# 1. ALGORITMI PARALLELI

In determinate situazioni, la velocità di esecuzione di un compito è l'elemento critico: si pensi alle applicazioni al tempo reale, in cui ad esempio il calcolo della risposta di un filtro deve essere fatto all'interno dell'intervallo di campionamento. In questi casi può essere utile far eseguire il compito a più processori, sperando che l'aumento di risorse sia compensato da una diminuzione del tempo di calcolo. I modelli di calcolo in cui siano presenti più processori che lavorano contemporaneamente sono detti *paralleli*, così come il modello dotato di un solo processore è detto *sequenziale*. In questa sezione considereremo modelli di calcolo paralleli in cui i processori possono essere sincronizzati con un clock globale.

**Esempio 1.1**: Se un uomo con un secchio svuota una vasca in un tempo t, ci si può aspettare che p uomini svuotino la stessa vasca in un tempo dell'ordine di t/p. La presenza di più esecutori che lavorano contemporaneamente può, in certi casi, ridurre il tempo necessario ad eseguire un compito.

**Esempio 1.2**: Re Sole soleva farsi vestire dalla servitù. E' possibile sfruttare un arbitrario numero p di addetti per ridurre il tempo di un fattore p?

**Esempio 1.3**: Si voglia risolvere il problema PROGRAMMAZIONE LINEARE:

Istanza : un vettore a, una matrice A

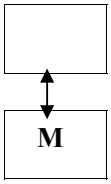
Risposta : Max $\sum_{i=1,n} a_i x_i$ coi vincoli $\sum_{i=1,n} x_i . A_{ik} \geq 0$ (k=1,m)

Questo problema può essere risolto con un algoritmo sequenziale in tempo $O(n^2)$. E' possibile ridurre drasticamente il tempo di esecuzione, per esempio ottenendo tempi $O(\log^2 n)$, sfruttando la presenza di O(n) processori?

I tre problemi precedenti mostrano caratteristiche diverse per quanto riguarda l'esecuzione parallela. Nel primo caso, la presenza di "più processori" può essere sfruttata per ridurre il tempo di esecuzione, nel secondo e terzo caso non si vede come si possano far lavorare i processori in modo che si riesca a ottenere la drastica riduzione di tempo richiesta: apparentemente, il parallelismo è utile per alcuni tipi di problemi, meno per altri. In questo corso non ci occuperemo tuttavia dell'interessante problematica di individuazione dei problemi che possono essere risolti efficientemente con sistemi paralleli.

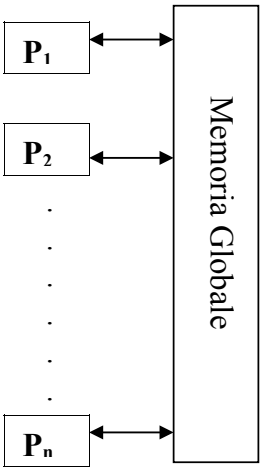
Ci limitiamo qui a introdurre alcuni modelli di calcolo parallelo, in cui i processori possono essere sincronizzati con un clock globale, e a presentare per tali modelli algoritmi risolutivi di semplici problemi.

Richiamiamo per prima cosa un classico modello di calcolo sequenziale, cioè la *macchina RAM*; ridotta all'osso, essa consiste di un processore P collegato ad una memoria M attraverso una unità di accesso:

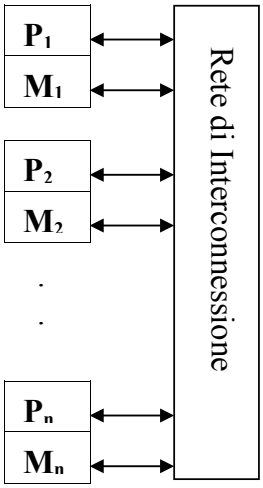



Macchina RAM

Nei modelli di calcolo parallelo, data la presenza di più processori, un elemento critico è la modalità di comunicazione tra processori. I due casi limite sono rappresentati dal modello a *memoria condivisa* e dal modello a *memoria distribuita*.




