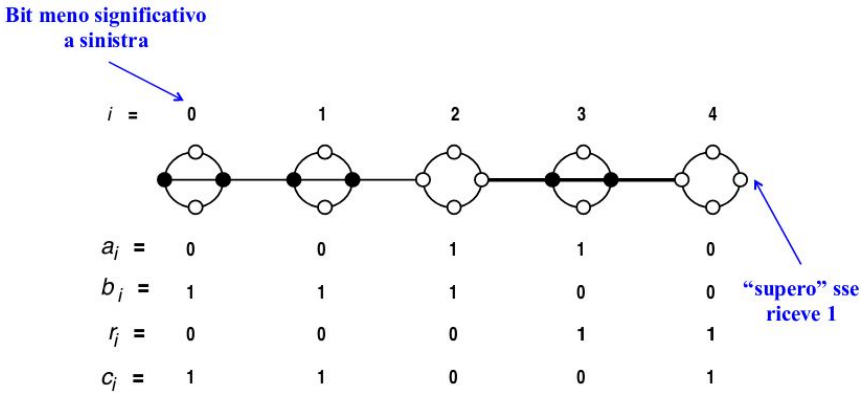


Viene inviato un segnale luminoso dalla porta W del nodo $P_{0,0}$ se questo arriva alla porta E del nodo $P_{0,2n-1}$ viene emesso in output il risultato 0 (numero pari di 1), se invece il segnale arriva al processore $P_{1,2n-1}$ il nodo $P_{0,2n-1}$ non riceve il segnale e quindi emette il risultato 1 (in quanto si avrà un numero dispari di 1). Quindi in linea di massima la lettura del risultato è la medesima dell'OR: se il segnale arriva significa che il numero di 1 è pari e lo XOR ha risultato negativo, altrimenti se il segnale non arriva lo XOR ha risultato positivo in quanto si ha un numero dispari di 1.

Addizione tra due interi di n bit

Dati due numeri a e b di n bit ciascuno, se entrambi hanno l'i-esimo bit uguale ad 1, quindi $a_i = b_i = 1$ viene *generato* un riporto (la porta E del nodo emette un segnale), se invece i due bit sono differenti, quindi $a_i \neq b_i$ il riporto (se presente) viene propagato in avanti. Ancora, se i due bit sono uguali, ma entrambi a 0, $a_i = b_i = 0$ vi

è un'interruzione del riporto. Se i due bit i-esimi sono uguali tra loro (due 1 o due 0) la configurazione del nodo è ad O, altrimenti è EW.



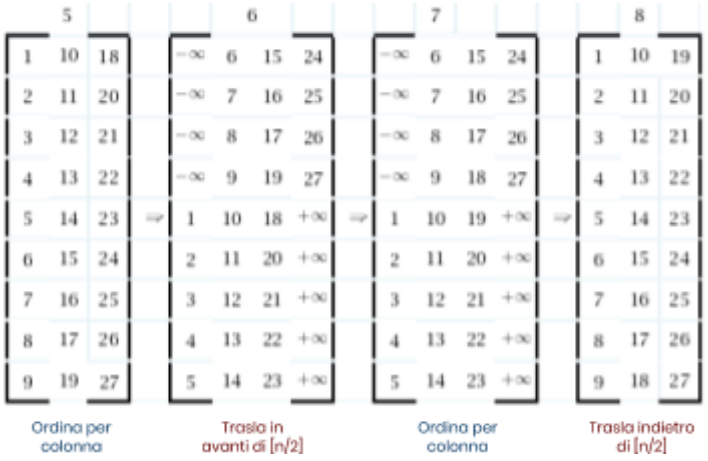
Il tempo è O(1), mentre il numero di PU è O(n).

Ordinamento: Columnsort

Dati n elementi, con una mesh riconfigurabile n × n è possibile ordinarli in tempo O(1). L'algoritmo prende in input una matrice A di dimensione r × s dove $r \geq 2(s-1)^2$ ed r mod s = 0 (il numero di righe è più grande del doppio del quadrato delle colonne ed è un suo multiplo) e la riordina per colonne. L'algoritmo consta di 8 fasi:

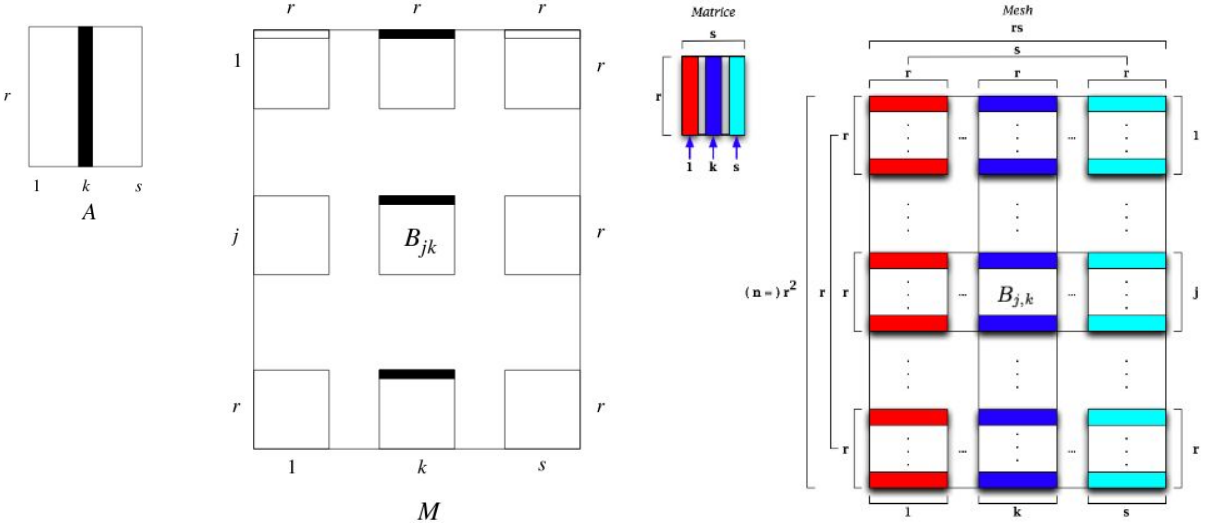
- Durante le fasi dispari (fasi 1, 3, 5, 7) gli elementi di ogni colonna vengono ordinati in parallelo.
- Per le fasi pari avvengono le seguenti azioni:
  - *Fase 2*: gli elementi della matrice di input A, presi in ordine di colonna, vengono riarrangiati per riga.
  - *Fase 4*: in maniera opposta rispetto alla fase 2, gli elementi vengono presi in ordine di riga e vengono riarrangiati per colonna.
  - *Fase 6*: gli elementi della matrice vengono scalati "in basso" di ⌊r / 2⌋ posizioni, aggiungendo una colonna in più. Viene effettuato un padding con -∞ nelle posizioni vuote della prima colonna e con +∞ nelle posizioni vuote dell'ultima colonna.
  - *Fase 8*: gli elementi della matrice vengono riscalati "in alto" di ⌊r / 2⌋ posizioni, eliminando la colonna aggiunta precedentemente.

Di seguito è eseguito l'algoritmo sulla matrice numero 0, per r = 9 ed s = 3.



Una versione meno efficiente, ma più semplice da spiegare, impone ulteriori vincoli sulle proprietà di r ed s, ovvero $r = \sqrt{n} = n^{1/2}$ ed $s = \frac{1}{2} \cdot \sqrt[4]{n} = \frac{1}{2} \cdot n^{1/4}$. Quindi il numero di elementi da ordinare risulta $r \cdot s = \frac{1}{2}n^{3/4}$. Pertanto la dimensione della mesh utilizzata è $r^2 \times rs = n \cdot \frac{1}{2}n^{3/4}$.

Gli elementi della matrice A sono caricati per colonna sulla prima riga della mesh, pertanto essa conterrà tutti gli elementi della matrice A, poiché la mesh è larga $r^2 \times rs$. Inoltre si può considerare la mesh partizionata in $r \times s$ sotto-mesh quadrate $B_{j,k}$ di dimensione $r \times r$, per $1 \le j \le r$, $1 \le k \le s$. Pertanto, la colonna k della matrice A sta inizialmente nella prima riga della sotto-mesh $B_{1,k}$.

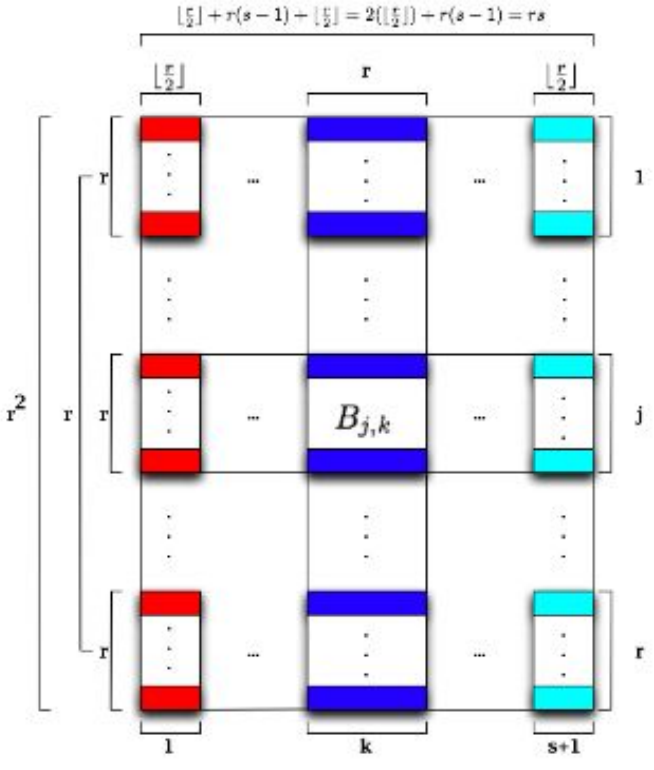


Le fasi della computazione sono le seguenti:

- Le **fasi dispari** (1, 3 e 5) ordinano le colonne di A:
  - Prima di tutto si configurano gli switch a NS, in modo da distribuire verso il basso il contenuto della prima riga della mesh M (cioè tutti gli $r \times s = \frac{1}{2}n^{3/4}$ elementi da ordinare) a tutte le $n = r^2$ righe della mesh;
  - M è composta da rs sotto-mesh quadrate $B_{j,k}$, di dimensione $r \times r$. Ciascuna di queste sotto-mesh contiene gli elementi della k-esima colonna di A, ripetuti per ogni riga, e viene utilizzata per determinare

la posizione dell'elemento j-esimo all'interno dell'ordinamento della k-esima colonna di A. Per far ciò si effettua un "torneo" per determinare la posizione dell'elemento j-esimo all'interno della colonna, calcolando gli r bit relativi ai confronti $a_{j,k} \geq a_{i,k}$, per $1 \leq i \leq r$, ed effettuandone la sommatoria.
    - Ogni sotto-mesh $B_{j,k}$ utilizza in seguito i suoi sotto-bus colonna per spostare gli elementi fino alla posizione calcolata. Possono verificarsi conflitti qualora questi elementi siano uguali e quindi debbano occupare la medesima posizione. Questi conflitti vengono risolti dando la priorità ai blocchi $B_{j,k}$ con indice di riga inferiore, riempiendo successivamente le posizioni rimaste vuote copiando il primo elemento che si trova sulla destra. In questo modo si è completata la fase di ordinamento in colonna.