Memoria condivisa




Memoria distribuita

Nel modello a memoria condivisa tutti i processori possono accedere, attraverso una unità di accesso, alle stesse locazioni di memoria nella stessa unità di tempo. Il meccanismo di comunicazione tra due processori $P_k$ e $P_j$ è molto semplice:

1. $P_k$ scrive il messaggio in un'area di memoria
2. $P_j$ lo legge dalla stessa area

Va segnalato che i processori non sono direttamente connessi e che la comunicazione avviene in tempo costante O(1).
Sfortunatamente, è oggi difficile realizzare tale modello con un adeguato numero di processori; è tuttavia interessante riferirsi a questo modello di calcolo parallelo perché, trascurando i tempi di comunicazione, esso permette all'utente di porre l'attenzione sul potenziale parallelismo disponibile nella ricerca di soluzioni efficienti ad un dato problema.

Nei modelli a memoria distribuita ogni processore può accedere invece solo alla sua memoria privata (locale). La comunicazione avviene qui attraverso l'invio di messaggi attraverso una rete di interconnessione, che può essere descritta da un grafo i cui vertici sono i processori e i lati rappresentano collegamenti diretti tra processori. Poiché in tali reti non tutti i processori sono collegati direttamente, non è possibile ipotizzare tempi di comunicazione costanti, contrariamente a quel che succedeva nel caso a memoria condivisa.

Nelle prossime sezioni presenteremo in qualche dettaglio il modello PRAM (Parallel Random Access Machines), uno dei più popolari modelli a memoria condivisa. In particolare, risolvendo semplici esempi quali SOMMATORIA, SOMME PREFISSE, ORDINAMENTO, sviluppiamo alcune idee per il progetto di algoritmi paralleli; viene infine descritta una tecnica detta "CICLO EULERIANO".

Verranno poi discusse alcune topologie di interconnessione (array lineare, mesh quadrata, albero, ipercubo). Di esse saranno analizzati alcuni parametri critici (grado, diametro, ampiezza di bisezione) e le prestazioni, su problemi *benchmark* quali MASSIMO e ORDINAMENTO, di algoritmi implementati nelle varie topologie.

## 2. MODELLO P-RAM

Il più semplice modello di calcolo parallelo è quello a memoria condivisa, detto P-RAM, che discutiamo in questo paragrafo. Tale modello consiste di *p-processori* con una memoria globale, condivisa da tutti.

La memoria globale è usata dai processori per scambiarsi dati *in tempo* O(1): perché il processore k e il processore j si scambino un valore, basta che il processore k scriva tale valore in una variabile condivisa e il processore j vi acceda in lettura.

Il calcolo procede per *passi*. Ad ogni passo ogni processore può fare una operazione sui dati che possiede, oppure può leggere o scrivere nella memoria condivisa. In particolare, è possibile selezionare un insieme di processori che eseguono tutti la stessa istruzione (su dati generalmente diversi), mentre gli altri processori restano inattivi; i processori attivi sono sincronizzati, nel senso che eseguono la stessa istruzione simultaneamente e l'istruzione successiva può essere eseguita solo quando tutti hanno terminato l'esecuzione.

Questo modello di macchine è detto di tipo SIMD (Single Instruction Multiple Data). Si possono ulteriormente specificare vari modelli di PRAM, in termini di limitazione agli accessi a memoria condivisa:

1. EREW (Exclusive Read Exclusive Write): non e' ammesso l'accesso contemporaneo da parte di più processori alla stessa locazione di memoria.
2. CREW (Concurrent Read Exclusive Write): l'accesso contemporaneo è permesso in lettura.
3. CRCW (Concurrent Read Concurrent Write): l'accesso contemporaneo è permesso in lettura ed in scrittura.

Per semplicità, in seguito faremo riferimento al modello EREW, il più realistico: in questo caso è compito del programmatore disegnare l'algoritmo in modo che non accadano conflitti in lettura o scrittura. In seguito denoteremo con l'intero i il processore $P_i$.

La tipica istruzione di un algoritmo parallelo è la seguente:

*for all* $i\,(a \le i \le b)$ *do in parallel* *Operazione*

la cui semantica è quella di eseguire *Operazione* su tutti i processori *i* con $a \le i \le b$, mentre gli altri restano inattivi.

Dato un algoritmo parallelo *A*, diremo $T_A(n, p)$ il tempo di esecuzione di *A* su dati di dimensione *n*, quando *A* utilizza p processori; il caso sequenziale si ha ovviamente

quando $p=1$. Obbiettivo di un algoritmo parallelo è quello di diminuire i tempi di calcolo aumentando il numero di processori: si spera allora in un *trade-off* tra il tempo e il costo (numero di processori) che va stimato per valutare l'efficacia di un algoritmo parallelo. Due misure naturali dell'efficacia sono lo *Speed-up* $S_A(p)$ e l'*Efficienza* $E_A(n,p)$:

$$S_A = \frac{\mathrm{T}_A(n,1)}{\mathrm{T}_A(n,p)}$$
$$E_A = \frac{S_A(p)}{p}$$

Poiché $E_A(n,p) = \frac{\mathrm{T}_A(n,1)}{p \cdot \mathrm{T}_A(n,p)}$, l'efficienza risulta essere il rapporto tra il tempo dell'algoritmo sequenziale e il tempo totale consumato dai processori, come se fossero usati sequenzialmente.

Dato un algoritmo A che lavora con p processori con una data efficienza E, è in generale possibile estendere l'algoritmo a lavorare con un numero inferiore di processori senza che l'efficienza diminuisca significativamente:

**Fatto**: se k>1, allora $E_A\left(n, \frac{p}{k}\right) \geq E_A(n,p)$

Dato un algoritmo *A* che lavora con *p* processori, basta infatti costruire un algoritmo modificato che utilizza p/k processori. Ad ogni nuovo processore si fa corrispondere un blocco di k vecchi processori: ogni nuovo processore usa al più k passi per emulare il singolo passo parallelo dei k processori corrispondenti. Vale quindi:

$$\mathrm{T}_A\left(n, \frac{p}{k}\right) \leq k \cdot \mathrm{T}_A(n,p)$$

Osservando che $E_A\left(n, \frac{p}{k}\right) = \frac{\mathrm{T}_A(n,1)}{\frac{p}{k} \cdot \mathrm{T}_A\left(n, \frac{p}{k}\right)} \geq \frac{\mathrm{T}_A(n,1)}{p \cdot \mathrm{T}_A(n,p)} = E_A(n,p)$, concludiamo che l'efficienza non diminuisce diminuendo i processori. In particolare, poichè $E_A(n,p) \leq E_A(n,1) = 1$, l'efficienza non può superare 1.

## 3. ALGORITMI SU P-RAM

In questa sezione presentiamo alcuni elementi di dprogetto di algoritmi paralleli, sviluppando sulla base di esempi algoritmi paralleli per semplici compiti. Trattiamo in particolare due problemi (SOMMATORIA e ORDINAMENTO) e alcune varianti (SOMMA PREFISSA). Presentiamo algoritmi per SOMMATORIA e SOMME PREFISSE che lavorano in tempo O(log n) e un algoritmo, ordinamento bitonico, che ordina un vettore di n elementi in tempo $O(\log^2 n)$. Discutiamo infine una tecnica per il disegno di algoritmi su alberi detta *cammino eluleriano*.

### Esempio 3.1: problema SOMMATORIA.

Il problema SOMMATORIA può essere così formulato:

Istanza: interi *a[1], a[2], .., a[n]*

Risposta: $S = \sum_{K=1}^{n} a[k]$

In seguito supponiamo per semplicità che n sia potenza di 2: $n = 2^S$, per qualche s; non c'è perdita di generalità, poiché tra n e 2n c'è sempre una potenza di 2.

Per delineare un primo algoritmo, supponiamo di avere a disposizione $\frac{n}{2}$ processori, detti rispettivamente processore 1, processore 2,.., processore $\frac{n}{2}$. Supponiamo che inizialmente gli interi *a[1], a[2], .., a[n]* siano memorizzati nell'array $(x[1],..,x[n])$. Si può ottenere una soluzione naturale al problema SOMMATORIA procedendo come segue:

al passo 1, ogni processore k $\left(1 \le k \le \frac{n}{2}\right)$ calcola $x[2k] = x[2k] + x[2k-1]$

al passo 2, ogni processore k $\left(1 \le k \le \frac{n}{4}\right)$ calcola $x[4k] = x[4k] + x[4k-2]$

. . .