- Per quanto riguarda le **fasi pari**:
    - Fasi 2 e 4: consistono nel permutare gli elementi della prima riga della mesh analogamente a quanto descritto nell'algoritmo su matrice, ovvero prima considerandoli concettualmente in ordine di colonna, e ridisponendoli in ordine di riga, e poi viceversa; i risultati delle permutazioni devono essere sistemati sempre lungo la prima riga in modo da essere distribuiti a tutta la mesh nelle fasi dispari.
    - Fase 6: diversamente da quella illustrata per la matrice, non c'è necessità di spostare gli elementi poiché è possibile riconfigurare la mesh sottostante per ottenere lo stesso risultato; infatti se all'inizio vi sono s blocchi $B_{j,k}$ di dimensioni r × r, in questa fase si ripartizionano in r × (s+1) blocchi in modo tale che la prima e l'ultima colonna abbiano blocchi di dimensione r × [r/2] e le restanti di dimensione r × r, come mostrato nella seguente figura. A questo punto viene eseguita una fase dispari, tenendo conto della nuova suddivisione, e si ottiene la sequenza ordinata sulla prima riga della mesh.



Programmazione dinamica: massima sottosequenza comune

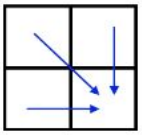
Vengono fornite in input due sequenze di simboli, P e T. Mentre in output è fornita la più lunga sottosequenza comune a P e T. Esempio: P = "AAAATTGA", T= "TAACGATA" LCS(P,T) = "AAATA". LCS(P,T) non è composta da caratteri adiacenti (non è una stringa) e può essere risolta in sequenziale con la Programmazione Dinamica.

Si crea una tabella per memorizzare la lunghezza dei vari sottoproblemi di LCS:
- D è una tabella (m + 1) × (n + 1) elementi con m lunghezza di P e n lunghezza di T.
- L'elemento D[i, j] è la lunghezza della massima sottosequenza comune di P(i) e T(j).
- S(i) è il prefisso della sequenza S, aumentata di un carattere iniziale vuoto, che termina in posizione i (con i ≥ 0).

L'obiettivo è calcolare D[m, n], ossia la lunghezza della LCS di P e T.
La formula ricorsiva è:

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ D[i-1, j-1] + 1 & \text{se } i > 0 \wedge j > 0 \wedge p_i = t_j \\ \max\{D[i-1, j], D[i, j-1]\} & \text{se } i > 0 \wedge j > 0 \wedge p_i \neq t_j \end{cases}$$

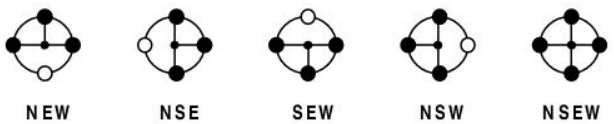
Esempio:
P = "TACCBT"
T = "ATBCBD"



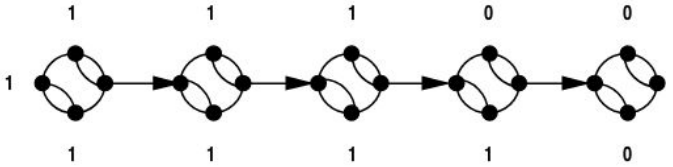
↘ deriva da *i*-1,*j*-1
↓ deriva da *i*-1,*j*
→ deriva da *i*,*j*-1

i \ j		0	1	2	3	4	5	6
			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	↓0	↘ 1	→1	→1	→1	→1
2	A	0	↘ 1	↓ 1	↓ 1	↓ 1	↓ 1	↓ 1
3	C	0	↓ 1	↓ 1	↓ 1	↘ 2	→2	→2
4	C	0	↓ 1	↓ 1	↓ 1	↘ 2	↓ 2	→2
5	B	0	↓ 1	↓ 1	↘ 2	↓ 2	↘ 3	→3
6	T	0	↓ 1	↘ 2	↓ 2	↓ 2	↓ 3	↓ 3

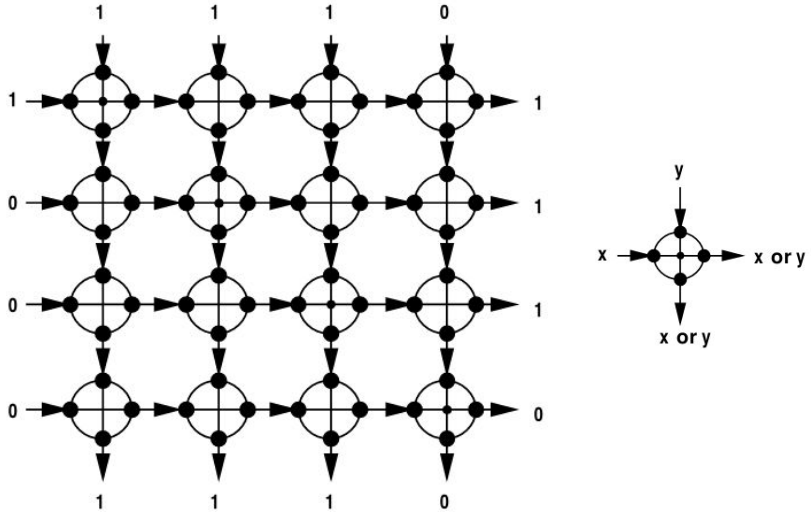
Per applicare questo modello alle mesh riconfigurabili è necessario introdurre le operazioni di incremento (+ 1) e di massimo (max). Per far ciò si introduce un modello orientato di mesh riconfigurabile in cui si hanno le seguenti ulteriori 5 configurazioni:



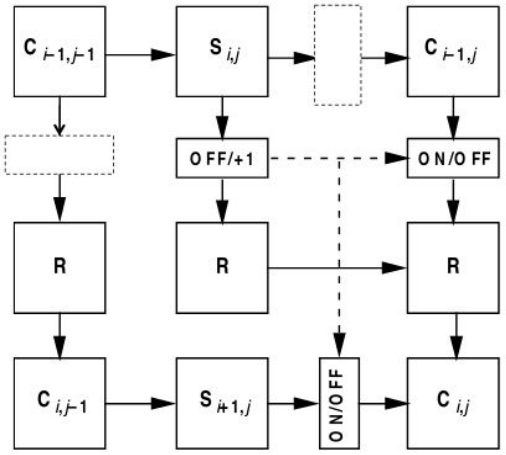
L'operazione di **incremento** (esempio di incremento di +2):



L'operazione di **massimo**:



La mesh riconfigurabile per determinare la massima sottosequenza comune è costituita da vari elementi $C_{i,j}$, $S_{i,j}$, R che rappresentano reti per il massimo di dimensioni O(h × h). $S_{i,j}$ memorizza p_i e t_j e imposta gli interruttori ON/OFF e OFF/+1 secondo l'esito del confronto tra p_i e t_j. Il tempo è T = O(1), il numero di processori è P = $O(mnh^2)$ con h = min{m, n}.



Algoritmi concorrenti

(processori asincroni - memoria globale)

In questo modello di elaborazione si ha una memoria comune a tutti i processori, i quali elaborano in modo asincrono, ognuno col proprio clock. Ogni processore esegue un processo sequenziale (uguale o diverso dai processi eseguiti dagli altri

processori). Fondamentalmente si parte da un algoritmo sequenziale che risolve un problema e si divide la computazione in processi concorrenti (p.e. con pipeline), si usano inoltre meccanismi di sincronizzazione per far comunicare i processi tra loro. Infine si allocano i processi ai processori (scheduling) bilanciandone il carico. Alcune problematiche relative alla parallelizzazione sono:
- Effetto Amdhal: per piccole dimensioni si hanno molti processori inattivi.
- Granularità: è meglio avere molto lavoro tra due sincronizzazioni successive e poche sincronizzazioni totali piuttosto che il contrario.
- Software lockout: prestazioni limitate dall'eccessiva competizione per l'uso delle risorse.

Servono costrutti particolari per l'esecuzione di processi concorrenti, ad esempio fork P, join P sono costrutti non strutturati; cobegin P1 || P2 || P3 coend sono invece strutturati.

La comunicazioni tra processi avviene tramite variabili condivise in memoria. Si hanno sezioni critiche, ovvero porzioni di codice che elaborano variabili condivise. Si ha mutua esclusione dei processi sull'accesso alle variabili condivise. Si hanno primitive di sincronizzazione per garantire accesso mutuamente esclusivo alle sezioni critiche (attese dei processi): lock(x), unlock(x), non strutturato. Si hanno semafori (es. P e V), non strutturato. Si hanno monitor, strutturato.

Alcune problematiche relative all'uso di primitive di sincronizzazione sono ad esempio lo stallo (o deadlock): tutti i processi sono bloccati e in attesa tra di loro; e starvation: quando un processo non è bloccato ma non riesce ad accedere mai alla risorsa.

Per quanto riguarda le primitive di sincronizzazione, abbiamo:
- **lock**(x) - il processo acquisisce la variabile x se è libera, altrimenti è messo in una coda di attesa per x.
- **unlock**(x) - il processo libera x che è assegnata ad un processo della coda di attesa di x (la disciplina della coda è FIFO: primo arrivato, primo servito).

Per quanto riguarda il deadlock la prevenzione può essere fatta se ogni processo rispetta l'ordine lessicografico sulle lock (es. x prima di y in ordine alfabetico) oppure acquisisce contemporaneamente tutte le variabili. La rilevazione di uno stallo può avvenire mantenendo un grafo orientato delle attese processi-variabili, periodicamente verificando l'esistenza di un ciclo e se presente eliminando un processo del ciclo.

L'esecuzione concorrente avviene tramite la dicitura:
for all j **where** 1 ≤ j ≤ p **do concurrently** PROCESSO(j)
Il PROCESSO(j) è un processo sequenziale (può usare lock e unlock) e l'istruzione che segue il for all non è eseguita finché tutte le p procedure non sono terminate. Il tempo è dato da quello della procedura che termina per ultima.