. . .

al passo s, il processore k $(k = 1)$ calcola $x\left[2^S k\right] = x\left[2^S k\right] + x\left[2^S k - 2^{S-1}\right]$

Questa idea è concretizzata dal seguente algoritmo parallelo:

*ALGORITMO 1( interi a[1],.., a[n] memorizzati in* $(x[1],..,x[n])$*)*

*for* $j = 1, \log_2 n$ *do*

*for all* $k\left(1 \le k \le \frac{n}{2^j}\right)$ *do in parallel*

$$x\left[2^j k\right] = x\left[2^j k\right] + x\left[2^j k - 2^{j-1}\right]$$

*return* $x[n]$

La correttezza dell'algoritmo è provata dimostrando per induzione la verità delle seguenti due affermazioni:

- non vi sono conflitti tra i processori, nè in lettura nè in scrittura.
- al passo j per ogni k$\left(1 \le k \le \frac{n}{2^j}\right)$ la variabile $x\left[2^j k\right]$ memorizza il valore $a\left[2^j k\right] + ... + a\left[2^{j-1}(k-1)+1\right]$

La prova per induzione richiede di provare l'asserzione per j=1 e di provarla per j, supponendola vera per j-1.

Si verifica direttamente che primo passo in $x[2k]$ viene memorizzato $a[2k+1] + a[2k]$. Al passo j viene eseguita l'istruzione $x\left[2^j k\right] = x\left[2^{j-1}.2k\right] + x\left[2^{j-1}(2k-1)\right]$. Supponendo che, per ipotesi di induzione, al passo j-1 $x[2^{j-1}.2k]$ contenga $a\left[2^{j-1}2k\right] + ... + a\left[2^{j-1}(2k-1)+1\right]$ e $x\left[2^{j-1}(2k-1)\right]$ contenga $a\left[2^{j-1}(2k-1)\right] + ... + a\left[2^{j-1}(2k-2)+1\right]$, concludiamo che al passo j la variabile $x\left[2^j k\right]$ contiene $a\left[2^{j-1}2k\right] + ... + a\left[2^{j-1}(2k-2)+1\right] = a\left[2^j k\right] + ... + a\left[2^j(k-1)+1\right]$.

Poiché al passo $\log_2$ n la variabile $x[n]$ contiene $a[n] + ... + a[1]$, concludiamo che l'algoritmo risolve il problema in tempo $\log_2 n$ utilizzando n/2 processori:

$$T_{ALG1}\left(n, \frac{n}{2}\right) = \log_2 n$$

Poiché nel caso sequenziale il tempo richiesto è n-1, l'efficienza risulta:

$$E_{ALG1}\left(n, \frac{n}{2}\right) = \frac{n-1}{\frac{n}{2}.\log_2 n} \approx \frac{2}{\log_2 n}$$

L'efficienza di questo algoritmo tende quindi a 0 per n tendente a $\infty$, sia pur molto lentamente.

Allo scopo di diminuire il numero di processori e nel contempo di migliorare l'efficienza, si osserva che il precedente algoritmo utilizza n/2 processori che lavorano tutti solo nella prima fase del calcolo. Progettiamo allora un secondo

algoritmo ( *ALGORITMO 2* ) che lavora con p<n/2 processori e che sarà più lento solo nella prima fase. Di questo algoritmo diamo qui una descrizione informale.

*ALGORITMO 2*

- *Posto* $\Delta = \left\lceil \frac{n}{p} \right\rceil$, *viene assegna al processore k (1≤k≤p) il sottovettore* $(x[\Delta \cdot (k-1)+1].....x[\Delta \cdot k])$ : *lavorando in parallelo con gli altri, in tempo* $\Delta$ *il processore k calcola sequenzialmente la somma* $S[k]$ *delle componenti di tale vettore in tempo* $\Delta$*, ponendo il risultato in* $x[k \cdot \Delta]$.
- *Si applica ora ALGORTMO 1 al vettore* $(x[\Delta],..,x[p\Delta])$*, ottenendo la risposta corretta in tempo* $\log_2 p$ .

Il tempo complessivamente usato da *ALGORITMO 2* è dato da:

$$T_{ALG2}(n,p) = \frac{n}{p} + \log_2 p$$

La sua efficienza risulta inoltre $E_{ALG2}(n,p) = \frac{n-1}{n + p\log_2 p}$

Scegliendo p in modo tale che $p \cdot \log_2 p = n$, si hanno le due conseguenze:

- $E_{ALG2} = \frac{n-1}{2n} \approx \frac{1}{2}$
- $p \cdot \log_2 p = n$ implica $p \approx \frac{n}{\log_2 n}$

*ALGORITMO 2* lavora dunque con $\frac{n}{\log_2 n}$ processori, ha un tempo di calcolo $\log_2 n + \log_2 p \le 2\log_2 n$ ed una efficienza pari a ½. Il tempo è allora logaritmico $(2\log_2 n)$ con un numero sublineare di processori $\left(\frac{n}{\log_2 n}\right)$.

**Esempio 3.2: problema SOMME PREFISSE**

Affrontiamo ora una estensione del problema SOMMATORIA, che chiameremo SOMME PREFISSE, così definito:

SOMME PREFISSE:
Istanza: interi *a[1], a[2], .., a[n]*
Risposta: *S[1] = a[1], S[2]=a[1]+a[2], ..., S[n]= a[1]+a[2]+ ...+ a[n]*

La risposta è quindi costituita dai valori $S[k] = a[1] + .. + a[k]$, per ogni *k* da 1 a n. Questo problema può essere considerato un archetipo di vari problemi su liste.

Supponiamo che i numeri *a[1], a[2], .., a[n]* siano inizialmente memorizzati nell'array M=$(M[1], M[2], ..., M[n])$, e utilizziamo una tabella $succ[k]$ che useremo per indirizzare ad M. Inizialmente, la tabella *succ* è data da:

$$succ[k] = \begin{cases} k+1 & k \le n \\ 0 & \mathrm{k} = n \end{cases}$$

Anche in questo caso, senza perdita di generalità, possiamo ipotizzare che n sia una potenza di 2. Una soluzione parallela per macchine EREW con n processori è la seguente:

*ALGORITMO SOMMEPREFISSE( interi a[1],.., a[n] memorizzati in* $(M[1],..,M[n])$*)*

*for* $k = 1, \log n$ *do*
  *for all* $\mathrm{k}(1 \le \mathrm{k} \le n)$ *do in parallel* :
    *if* $succ[k] \neq 0$ *do* $\begin{cases} M[succ[k]] = M[k] + M[succ[k]] \\ succ[k] = succ[succ[k]] \end{cases}$
*risposta* : $(M[1], M[2], ..., M[n])$

La correttezza dell'algoritmo è provata dalle seguenti considerazioni.

- Si osserva per prima cosa che, dopo *t* passi, la tabella *succ* è tale che:
  $$succ[k] = \begin{cases} k + 2^t & k + 2^t \le n \\ 0 & \text{altrimenti} \end{cases}$$
  Inizialmente *succ* descrive quindi una lista di *n* elementi; dopo *t* iterazioni essa descrive $2^t$ liste di $\frac{n}{2^t}$ elementi e quindi, in particolare, dopo $\log_2 n$ iterazioni descrive *n* liste di 1 elemento.

- Si prova poi, per induzione, che alla iterazione *t* il valore M[k]= è:
  $$M[k] = \begin{cases} a_K + .. + a_{K+1-2\hat{}t} & se \quad k > 2^t \\ a_K + .. + a_1 & se \quad k \le 2^t \end{cases}$$

- Osserviamo infine che, durante l'esecuzione, il vettore *succ* è tale che, se $i \neq j$, $succ(i) \neq 0$, $succ(j) \neq 0$, allora $succ(i) \neq succ(j)$: non si hanno quindi conflitti in lettura nell'esecuzione dell'algoritmo.

Poiché la complessità in tempo è $O(\log_2 n)$ e il numero di processori è *n*, l'efficienza risulta essere $O\left(\frac{1}{\log_2 n}\right)$.

**Esempio 3.3: problema ORDINAMENTO**

Il problema ORDINAMENTO può essere così formulato:

Istanza: un vettore $(A[1],\ldots,A[n])$, che qui consideriamo, per semplicità, a componenti intere positive tutte diverse tra loro; supponiamo inoltre che $n$ sia potenza di 2.
Risposta: una permutazione $p(1), .., p(n)$ tale che $A[p(1)]<\ldots<A[p(n)]$.

Ci limitiamo a considerare algoritmi di ordinamento per confronto. E' noto che, nel caso sequenziale, ogni algoritmo di ordinamento richiede almeno $\Omega(n.\log n)$ confronti ed esistono algoritmi sequenziali che richiedono al più $O(n.\log n)$ confronti.