Le prestazioni sono misurate tramite lo **speedup** = (tempo algoritmo sequenziale) / (tempo algoritmo concorrente). Lo speedup massimo è p, che è raramente raggiunto a causa dei tempi persi per le sincronizzazioni su variabili condivise.

Sommatoria
Di seguito l'algoritmo per il calcolo della sommatoria di un array a di n valori.

```
procedure SOMMATORIA (a1, a2, . . . , an)
  begin
    somma-globale := 0
    for all i where 1 ≤ i ≤ p do concurrently SOMMA(i)
  end

process SOMMA(i)
  begin
    somma-locale := 0
    for j := i to n by p do
      somma-locale := somma-locale + aj          ← somma n/p elementi
    lock(somma-globale)
      somma-globale := somma-globale + somma-locale   ← aggiorna il globale
    unlock(somma-globale)
  end
```

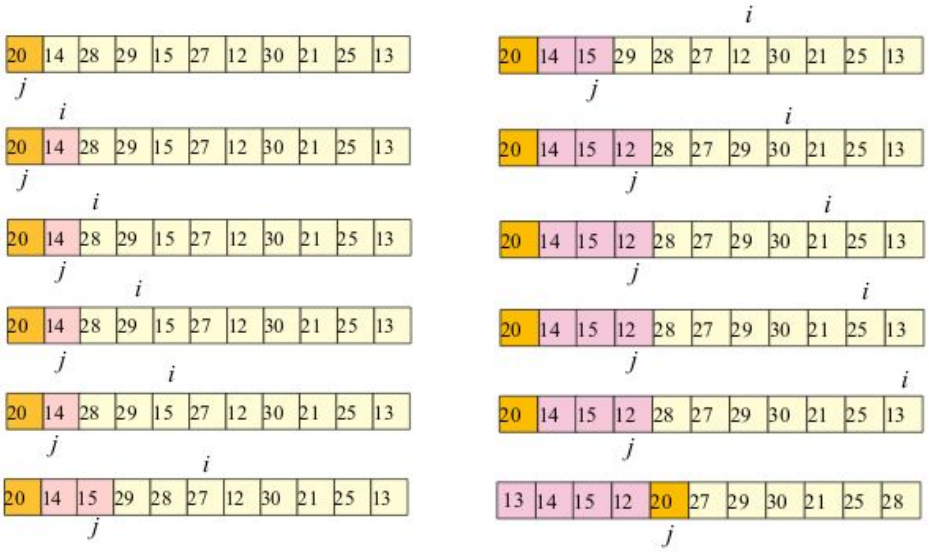
La sezione in grigio viene eseguita su ogni processore sequenzialmente, richiedendo quindi tempo O(p). Il tempo totale è T = O(n / p + p) dato dal primo for eseguito parallelamente tra tutti i p processori, più la porzione di codice eseguita sequenzialmente. Quindi lo speedup essendo il tempo dell'algoritmo sequenziale O(n) diviso il tempo dell'algoritmo concorrente O(n/p) risulta essere: O(p). Tuttavia se avessimo p = √n, la complessità sarebbe minimizzata ottenendo un O(n) / O(n/√n) = O(√n), quindi speedup = O(√n).

Ordinamento: Quicksort
Per ordinare gli elementi $a_1, a_2, \ldots, a_n$, li si riarrangia attorno ad un elemento x, detto perno, scelto a caso, in modo che gli elementi minori di x precedano x e quelli maggiori lo seguano, poi si ripete ricorsivamente alle due metà prima e dopo di x. Solitamente il perno viene scelto casualmente, e scambiato con il primo elemento della sequenza. Gli altri elementi della sequenza vengono suddivisi in tre gruppi, quelli < x, quelli ≥ x e quelli non ancora esaminati. Vengono utilizzati due indici, **i** dà la *posizione del primo elemento y non ancora esaminato* e **j** indica la *posizione finale in cui va spostato il perno*.
Ad ogni iterazione:
- se y ≥ x allora y rimane fermo,
- se y < x allora y viene scambiato con il primo elemento tra quelli ≥ x.

Alla fine x è scambiato con a_j posizionando così il perno in una posizione in cui a sinistra ha tutti elementi minori, e a destra tutti maggiori. Questa procedura è

effettuata sulla porzione di sequenza compresa tra a_h e a_k. La posizione del perno è restituita dal parametro j in uscita. Di seguito lo pseudocodice sequenziale:

```
procedure PERNO(a, h, k, j)
  begin
    m := RANDOM(h, k)
    a_m ↔ a_h
    x := a_h
    j := h
    for i := h + 1 to k do
      if a_i < x then begin
        j := j + 1
        a_i ↔ a_j
      end
    a_h := a_j
    a_j := x
  end
```



La complessità computazionale del quicksort sequenziale (ricorsivo) è data da S(n), numero medio di confronti per la sequenza $a_1, a_2, \ldots, a_n$, ipotizzando $n = 2^r - 1$ e $p = 2^q$, con $q < i$, la funzione PERNO in media divide a metà ed esegue n - 1 confronti.

$$S(n) = 2 \qquad n = 3$$
$$S(n) = 2S((n-1)/2) + n - 1 \qquad n = 7, 15, 31, \ldots$$

La soluzione è $\alpha = \log a / \log b = \log 2 / \log 2 = 1$; $\beta = 1$, quindi $S(n) = O(n \log n)$, più precisamente, per sostituzione, $S(n) = (n+1)\log(n+1) - 2n$. S(n) rappresenta il numero totale medio di confronti che l'algoritmo concorrente deve effettuare.

Per quanto riguarda il *quicksort concorrente* viene mantenuta una pila comune dove sono memorizzati gli estremi di ciascuna porzione da ordinare, e per

accedervi ogni processo usa le primitive lock e unlock. Ogni processore esegue ciclicamente lo stesso processo:
- *estrae* gli estremi della porzione da ordinare dalla testa della pila.
- *partiziona* sequenzialmente la porzione con la procedura PERNO.
- *reinserisce* gli estremi delle due porzioni nella pila.
È costituito da due fasi: nella prima fase ci sono più processi che porzioni da ordinare (molti processi sono inattivi), nella seconda fase ci sono più porzioni da ordinare che processi, quindi tutti i p processori sono attivi e con ugual carico di lavoro.

ITERAZIONE	PORZIONI	CONFRONTI PER PROCESSO	CONFRONTI TOTALI
0	1	$c_0 = n - 1 = (n + 1)/1 - 2$	$2^0 c_0 = n - 1$
1	2	$c_1 = c_0/2 - 1 = (n + 1)/2 - 2$	$2^1 c_1 = n - 3$
...	...	...	...
i	2^i	$c_i = c_{i-1}/2 - 1 = (n + 1)/2^i - 2$	$2^i c_i = n - 2^{i+1} + 1$
...	...	...	...
m	$p = 2^m$	$c_m = c_{m-1}/2 - 1 = (n + 1)/2^m - 2$	$2^m c_m = n - 2p + 1$

Nella prima fase vi sono meno processori attivi del numero di porzioni da analizzare (molti inattivi), il tempo nella prima fase $T_1(n)$ è la sommatoria di tutti i confronti per processo che sono da fare per tutte le m iterazioni: $T_1(n) = \Sigma_{0 \le i \le m} c_i$ mentre il numero dei confronti effettuati complessivamente nella prima fase è dato dalla somma dei confronti totali di tutte le m iterazioni: $C_1(n) = \Sigma_{0 \le i \le m} c_i 2^i$. Infine il tempo richiesto nella seconda fase è pari al numero di confronti che ancora rimangono da effettuare diviso il numero di processi: $T_2(n) = (S(n) - C_1(n)) / p$. Per un totale di tempo $T(n) = T_1(n) + T_2(n)$ ed uno speedup = $S(n) / T(n)$.

Cammini minimi: algoritmo di Pape-D'Esopo
Il problema prevede la risoluzione del problema dei cammini minimi da un singolo nodo r a tutti gli altri nodi in un grafo orientato e pesato di n nodi, con p processori. Si prende in input un grafo orientato pesato G = (V, A) e un nodo di partenza r. Nel grafo possono essere presenti pesi degli archi negativi, ma non cicli negativi (ovvero cicli il cui costo complessivo è negativo). Si vogliono trovare i cammini di costo *minimo* dal nodo r a tutti i nodi u appartenenti a V, l'insieme dei vertici del grafo G.

Una soluzione ammissibile è un albero T di copertura, radicato in r, che include un cammino da r ad ogni altro nodo. Ogni nodo u è caratterizzato da un valore d_u pari alla distanza del nodo da r in T, ovvero il costo del cammino tra r ed u in T. Per il teorema di Bellman, dato $c_{u,v}$ la lunghezza di un generico arco $(u,v) \in A$, una soluzione ammissibile T è ottima se e solo se:

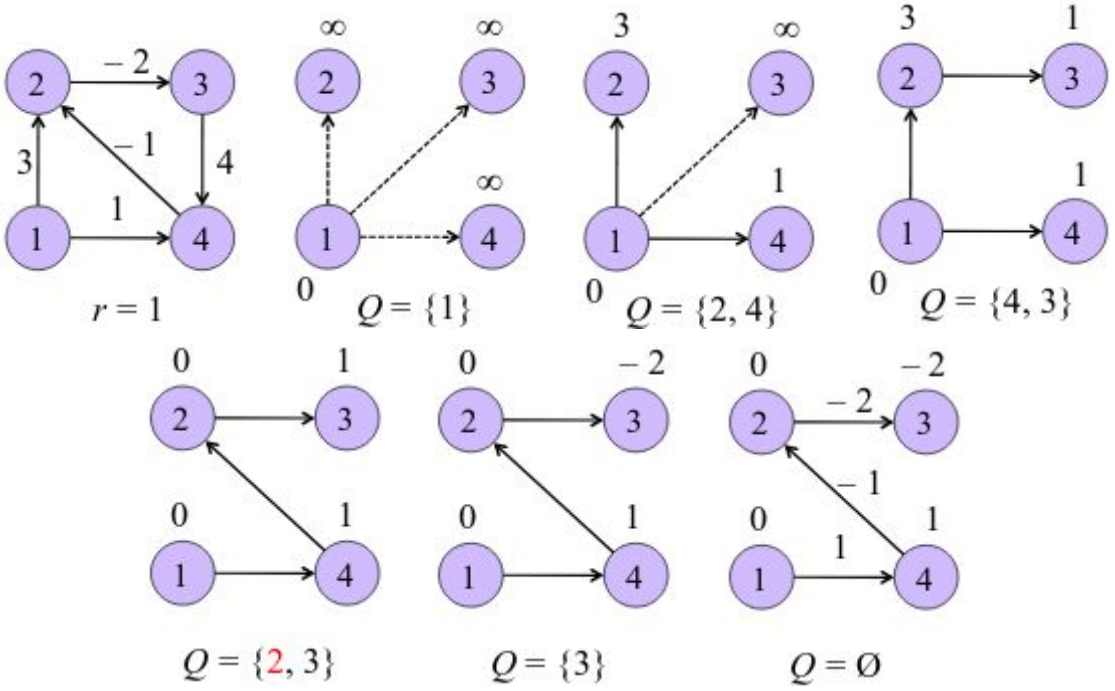
$d_u + c_{u,v} = d_v$ per ogni arco $(u, v) \in T$
$d_u + c_{u,v} \ge d_v$ per ogni arco $(u, v) \in A$ (compresi gli archi $\notin T$)

Per quanto riguarda la struttura dei dati:

- Si visita G utilizzando una *dequeue* Q (si possono inserire e togliere elementi sia in testa che in fondo alla coda), che inizialmente contiene solo r.
- Si estrae un nodo u dalla testa di Q e si controllano le condizioni di Bellman su tutti gli (u, v) che escono da u.
- Se $d_u + c_{u,v} < d_v$ la distanza d_v viene diminuita e il nodo v è inserito in Q.
- Se v è inserito in Q per la prima volta, allora viene messo in fondo, altrimenti in testa.