Uno di questi algoritmi è il cosiddetto SortMerge, che è una classica applicazione della tecnica "divide et impera". L'idea essenziale di tale algoritmo è di risolvere banalmente il problema per n=2, mentre per n>2:

- avendo in ingresso $(A[1],\ldots,A[n])$, si divide il vettore in due vettori uguali $A_s=(A[1],\ldots,A[n/2])$ e $A_d=(A[1+n/2],\ldots,A[n])$, che vengono ordinati ricorsivamente.
- I due vettori ordinati vengono utilizzati per costruire l'ordinamento del vettore originale, applicando l'operazione di Merge.

Il punto centrale è che, avendo due vettori già ordinati di dimensione n/2, l'operazione di Merge si effettua efficientemente (in ambito sequenziale) in tempo O(n). Se cerchiamo di parallelizzare il precedente algoritmo in maniera efficiente, vediamo che la prima fase non presenta problemi se, avendo n processori, assegniamo ai primi n/2 il primo vettore, ai secondi n/2 il secondo vettore. La difficoltà sta nel trovare un efficiente algoritmo parallelo per l'operazione di Merge, poiché l'algoritmo sequenziale non appare parallelizzabile.

Discutiamo qui un'idea generale che permette di ottenere algoritmo parallelo efficiente per l'ordinamento. Si procederà con una esposizione ad alto livello, che renda chiara l'idea evitando dettagli in prima approssimazione trascurabili. Per prima cosa, consideriamo le due operazioni di giustapposizione e trasposizione:

- Date due sequenze $A=(A[1],\ldots,A[n])$ e $B=(B[1],\ldots,B[m])$, la loro giustapposizione $A \bullet B$ è la sequenza $A \bullet B=(A[1],..,A[n],B[1],\ldots,B[m])$.
- Data una sequenza $A=(A[1],\ldots,A[n])$, la sua trasposta $A^R=(A[n],\ldots,A[1])$.

Osserviamo che, avendo a disposizione un adeguato numero di processori, tali operazioni su P-RAM si effettuano in tempo parallelo O(1).

Ritornando all'algoritmo SORTMERGE, se i vettori $A_s=(A[1],\ldots,A[n/2])$ e $A_d=(A[1+n/2],\ldots,A[n])$ sono già stati ordinati, basterà calcolare $B= A_s \bullet (A_d)^R$ in tempo O(1) e ordinare $B$. Il punto chiave consiste nell'osservare che $B$ è una sequenza

*bitonica* e che le sequenze bitoniche possono essere ordinate con un algoritmo parallelo efficiente.

A tal riguardo, diremo che la sequenza $(A[1],\ldots,A[n])$ è *unimodale* se esiste k tale che $A[1]<..<A[k])>A[k+1]>..>A[n])$ oppure $A[1]>..>A[k])<A[k+1]<\ldots<A[n]),$ mentre $(A[1], \ldots,A[n])$ è *bitonica* se esiste una sua permutazione circolare che è unimodale, cioè se esiste j per cui la sequenza $(A[j],\ldots A[n], A[1],\ldots,A[j-1])$ è unimodale.

Ad esempio, la sequenza (2,3,7,8,4) è unimodale e quindi bitonica, mentre (7,8,4,2,3) non è unimodale ma bitonica, perché la sua permutazione circolare (2,3,7,8,4) è unimodale; la sequenza (1,6,2,5,3,4) non è invece nemmeno bitonica. Si osservi che, se $A_s$ e $A_d$ sono sequenze crescenti, allora $A_s \bullet (A_d)^R$ è unimodale e quindi bitonica.

Data una sequenza $A=(A[1],\ldots,A[2n])$ di 2n elementi, diremo $A_{MAX}$ e $A_{MIN}$ le sequenze $(A_{MAX}[1],\ldots, A_{MAX}[n])$ e $(A_{MIN}[1],\ldots, A_{MIN}[n])$ di n elementi dove:

- $A_{MAX}[k]) = Max\{A[k], A[n+k]\}$ $(k=1,n)$
- $A_{MIN}[k]) = Min\{A[k], A[n+k]\}$ $(k=1,n).$

Per sequenze bitoniche, si possono verificare le seguenti proprietà:

**Fatto 3.1:** Se $A$ è bitonica, allora:

1. Ogni elemento di $A_{MIN}$ è minore di ogni elemento di $A_{MAX}$.
2. $A_{MIN}$ e $A_{MAX}$ sono bitoniche.

Queste proprietà suggeriscono una semplice procedura che, avendo in ingresso una sequenza bitonica, dà in uscita la sequenza ordinata in modo crescente:

*Procedura BitonicMerge(A: un sequenza bitonica di $n=2^k$ elementi)*
*If n=2 then return* $(A_{MIN}[1], A_{MAX}[1])$
*else* 1. $A_1$= BitonicMerge $(A_{MIN})$
2. $A_2$= BitonicMerge $(A_{MAX})$
return $(A_1 \bullet A_2)$

La correttezza è dimostrata immediatamente per induzione ricordando che se $A$ è bitonica $A_{MIN}$ e $A_{MAX}$ sono a loro volta bitoniche, e tutti gli elementi di $A_{MIN}$ sono minori di quelli di $A_{MAX}$. Per quanto riguarda il tempo di esecuzione parallelo $T_{BM}(n)$ di una implementazione del precedente algoritmo su PRAM con n processori, pur senza entrare nei dettagli implementativi si può osservare che il calcolo di $A_{MIN}$ e $A_{MAX}$ a partire da $A$ può essere effettuto in tempo costante O(1) e i passi 1. e 2. possono essere eseguiti in parallelo. Vale quindi:

$$T_{BM}(n)= T_{BM}(n/2)+O(1)$$

La soluzione della precedente equazione di ricorrenza è $T_{BM}(n) = O(\log n)$: su PRAM con n processori una sequenza bitonica può essere ordinata in tempo O(log n).

Utilizzando la procedura BitonicMerge si ottiene il seguente algoritmo, che diamo in forma ricorsiva senza entrare in ulteriori dettagli implementativi, che permette di ordinare una qualsiasi sequenza (non solo sequenze bitoniche):

*Procedura BitonicSort(A: un sequenza di $n=2^k$ elementi)*
*If n=2 then return ($A_{MIN}$ [1], $A_{MAX}$ [1])*
*Else 1. $A_1$= BitonicSort (A[1],…,A[n/2])*
*2. $A_2$= BitonicSort (A[1+n/2],…,A[n])*
*return BitonicMerge($A_1 \bullet A_2^R$)*

La correttezza dell'algoritmo è provata facilmente per induzione, osservando che se $A_1$e $A_2$ sono ordinate, allora è $A_1 \bullet A_2^R$ è bitonica e quindi *BitonicMerge($A_1 \bullet A_2^R$)* è ordinata.

Osserviamo che su PRAM con n processori i passi 1. e 2. possono essere eseguiti in parallelo. Detto $T_{BS}(n)$ il tempo necessario a ordinare una sequenza di n elementi con l'algoritmo BitonicSort, si ha quindi:

$T_{BS}(n)= T_{BS}(n/2)+ T_{BM}(n)$

Ricordando che $T_{BM}(n) = O(\log n)$, si ottiene $T_{BS}(n)= T_{BS}(n/2)+ O(\log n)$. Risolvendo quest'ultima equazione si ha che $T_{BS}(n) = O(\log^2 n)$.

In conclusione, su P-RAM con n processori una sequenza di n elementi può essere ordinata in tempo $O(\log^2 n)$ con l'algoritmo *BitonicSort*.
Le idee e i risultati sulle sequenze bitoniche possono essere utilizzati come base per disegnare una classe di algoritmi di ordinamento (per confronto). Ad esempio, la versione sequenziale dell'algoritmo di *BitonicSort* è meno efficiente di *SortMerge*, poiché lavora in tempo $O(n. \log^2 n)$, mentre il *BitonicSort* può essere efficientemente realizzato su su varie architetture parallele, quali le reti di interconnessione come le mesh e l'ipercubo.

## 4. TECNICA DEL "CICLO EULERIANO"

Presentiamo in questo paragrafo una tecnica di base spesso utile nella progettazione di algoritmi paralleli su P-RAM che lavorano su alberi, perché riduce spesso il problema a un problema che lavora sulla più "trattabile" (dal punto di vista del parallelismo) struttura di lista.

Richiamiamo rapidamente alcune nozioni su grafi. Un ciclo euleriano in un grafo orientato $G$ è un ciclo che attraversa ogni arco una e una sola volta. E' facile dimostrare che condizione necessaria e sufficiente perché $G$ ammetta un ciclo euleriano è che $G$ sia connesso e che in ogni vertice il numero di archi entranti sia uguale al numero di archi uscenti.