Per capire se un nodo v viene inserito in Q per la prima volta o meno, si inizializza $d_v = +\infty$ per ogni nodo v che è diverso dal nodo radice r, mentre $d_r = 0$. Se d_v passa da $+\infty$ ad un valore finito, allora è la prima volta.

La complessità p è esponenziale nel caso pessimo, nella pratica però per grafi sparsi "quasi" planari la complessità è lineare al numero di archi. È comunque l'algoritmo più efficiente nella pratica, per quanto riguarda ad esempio reti di circolazione stradale.




```
procedure CAMMINI-MINIMI(G, r);
  begin
    for all i where 1 ≤ i ≤ p do concurrently INIZIALIZZA(i)
    Q := { r }
    stop := false
    for all i where 1 ≤ i ≤ p do concurrently RICERCA(i)
  end;
```

```
process INIZIALIZZA(i);
  begin
    for j := i to n by p do          ← concorrentemente ogni gruppo di p elementi fino ad n
      if j = r then begin
        d_j := 0;
        p_j := 0 end
      else begin
        d_j := +∞;
        p_j := r end
  end
```

T memorizzato con un vettore dei padri

T iniziale ha r padre di tutti gli altri nodi

```
process RICERCA(i)
  begin
    attesa(i) := false
    while not stop do begin
      lock(Q)
      if Q = ∅ then begin
          attesa(i) := true
          if i = 1 then
            stop := attesa(1) and attesa(2) and . . . and attesa(p)
          unlock(Q)
        end
      else begin
        estrai u dalla testa di Q
        attesa(i) := false
        unlock(Q)
        for each (u, v) ∈ A do begin
          distanza := d_u + c_u,v
          lock(d_v)
          if distanza < d_v then begin
            if d_v = +∞ then
              prima := true
            else
              prima := false
            d_v := distanza
            p_v := u
            unlock(d_v);
            lock(Q);
            if v ∉ Q then begin
              if prima then
                inserisci v in fondo a Q
              else
                inserisci v in testa a Q;
              unlock(Q)
            end else
              unlock(Q);
```

Condizione di terminazione: un processo trova Q vuota e tutti i processi sono in attesa

La verifica è effettuata solo dal processo 1 tramite la variabile condivisa *stop*

```
      end
      unlock(d_v);
    end
  end
 end
end
```

Tecnica Branch and Bound: Commesso Viaggiatore

È una tecnica enumerativa per risolvere problemi di ottimizzazione che elenca le soluzioni del problema, ma cerca di evitare di elencarle tutte. L'enumerazione viene fatta generando ed esplorando un albero, detto *albero* delle scelte, si evita di esplorare tutto l'albero tramite *potature*. Un nodo interno dell'albero indica una soluzione parziale del problema, ovvero una sequenza di scelte, gli archi da un nodo ai figli indicano invece le scelte alternative.

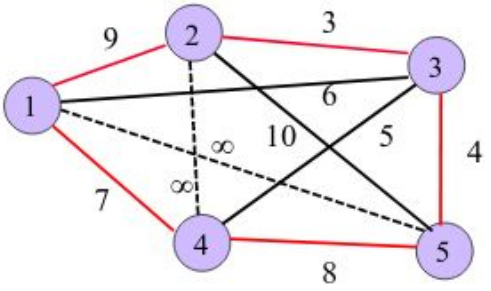
Si ha come ipotesi che il problema da risolvere sia un problema di minimo, con costo di ogni scelta non negativo, e che quindi ogni scelta, aggiunta alle scelte già effettuate non faccia diminuire il costo della soluzione parziale già ottenuta.

La funzione Lower Bound LB(S) dipende dalla sequenza S di scelte fatte, ed ogni soluzione ammissibile ottenuta aggiungendo ulteriori scelte ad S ha costo maggiore o uguale ad LB(S). Il tempo di calcolo è esponenziale nel caso pessimo, ma nella pratica è utile per problemi NP-ardui. Il tempo è influenzato da vari fattori, come:
- La regola di *branch*: ovvero il criterio per effettuare le scelte e generare i figli di un nodo dell'albero delle scelte.
- La funzione di *bound*: LB deve dare una limitazione più "stretta" possibile (più vicina al costo minimo) ed essere calcolabile velocemente.
- L'ordine di visita dell'albero delle scelte può essere: anticipato (LIFO, risparmia memoria), per livelli (FIFO) o per priorità (es. un nodo con lower bound è più promettente).

Esempio del commesso viaggiatore

Date n città e le distanze $d_{i\,j}$ tra tutte le possibili coppie i e j di città distinte, determinare un percorso ciclico che includa ogni città una e una sola volta percorrendo complessivamente la minor distanza possibile.
In input si prende un grafo non orientato completo di n nodi con i costi sugli archi, e in output si restituisce un circuito Hamiltoniano (ovvero una permutazione nodi) di costo minimo. Il sistema di memorizzazione è una matrice delle distanze.

	1	2	3	4	5
1	∞	9	6	7	∞
2	9	∞	3	∞	10
3	6	3	∞	5	4
4	7	∞	5	∞	8
5	∞	10	4	8	∞



Soluzione ottima: 1, 2, 3, 5, 4, 1

La scelta da effettuare per ogni nodo è l'inclusione o l'esclusione di un arco (i, j).
La regola di branch consiste nel considerare l'arco la cui esclusione massimizza il lower bound. La visita avviene per priorità, considerando prima il nodo dell'albero delle scelte con lower bound minimo. Il lower bound utilizza il seguente metodo di riduzione sulla matrice delle distanze (con LB inizializzato a 0):
Riduzione
- calcolare $r_i = \min_j\{d_{ij}\}$ per ogni riga i della matrice
- se $r_i > 0$, sottrarre r_i a tutti gli elementi d_{ij} della riga i e sommare r_i ad LB
- calcolare $c_j = \min_i\{d_{ij}\}$ per ogni colonna j della matrice
- se $c_j > 0$, sottrarre c_j a tutti gli elementi d_{ij} della colonna j e sommare c_j ad LB.

	1	2	3	4	5	
1	∞	9	6	7	∞	6
2	9	∞	3	∞	10	3
3	6	3	∞	5	4	3
4	7	∞	5	∞	8	5
5	∞	10	4	8	∞	4
						21

	1	2	3	4	5
1	∞	3	0	1	∞
2	6	∞	0	∞	7
3	3	0	∞	2	1
4	2	∞	0	∞	3
5	∞	6	0	4	∞
	2	0	0	1	1	4

	1	2	3	4	5
1	∞	3	0	0	∞
2	4	∞	0	∞	6
3	1	0	∞	1	0
4	0	∞	0	∞	2
5	∞	6	0	3	∞

Matrice ridotta

LB = 21 + 4 = 25

Esclusione di un arco
- si pone $d_{ij} = \infty$
- si riapplica la riduzione

	1	2	3	4	5	
1	∞	3	0	0	∞	0
2	4	∞	∞	∞	6	4
3	1	0	∞	1	0	0
4	0	∞	0	∞	2	0
5	∞	6	0	3	∞	0
						4

	1	2	3	4	5
1	∞	3	0	0	∞
2	0	∞	∞	∞	2
3	1	0	∞	1	0
4	0	∞	0	∞	2
5	∞	6	0	3	∞
	0	0	0	0	0	0

Escludi (2, 3)

LB = 25 + 4 + 0 = 29

	1	2	3	4	5
1	∞	3	0	0	∞
2	0	∞	∞	∞	2
3	1	0	∞	1	0
4	0	∞	0	∞	2
5	∞	6	0	3	∞

Matrice ridotta

Inclusione di un arco
- si somma d_{ij} ad LB
- si "cancellano" la riga i e la colonna j
- si pone $d_{ji} = \infty$
- si riapplica la riduzione

	1	2	3	4	5	
1	∞	3		0	∞	0
2						0
3	1	∞		1	0	0
4	0	∞		∞	2	0
5	∞	6		3	∞	3
						3

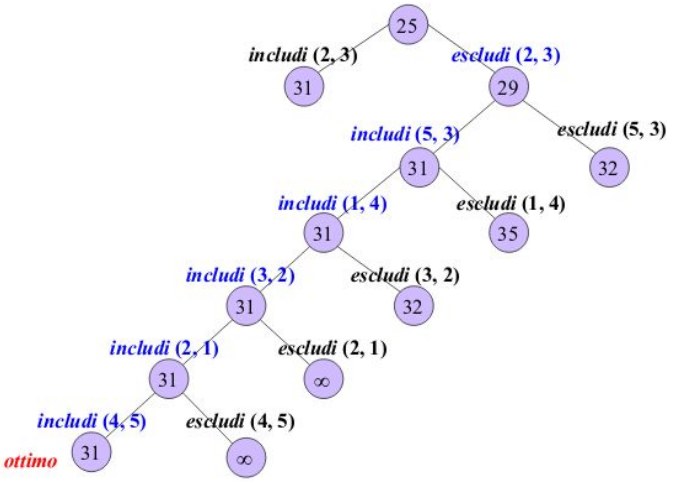
Includi (2, 3)

	1	2	3	4	5
1	∞	3		0	∞
2					
3	1	∞		1	0
4	0	∞		∞	2
5	∞	3		0	∞
	0	3	0	0	0	3

LB = 25 + 0 + 3 + 3 = 31

	1	2	3	4	5
1	∞	3		0	∞
2					
3	1	∞		1	0
4	0	∞		∞	2
5	∞	3		0	∞

Matrice ridotta



Per quanto riguarda la versione concorrente
Ciascuno dei p << n processi esegue ciclicamente le seguenti azioni:
1 - seleziona, da una lista condivisa L, il nodo dell'albero delle scelte non ancora esaminato che ha lower bound più piccolo.
2 - considera l'arco la cui esclusione massimizza il lower bound e genera i due figli del nodo corrispondenti ai due casi in cui l'arco selezionato è incluso/escluso dalla soluzione parziale finora costruita.
3 - inserisce i due nodi generati nella lista L dei nodi non ancora esaminati.
L deve essere bloccata nei passi 1 e 3. Il tempo per la mutua esclusione su L è trascurabile rispetto al tempo di computazione del passo 2.

Somme prefisse

Calcolo in modo concorrente delle somme prefisse di una certa sequenza di valori $a_1, a_2, \ldots, a_n$. Il tempo è $O(n/p + p)$.

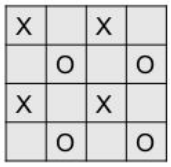
```
procedure SOMMEPREFISSE (a_1, a_2, ... ,a_n)
  begin
    for all i where 1 ≤ i ≤ p do concurrently SOMMA(i)
    for i := 2 to p do s_i := s_i + s_{i-1}          ← ogni blocco è sommato al prec.
    for all i where 1 ≤ i ≤ p do concurrently AGGIUSTA(i)
  end

process SOMMA(i)
  begin s_i := 0
    for j := (i –1)n/p + 1 to i*n/p do
      s_i := s_i + a_j                              ← ogni blocco di n/p elementi viene sommato
  end

process AGGIUSTA(i)
  begin
    b_{i*n/p} := s_i                                ← alla somma del blocco vengono sottratti i singoli valori
    for j := i*n/p – 1 down to (i –1)n/p + 1 do b_j := b_{j+1} – a_{j+1}
  end
```

Moltiplicazione di matrici

Siano date due matrici n x n e p processori concorrenti con n e p potenze di 2, e p << n. Ogni processo viene chiamato p volte, e ad ogni chiamata le righe di A sono sempre le stesse n/p ma variano le colonne di B. Il tempo per ogni processo è $O(n^3 / p^2)$ quindi in totale $T = O(n^3 / p)$.

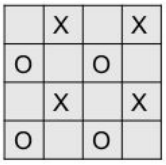
```
procedure PRODOTTOMATRICI(A, B, n)
  begin
    for l := 0 to p – 1 do
      for all i where 1 ≤ i ≤ p do concurrently MOLTIPLICA(i, l)
  end

process MOLTIPLICA(i, l)
  begin
    for h := i to n by p do
      for m := 0 to n/p – 1 do begin
        j := i + l + mp;
        if j > n then j := j – n;
        c_{h,j} := 0
        for k := 1 to n do
          c_{h,j} := c_{h,j} + a_{h,k} b_{k,j}
      end
  end
```

Esempio di moltiplicazione tra due matrici 4 x 4 con 2 processori (n / p = 2):



Processo i = 1, Chiamata l = 0 → X

Processo i = 2, Chiamata l = 0 → O



Processo i = 1, Chiamata l = 1 → X

Processo i = 2, Chiamata l = 1 → O

Riduzione conflitti in lettura sulle righe di A e sulle colonne di B.

Algoritmi distribuiti
(processori asincroni - memoria locale)

In questo modello ogni processore ha un certo ID ognuno diverso dagli altri. I processori sono connessi tra loro secondo una topologia (una rete a grafo), in cui i nodi sono i processori e gli archi le connessioni dirette tra coppie di processori. Ogni processore esegue un processo sequenziale, uguale o diverso dai processi eseguiti dagli altri processori. Non esiste un processo principale, ma è possibile eleggerlo tramite l'elezione di un leader.

Ogni processore ha solo conoscenza parziale della rete: conosce solo il proprio ID e quelli dei suoi vicini, non conosce il numero totale di processori. I dati sono partizionati tra i processori ed ognuno ha solo conoscenza parziale del problema da risolvere. Una computazione può essere iniziata da un solo processo oppure da più processi contemporaneamente (processi iniziatori), nel caso in un vi sia un singolo iniziatore la computazione è centralizzata, l'algoritmo è più semplice ma meno generale; se vi sono invece più iniziatori, la computazione è pienamente distribuita, l'algoritmo è più generale ma è più difficile dimostrarne la correttezza.

I processori comunicano tramite scambio di messaggi, usando primitive di comunicazione (come *send* e *receive*). Un messaggio "viaggia" su un arco (p, q) in tempo O(1) e si assume non siano perduti lungo il tragitto arrivando integri. Un messaggio m spedito da p a q prima di un altro messaggio m' viene ricevuto da q prima di quest'ultimo. Ogni processore ha una coda (FIFO) di messaggi di input e la complessità è rappresentata dal numero totale di messaggi scambiati.

Il controllo di ciascun processo è guidato da eventi tramite il costrutto:
when evento1 **do** azione1;
when evento2 **do** azione2;
when evento3 **do** azione3;

Alcuni esempi di eventi sono l'estrazione di un messaggio dalla coda di input, oppure uno "stimolo" esterno (es. interrupt). Non esiste un ordinamento totale tra gli eventi, se ne vogliamo uno occorre usare marche temporali (timestamps).

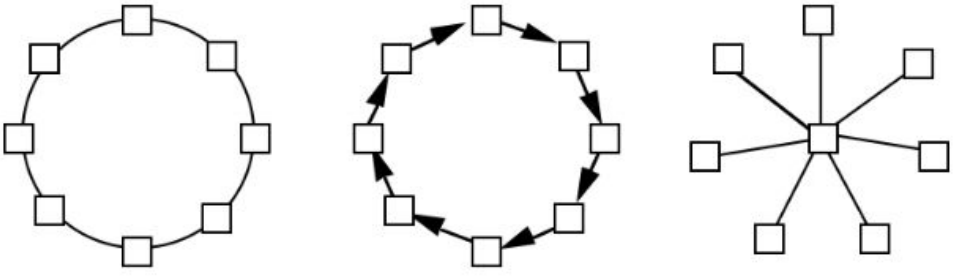
Le tecniche di progettazione sono principalmente 3:
1. *Diffusione*: un particolare processo inizialmente può solo spedire messaggi, mentre gli altri processi possono spedire un messaggio solo dopo averne ricevuto uno.
2. *Token*: un "gettone" (o "token") rappresenta un privilegio che circola nella rete, il processo che riceve il token effettua l'elaborazione e passa il token ad un altro processo.
3. *Timestamping*: viene mantenuto un clock globale logico per mezzo di timestamps che permettono di ordinare totalmente gli eventi (ad esempio per evitare "starvation").

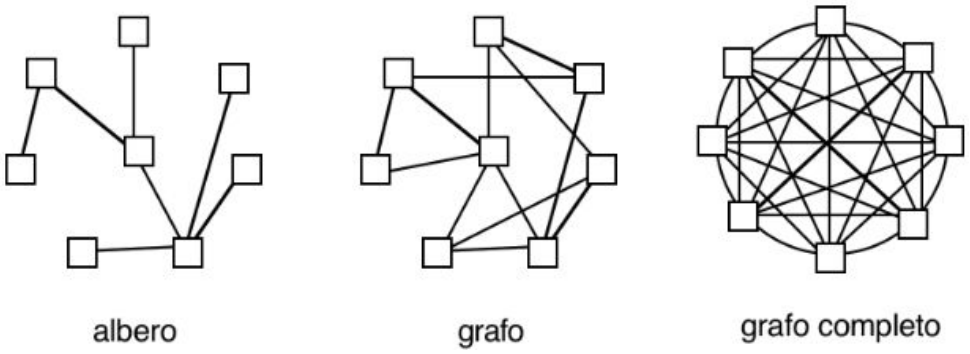
Ogni processo ha un proprio orologio (contatore), quando un processo invia un messaggio lo incrementa, quando un processo riceve un messaggio, lo aggiorna con uno più il massimo tra il proprio orologio e quello del mittente. Un esempio di codice per il processo i è il seguente:

if devi inviare m **then**
  begin
    $h_i := h_i + 1$
    send(m, h_i, i)
  end

if received(m, h_j, j) **then**
  $h_i := \max\{h_i, h_j\} + 1$

Tipologie di reti:





Il problema della **mutua esclusione** risulta banale nell'anello unidirezionale:
wait(token) **from** vicino di sinistra
. . . "sezione critica" . . .
send(token) **to** vicino di destra

Per quanto riguarda la struttura a grafo completo si può banalmente usare 3(n– 1) messaggi: n– 1 messaggi per richiedere il permesso agli altri processi, n– 1 messaggi per ottenere il permesso da tutti gli altri processi, e altri n– 1 messaggi per comunicare l'ingresso nella sezione critica. Oppure si può utilizzare l'algoritmo di Ricart-Agrawala con n messaggi: n– 1 messaggi per richiedere il permesso agli altri processi e 1 messaggio per ottenere il permesso. Per far ciò si fa circolare un vettore *token*[1 . . . n] di timestamp in cui il *token*[i] indica ultima volta che il processo i ha avuto il token. Ed ogni processo usa un vettore *richiesta*[1 . . . n] in cui *richiesta*[i] indica l'ultima volta che il processo i ha richiesto il token. Di seguito spiegato nel dettaglio.

[inizialmente *token-presente* = **false** per tutti i processi tranne quello con il token]

```
if not token-presente then
  begin
    orologio := orologio + 1
    broadcast(richiesta, orologio, i)          ← trasmette a tutti i vicini
    wait(accesso, token)
    token-presente := true
  end
token-in-uso := true
. . . "sezione critica" . . .
token[i] := orologio
token-in-uso := false
for j := i mod n + 1 to (i – 2) mod n + 1 do   ← scansione circolare, principio di equità
  if richiesta[j] > token[j] and token-presente then   ← token richiesto dopo ultimo utilizzo
    begin
      token-presente := false
      send(accesso, token) to j                ← trasmette solo a j
    end
```

```
when received(richiesta, k, j) do
  begin
    richiesta[j] := max{richiesta[j], k}
    if token-presente and not token-in-uso then
      begin
        token[i] := orologio
        for j := i mod n + 1 to (i – 2) mod n + 1 do
          if richiesta[j] > token[j] and token-presente then
            begin
              token-presente := false
              send(accesso, token) to j
            end
      end
  end
```

In linea di massima il primo blocco di codice fa tre cose: prima richiede il token a tutti e si mette in attesa, quando lo riceve, esegue il codice nella sezione critica ed infine cerca tra tutti gli altri nodi se qualcuno ha richiesto il token ad un tempo superiore rispetto all'ultimo utilizzo che ne ha fatto (per escludere le richieste di token che sono già state soddisfatte).
L'assenza di stallo è data dal fatto che se tutti i processi sono bloccati il token è in transito ed alla ricezione un processo riparte.

Elezione del leader

Lo scopo è assegnare un privilegio permanente ad un processo, ipotizzando che ogni processo abbia un ID, e che tutti gli ID siano distinti. In una configurazione ad anello ciascun processo conosce solo il proprio ID e nessun processo conosce il numero n di nodi (processi) dell'anello. Il leader risulterà essere il processo con ID massimo. Infine è possibile che l'elezione sia indetta da più processi contemporaneamente. Vedremo due algoritmi: uno per l'anello unidirezionale (Chang-Roberts), e uno per anello bidirezionale (Hirschberg-Sinclair).

Elezione leader anello unidirezionale (Chang-Roberts)

```
when voglio indire l'elezione do
  begin
   partecipante := true
   send-right(elezione, mio-numero)
  end;

when received(elezione, j) do
  begin                          ← estinzione selettiva nel caso in cui j < mio-numero and partecipante
    if j > mio-numero then begin
      send-right(elezione, j)
      partecipante := false
    end
```

```
    if j < mio-numero and not partecipante then begin
      send-right(elezione, mio-numero)
      partecipante := true
    end;
    if j = mio-numero then
      send-right(eletto, mio-numero)
  end

when received(eletto, j) do
  begin
    leader := j
    partecipante := false
    if j ≠ mio-numero then
      send-right(eletto, j)
end
```

Il tempo è T = O(n) mentre il numero dei messaggi risulta essere M = $O(n^2)$ quando gli identificativi sono ordinati in modo decrescente e tutti indicono un'elezione nello stesso momento. ovvero ad ogni istante ognuno degli n nodi invia un messaggio, questo per n istanti (tempo necessario all'ID maggiore di raggiungere tutti e prendere il sopravvento). Il caso medio è M = O(n log n).

Elezione leader anello bidirezionale (Hirschberg-Sinclair)

Questo algoritmo si applica ad un anello bidirezionale, procede in fasi di "consultazione" (h = 1, 2, 3, . . .). Nella generica fase h, P_j trasmette la "candidatura" j ai vicini, affinché il messaggio giunga fino a distanza 2^h da P_j e poi torni indietro una risposta affermativa. Quando P_i riceve il valore del candidato j, lo confronta con i, se i > j allora P_i sostituisce P_j come "candidato"; se i < j allora P_j resta "candidato" e P_i è "sconfitto". Se P_i non riceve entrambe le risposte affermative si dichiara "sconfitto", altrimenti consulta un numero doppio di processi. Un processo sconfitto si limita a ritrasmettere i messaggi che riceve. Se P_j riceve indietro il messaggio di candidatura col proprio identificatore, allora si proclama leader.
Le primitive di comunicazione sono:
- send-left-right: che trasmette a entrambi i vicini.
- pass: che passa un messaggio da un vicino all'altro.
- respond: che risponde al vicino che ha inviato il messaggio.

I messaggi sono:
(candidatura, numero, lung, maxlung) che indicano rispettivamente: il codice della fase di candidatura, il numero del processore candidato mittente, la distanza percorsa fino ad un certo momento e la massima distanza da percorrere.
(risposta, esito, numero) che indicano rispettivamente: il codice della fase di risposta, l'esito booleano alla richiesta di voto del candidato e il numero del processore candidato destinatario.

```
when voglio indire l'elezione do
  begin
```

```
    stato := “candidato”; maxlung := 1
    while stato = “candidato” do begin
      numrisp := 0; ok := true
      send-left-right(candidatura, mio-numero, 0, maxlung);
      wait(numrisp = 2);
      if ok then maxlung := 2 * maxlung
      else stato := “sconfitto”
    end
  end

when received(risposta, esito, numero) do
  begin
    if numero = mio-numero then begin
      numrisp := numrisp + 1;
      ok := ok and esito
    end else
      pass(risposta, esito, numero)
  end

when received(candidatura, numero, lung, maxlung) do
  begin
    if numero < mio-numero then begin          ← candidato inferiore: indico io elezione
      respond(risposta, false, numero)
      if stato = “non-coinvolto” then voglio indire l’elezione
    end
    if numero > mio-numero then begin          ← candidato superiore: io sconfitto e passo
      stato := “sconfitto”; lung := lung + 1
      if lung < maxlung then
        pass(candidatura, numero, lung, maxlung)
      else
        respond(risposta, true, numero);
    end
    if numero = mio-numero then                ← candidato uguale: dico a tutti eletto
      if stato ≠ “eletto” then begin
        stato := “eletto”; vincitore := mio-numero
        pass(eletto, mio-numero)
      end
  end

when received(eletto, numero) do
  begin
    if vincitore ≠ numero then begin           ← aggiorno il leader e passo
      pass(eletto, numero)
```

vincitore := numero
stato := “non-coinvolto”
end
end

Nella fase h, la lunghezza della catena è 2^h, un processo si candida se non è stato sconfitto nella fase precedente (h – 1) da un altro processo che dista da lui al più 2^{h-1}. Ogni $2^{h-1} + 1$ processi consecutivi può esserci al più un candidato. Il numero dei candidati sarà quindi sempre $\leq \lfloor n / (2^{h-1} + 1) \rfloor$, mentre il numero dei messaggi inviati da ogni candidato sempre 4 (1 di candidatura e 1 di risposta per entrambe le direzioni). Il numero dei messaggi totali sarà sempre $\leq 4\ 2^h \lfloor n / (2^{h-1} + 1) \rfloor \leq 8n$. per un totale messaggi di O(n log n).

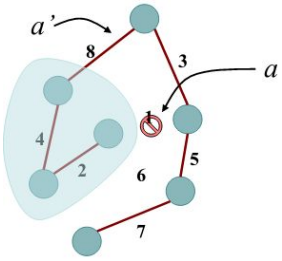
Elezione leader in reti anonime

Nelle reti anonime nè i nodi nè gli archi hanno valori identificativi. In linea generale se c’è già un leader, si possono assegnare gli ID tramite una visita. Se non è presente l’elezione del leader non è possibile, in un anello monodirezionale è possibile utilizzando un algoritmo probabilistico solo se tutti i nodi conoscono n, ma se i processi ignorano il numero di nodi dell’anello allora non è possibile calcolare n neanche con un algoritmo probabilistico.

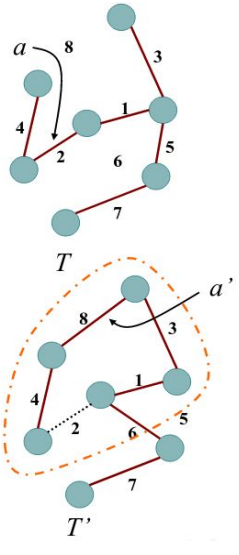
Minimum Spanning Tree di rete a grafo

Lo scopo è trovare un albero che connetta tutti i nodi, avente come somma di tutti i pesi degli archi il minimo possibile.

Un *frammento* di un MST è un sottoalbero connesso di copertura minimo.



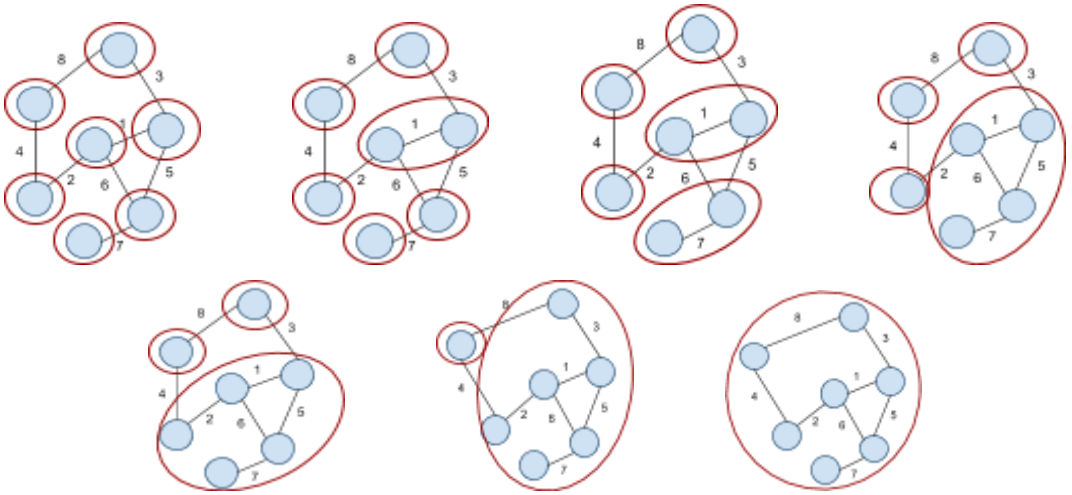
Proprietà 1: dato un frammento, se *a è il miglior arco uscente*, cioè l’arco di peso minimo con un nodo interno e l’altro esterno al frammento, allora aggiungendo a (e il nodo esterno) si ottiene ancora un frammento di un MST. Questo è banalmente dimostrabile in quanto se *per assurdo* l’arco a non appartenesse al MST ma vi appartenesse un altro ipotetico arco a’, allora aggiungendo anche a si formerebbe un circuito, cosa che dobbiamo evitare. Pertanto se lasciassimo a e togliessimo a’ per rimuovere il circuito l’albero ora ottenuto avrebbe costo inferiore al MST precedente, e questo crea una contraddizione dato che avevamo assunto che il MST non contenesse l’arco a bensì a’.



Proprietà 2: se tutti gli archi del grafo hanno pesi distinti, allora il MST è unico. Questo è dimostrabile facilmente, se per assurdo supponiamo che i MST siano due e non uno, diciamo T e T’, esisterà almeno un arco a che compare in uno (diciamo T) ma non nell’altro. Se aggiungiamo l’arco a al MST che non lo contiene (T’) si formerebbe un circuito, e a questo punto esiste un arco del circuito che non appartiene a T. Tuttavia dato che il peso di a è inferiore al peso di a’, sostituendo i due archi in T’ si

otterrebbe un MST di costo inferiore, e questa è una contraddizione.

In generale negli algoritmi per la ricerca del MST si parte con tanti frammenti formati da un solo nodo, i quali vengono ampliati man mano in un ordine qualsiasi. Per l'ampliamento di un frammento si aggiunge il nodo collegato all'arco uscente di peso inferiore (proprietà 1) e si combinano i frammenti che hanno un nodo in comune.



Minimum Spanning Tree distribuito

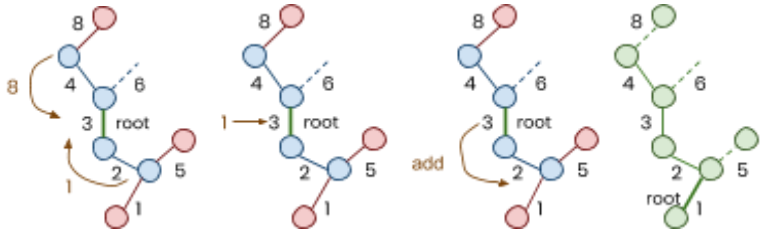
L'idea è quella di mantenere una foresta di copertura di frammenti, ovvero sotto-alberi dell'albero minimo. L'algoritmo inizia con una foresta di n frammenti, ciascuno contenente un solo nodo e nessun arco, e procede unendo coppie di frammenti per mezzo dell'arco di peso minimo che li collega, fino ad ottenere un solo frammento, che rappresenta il minimo albero di copertura. La versione distribuita prevede che più frammenti determinino il proprio minimo arco uscente contemporaneamente e che siano eseguite più unioni simultanee di coppie di frammenti. Vanno fatte due assunzioni: i pesi degli archi sono tutti distinti, in modo tale da evitare la formazione di circuiti a causa di unioni simultanee di frammenti; inoltre per evitare un caso pessimo in cui un frammento "grande" si unisca ripetutamente con uno "piccolo", generando $O(n^2)$ messaggi (un frammento di n/2 nodi che si collega n/2 volte con frammenti di un nodo) si usa la nozione di "livello" come stima della dimensione di un frammento. Pertanto i frammenti crescono simultaneamente per "livelli".

Ciascun frammento di livello L ha almeno 2^L nodi. In principio vi sono n frammenti di livello 0. Per ogni frammento F, ciascun nodo u di F conosce sia l'identità del frammento in cui si trova, sia quali archi incidenti in u sono o non sono in F. Per formare un frammento di livello L + 1 un generico frammento F di livello L procede come segue:

1. Su richiesta della radice del frammento, ciascun nodo del frammento cerca, tra gli archi in esso incidenti, l'arco di peso minimo uscente da F (se

esiste), chiedendo all'altro nodo dell'arco se appartiene o meno allo stesso frammento F.

2. Tutte le foglie di F spediscono verso la radice del frammento la lunghezza degli archi minimi uscenti da F trovati e i nodi intermedi ne calcolano il minimo, in modo tale che la radice abbia il minimo arco uscente da F.
3. La radice comunica con i nodi incidenti sul minimo arco uscente in modo tale da far sì che essi lo marchino come appartenente al nuovo frammento; il fatto che esista un unico arco minimo uscente comune a due frammenti deriva dall'assunzione di avere i pesi degli archi distinti.
4. Si stabilisce una nuova identità per il nuovo frammento, ad esempio scegliendo come identificativo del nuovo frammento il nuovo arco aggiunto e modificando di conseguenza la radice.



a) prima parte della fase 2; b) seconda parte; c) fase 3; d) fase 4.

Per diminuire la complessità nei casi in cui i frammenti siano di livelli diversi si fa in modo che quello di livello maggiore "assorba" quello di livello inferiore, piuttosto che "fondersi" in un nuovo frammento. In tal modo non è necessario propagare gli aggiornamenti a tutti i nodi, ma solo a quelli del frammento più piccolo. Riassumendo, si ha che:

Per bilanciare la crescita, ogni frammento F è caratterizzato da un livello L, tale che:

L(F)	=	**0**	se F contiene un solo nodo
L(F")	=	**L(F) + 1 = L(F') + 1**	se L(F) = L(F') si fondono e si crea F"
L(F')	=	**L(F)**	se L(F) > L(F'), F assorbe F'

L'assorbimento presenta un ulteriore vantaggio: se un frammento F scopre che il suo minimo arco uscente raggiunge un altro frammento F' di livello superiore [L(F) < L(F')], allora F può essere direttamente assorbito da F' senza attendere informazioni sul minimo arco uscente da F'.

Diversamente, se i due frammenti hanno lo stesso livello [L(F) = L(F')] prima di fondere i due frammenti è necessaria l'individuazione del minimo arco uscente comune ad essi.

Un ultimo problema è dovuto al fatto che, nel momento in cui un nodo u chiede ad un altro nodo v se si trova nel suo stesso frammento o meno, quest'ultimo potrebbe essere nello stesso frammento di u, ma ignorarlo al momento poiché non ha ancora ricevuto l'aggiornamento dell'identità dal frammento a cui è stato aggiunto. La soluzione consiste nel confrontare i livelli di u e di v: se il livello di v è

inferiore a quello di u, allora la risposta è ritardata finché il livello di v non diventa maggiore o uguale a quello di u. Quindi un nodo non da risposta di appartenenza o meno allo stesso frammento fino a quando il suo livello non è *almeno* uguale a quello del richiedente.

Considerazioni sui costi
Considerando un grafo con n nodi e m archi l'algoritmo prevede che un arco possa essere scartato una sola volta, quando entrambi i suoi nodi lo marcano come "rifiutato", al costo di 2 messaggi. Pertanto il numero totale di questi messaggi è O(m). Inoltre, poiché ogni livello L sarà sempre minore o uguale a log(n), dato che si raddoppia sempre la dimensione di ogni frammento, ogni nodo può passare al massimo per O(log n) livelli. Nell'ambito di un singolo livello, un nodo riceve al più un messaggio di inizio ricerca, al più un messaggio di accettazione in seguito ad uno di verifica, al più un messaggio di cambio/connessione, e invia al più un messaggio di minimo trovato, ovvero scambia O(1) messaggi. Ne consegue che il numero di messaggi scambiati in un solo livello è O(n). Considerando tutti gli O(log n) livelli e i messaggi di verifica e rifiuto, abbiamo un totale di **M = O(m + n log n)** messaggi. Per quanto riguarda la complessità in termini di tempo, si può dimostrare che il tempo necessario alla computazione è pari ad $O(n^2)$, che può essere ridotto a O(n log n) se si svegliano inizialmente tutti i nodi, cosa fattibile in tempo O(n).

- Identità dei frammenti
L'identità di un frammento è data dal peso di un suo arco, detto *nucleo*. Il nucleo di un frammento F'' è il miglior arco uscente comune sul quale si fondono due frammenti F ed F' allo stesso livello L = L'.

Ogni nodo ha uno stato, che può essere:
- ***addormentato***: lo stato iniziale.
- ***trova***: durante la ricerca del miglior arco uscente.
- ***trovato***: quando si è trovato il miglior arco uscente.

Se un nodo si sveglia dallo stato addormentato, trova il miglior arco uscente, lo marca ramo del MST, spedisce un messaggio *connetti* sull'arco, e passa allo stato trovato.

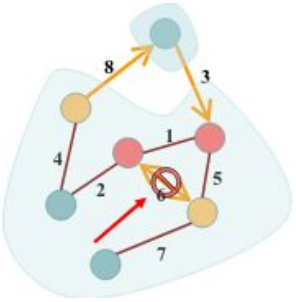
Un arco può essere classificato come:
- ***base***: se è ancora da classificare.
- ***rifiutato***: se è già interno al frammento.
- ***ramo***: se fa parte del MST.

Scoperta miglior arco uscente

1. Appena dopo una fusione tra due frammenti il peso del nucleo è l'identità del nuovo frammento. L'arco nucleo si comporta come la radice del frammento e i due estremi del nucleo fungono da coordinatori (leader).

Successivamente i due coordinatori spediscono un messaggio *inizia* verso le foglie, questo messaggio viene ritrasmesso dai nodi intermedi fino alle foglie. Tutti i nodi che ricevono *inizia* si portano nello stato *trova*.

2. Quando un nodo riceve un messaggio *inizia*, fa partire la ricerca del miglior arco uscente, spedendo un messaggio *verifica* sui rami base (per peso crescente). Quando un nodo riceve un messaggio *verifica*, le opzioni sono 3:
   a. Ha la stessa identità del frammento: allora spedisce un messaggio rifiuta e l'arco viene rifiutato.
   b. Il suo livello è inferiore a quello del mittente, ritarda la risposta finché il suo livello non è cresciuto.
   c. Il suo livello è maggiore di quello del mittente, spedisce un messaggio accetta e l'arco è accettato come candidato.
   d. Se *verifica* è stato spedito in entrambe le direzioni, l'arco è rifiutato senza messaggio rifiuta (fa già parte dello stesso frammento).



- Accordo sul miglior arco uscente
Per trovare un accordo sul miglior arco uscente ogni foglia spedisce *minimo* quando ha esaminato i suoi archi uscenti, ed ogni nodo interno spedisce *minimo* quando ha esaminato i suoi archi uscenti e ha ricevuto *minimo* da tutti i figli. Ogni nodo memorizza l'arco *ramo* da dove proviene il miglior arco uscente del suo sottoalbero, detto *migliore*. I due coordinatori si scambiano messaggi di *minimo* per accordarsi sul miglior arco uscente dei due sottoalberi (cioè di tutto il frammento). Quando hanno deciso, un messaggio *cambia* è spedito lungo gli archi *ramo* per raggiungere il miglior arco uscente. Gli archi *ramo* del cammino rovesciano i puntatori per puntare al nuovo nucleo, il miglior arco uscente diventa il nuovo nucleo e spedisce un messaggio connetti.

- Connessione di frammenti allo stesso livello
La connessione di frammenti allo stesso livello prevede che entrambi i nodi del miglior arco uscente spediscano messaggi *connetti*, che causano un incremento di livello. Inoltre diventano i nuovi coordinatori: il *nucleo* è cambiato e sono spediti messaggi *inizia*. In caso di frammenti a livello inferiore: se un frammento a livello inferiore (sul nodo u) è assorbito da un frammento a livello superiore (sul nodo v):
> Prima che v abbia spedito *minimo*: v spedisce ad u un messaggio inizia con lo stato *trova*, anche u si unisce alla ricerca del minimo arco uscente del frammento più grosso.
> Dopo che v abbia spedito *minimo*: v ha già trovato il miglior arco uscente, v non può spedire ad u un messaggio *inizia* con lo stato trovato e u non può unirsi alla ricerca del minimo arco uscente del frammento più grosso.

Algoritmo di Gallager-Humblet-Spira
Di seguito i prerequisiti per l'esecuzione dell'algoritmo: il grafo deve avere pesi finiti e distinti assegnati a ciascun arco. Ogni nodo conosce il peso per ogni arco incidente su quel nodo. Inizialmente, ogni nodo si trova in uno stato di quiescenza e

si sveglia spontaneamente o viene risvegliato dalla ricezione di un qualche messaggio da un altro nodo. I messaggi possono essere trasmessi indipendentemente in entrambe le direzioni su un arco e arrivano dopo un ritardo imprevedibile ma finito, senza errori. Ogni arco consegna messaggi in ordine FIFO.

I tipi di messaggi scambiati sono 7: *inizia, verifica, accetta, rifiuta, minimo, cambia* e *connetti*.
Si hanno inoltre 9 variabili mantenute da ogni nodo: *statonodo* che può assumere i valori ["addormentato", "trova", "trovato"], *statoarco* che può assumere i valori ["ramo", "rifiutato", "base"], *mpu* (miglior peso uscente), *migliore* (miglior arco uscente), *padre*, *candidato*, *F* (frammento), *L* (livello) e *conta-trova*.

Al *risveglio* il nodo setta tutte le variabili, aggiungendo subito il ramo con peso minimo al suo personale MST e inviando una richiesta di connessione tra il suo frammento (di un solo nodo) e quello del ramo di costo minimo.

```
when ti sei svegliato do SVEGLIA
procedure SVEGLIA
  begin
    e := arco [u, v] incidente in u con peso c[e] minimo
    statoarco[e] := "ramo"
    L := 0
    statonodo := "trovato"
    conta-trova := 0
    send(connetti, L) on e
  end
```

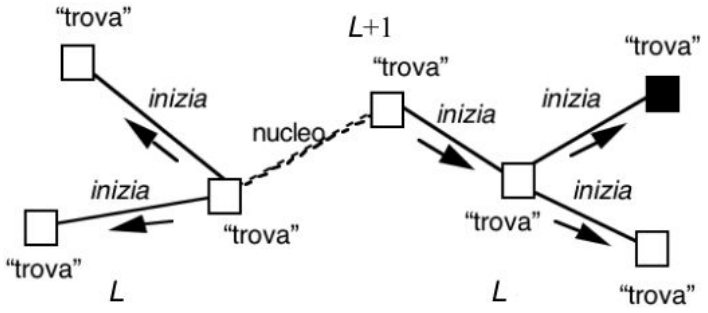
Quando un nodo riceve una *richiesta di connessione* su un arco, questo viene svegliato se dorme, se il nodo richiedente ha livello inferiore viene automaticamente unito al MST sull'arco di richiesta e sempre su questo arco viene mandato un messaggio inizia. Se il nodo richiedente ha livello maggiore o uguale invece ed è un arco base si attende a rispondere fino a quando non avrà livello maggiore o uguale al richiedente, se l'arco non è base si invia un messaggio inizia con lo stato "trova"..

```
when received(connetti, L') on a do
  begin
    if statonodo = "addormentato" then SVEGLIA;
    if L' < L then begin
      statoarco[a] := "ramo"
      send(inizia, L, F, statonodo) on a
      if statonodo = "trova" then
        conta-trova := conta-trova + 1
    end else
```

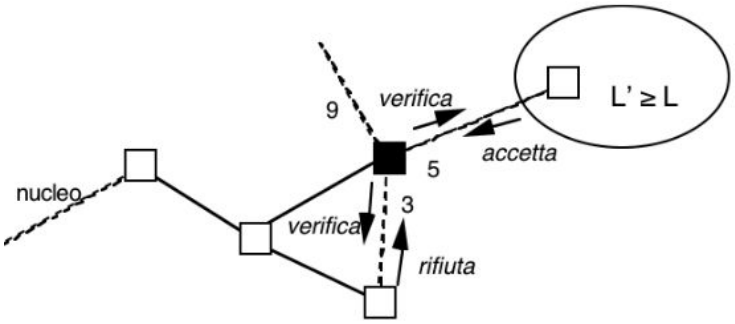
```
    if statoarco[a] := "base" then
      reinserisci il messaggio in fondo alla coda
    else
      send(inizia, L + 1, c[a], "trova") on a
  end
```

Quando si riceve una richiesta di inizio, il nodo che la riceve imposta livello, frammento di riferimento e stato come quelli che gli vengono passati, inoltre il padre del nodo è impostato all'arco su cui la richiesta è stata ricevuta. Poi per tutti i rami che partono dal nodo trasmette a sua volta la richiesta inizia e conta.

```
when received(inizia, L', F', S) on a do
  begin
    L := L'; F := F'; statonodo := S
    padre := a; migliore := λ ; mpu := ∞
    for each e such that e ≠ a and statoarco[e] = "ramo" do begin
      send(inizia, F', L', S) on e
      if S = "trova" then conta-trova := conta-trova + 1
    end
    if S = "trova" then TEST
  end
```

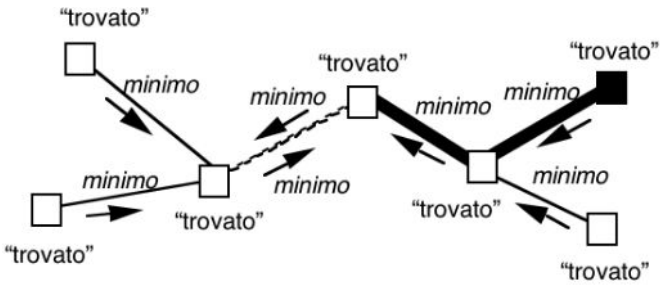



```
procedure TEST
  begin
    if esiste un arco incidente in u con stato "base" then begin
      candidato := arco incidente in u con stato "base" e peso minimo
      send(verifica, L, F) on candidato
    end else begin
      candidato := λ
      TROVATO-MINIMO
    end
  end
```

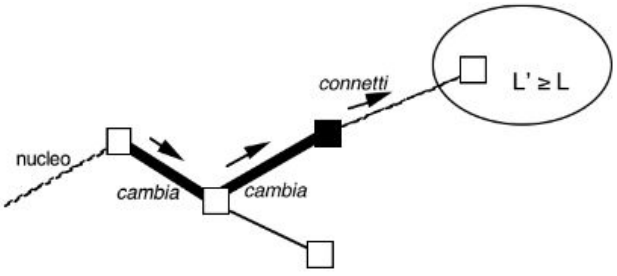



```
when received(verifica, L', F') on a do
  begin
    if statonodo = "addormentato" then SVEGLIA
    if L < L' then
      reinserisci il messaggio in fondo alla coda
    else if F ≠ F' then
      send(accetta) on a
    else begin
      if statoarco[a] = "base" then statoarco[a] := "rifiutato"
      if candidato ≠ a then
        send(rifiuta) on a
        else TEST
    end
  end

when received(accetta) on a do
  begin
    candidato := λ
    if c[a] < mpu then begin
      migliore := a
      mpu := c[a]
    end
    TROVATO-MINIMO
  end;

when received(rifiuta) on a do
  begin
    if statoarco[a] = "base" then
      statoarco[a] := "rifiutato"
    TEST
  end

procedure TROVATO-MINIMO
  begin
    if conta-trova = 0 and candidato = λ then begin
      statonodo := "trovato"
      send(minimo, mpu) on padre
```

```
    end
  end
```




```
when received(minimo, peso) on a do
  begin
    if a ≠ padre then begin
      conta-trova := conta-trova – 1
      if peso < mpu then begin
        mpu := peso
        migliore := a
      end
      TROVATO-MINIMO
    end else if statonodo = "trova" then
      reinserisci il messaggio in fondo alla coda
    else if peso > mpu then
      AGGIORNA-FRAMMENTI
    else if peso = mpu = ∞ then
      termina l'esecuzione
  end

when received(cambia) do AGGIORNA-FRAMMENTI

procedure AGGIORNA-FRAMMENTI
  begin
    if statoarco[migliore] = "ramo" then
      send(cambia) on migliore
    else begin
      send(connetti, L) on migliore
      statoarco[migliore] := "ramo"
    end
  end
```



In breve

- *Ricerca del minimo arco uscente*:

Tutto inizia quando i coordinatori (i nodi ai lati dell'arco considerato radice) inviano alle loro metà di frammento un messaggio INIZIA, il quale verrà inoltrato da ogni nodo interno su ogni altro arco incidente fino alle foglie. Ogni nodo che riceve questo messaggio ha l'obiettivo di trovare il minimo arco uscente tra i suoi archi BASE, ovvero quelli non già appartenenti al MST. Per far ciò invia su questi archi un messaggio VERIFICA (col suo F ed L). Il nodo che riceve questo messaggio ha 2 opzioni: o ritardare la risposta per assenza di informazioni sufficienti o rispondere subito. Il primo caso avviene quando il livello del nodo destinatario è inferiore a quello del mittente, in quanto il destinatario potrebbe non sapere ancora il frammento a cui appartiene. Il secondo caso invece prevede una risposta immediata e questa sarà ACCETTA se i due frammenti sono diversi o RIFIUTA dei i due frammenti sono uguali (per evitare cicli strani all'interno dello stesso frammento). Quando il nodo mittente riceve un RIFIUTA smette di considerare definitivamente l'arco (perchè già interno al frammento), se invece riceve un ACCETTA prende in considerazione il peso dell'arco confrontandolo con i pesi degli altri archi incidenti che hanno risposto positivamente alla verifica, e con i pesi minimi dei loro archi incidenti (e così via); identificando il minimo tra questi valori il nodo identifica l'arco minimo della porzione di frammento che passa da lui. Trovato questo valore MINIMO, viene inviato al padre il quale farà la stessa operazione, scremando i pesi minimi ad ogni arco fino ai coordinatori, i quali riceveranno i due archi di peso minimo in ognuna delle due metà del frammento. Si scambiano i valori tra loro due identificando di fatto il minimo arco uscente del frammento.

- *Aggiunta dell'arco minimo al MST*:

A questo punto si fa passare un messaggio CAMBIA lungo il percorso interno al frammento che conduce fino all'arco di peso minimo, il quale viene ufficialmente trasformato da BASE a RAMO del MST. Ora è necessario unire i due frammenti ai lati del minimo arco uscente appena trovato.

- *Unione dei due frammenti*:

Il nodo da cui esce il minimo arco trovato invia un messaggio CONNETTI al nodo all'altro capo dell'arco (che sarà ovviamente in un altro frammento), il quale riceverà il livello e il frammento del mittente. Se il destinatario ha livello superiore, viene semplicemente assorbito il frammento del mittente (che ha livello inferiore). Se invece il destinatario ha livello uguale o inferiore al mittente vi sono due casi: se l'arco NON è il minimo arco uscente anche del frammento destinatario viene ritardata la risposta in quanto potrebbe in futuro diventare il minimo arco uscente; se l'arco È il minimo arco uscente anche del frammento destinatario avviene una fusione tra i frammenti, che incrementeranno di 1 il livello e porranno l'arco come radice.

Al termine dell'unione viene in ogni caso inviato un nuovo messaggio INIZIA e tutto si ripete.

Ultime considerazioni
Se il problema è **NP-arduo**, resta intrattabile anche in parallelo.
Se il problema è **Log-spazio-completo**, è inerentemente sequenziale. Non ci sono algoritmi con tempo polilogaritmico ma si possono trovare algoritmi paralleli con tempo polinomiale inferiore a quello sequenziale (es. flusso massimo).