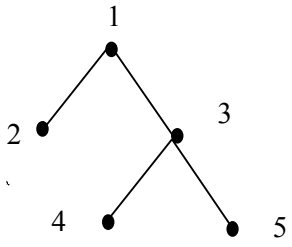
**Esempio 4.1**: Consideriamo un albero $T$, e sia $T_e$ il grafo orientato ottenuto sostituendo ogni lato $\{i, j\}$ di $T$ coi due archi *(i,j)* e *(j,i)* ; poiché in $T_e$ in ogni vertice il numero di archi entranti è uguale al numero di archi uscenti, il grafo $T_e$ ammette un cammino euleriano.

La tecnica del ciclo euleriano consiste nel cercare di risolvere un dato problema su alberi dividendolo nei due passi principali:

(1) Dato un albero $T$, si costruisce in parallelo un cammino euleriano $L$ in $T_e$
(2) Si applica a $L$, visto come lista, un algoritmo parallelo per liste

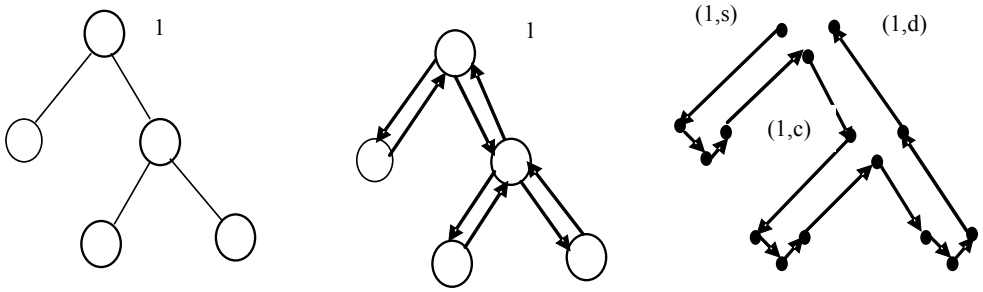
Descriviamo qui un algoritmo parallelo che risolve il problema (1) quando $T$ è un albero binario ordinato con radice.

Supponiamo che l'albero binario con radice $T$ sia descritto dalle tabelle *sin* (figlio sinistro), *des* (figlio destro), *padre* (padre), come nel seguente esempio.




| x | *sin(x)* | *des(x)* | *padre(x )* |
|---|---|---|---|
| 1 | 2 | 3 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 4 | 5 | 1 |
| 4 | 0 | 0 | 3 |
| 5 | 0 | 0 | 3 |

Per ogni vertice *v* dell'albero consideriamo 3 nuovi elementi *(v,s),(v,c),(v,d)*. Un cammino euleriano in $T_e$ può essere descritto dalla lista *L* che ha come indirizzi i nuovi elementi *(v,x)*, e la tabella *succ[(v,x)]* opportunatamente definita, estendendo l'esempio mostrato nella figura seguente:




Albero T Grafo $T_e$ Lista L

La tabella *succ* viene costruita quindi nel seguente modo:

se *v foglia*

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

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

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

se *v nodo interno*

$succ[(v,s)]=(sin[v],s)$

$succ[(v,c)]=(des[v],s)$

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

Si osservi che, fissato *(v,x),* le operazioni per determinare succ[*(v,x)*] sono locali: avendo a disposizione un processore per ogni elemento *(v,x),* la lista L viene costruita in parallelo in tempo O(1).

Abbiamo visto come, dato un albero T, sia possibile con un algoritmo parallelo efficiente costruire una lista L che individua un cammino euleriano. Mostriamo ora due problemi risolubili con questa tecnica:

- Attraversamento di un albero in preordine.
- Profondità dei nodi di un albero binario.

Entrambe i problemi richiedono di associare opportuni interi ad ogni nodo della lista L e risolvere il problema delle SOMME PREFISSE per la lista.

**Esempio 4.2: Attraversamento di un albero in preordine.**

Ricordiamo che, per un albero binario $T$, l'attraversamento in ordine anticipato (preordine) consiste nel visitare prima la radice, nell'attraversare in preordine il sottoalbero di sinistra e poi nell'attraversare in preordine il sottoalbero di destra. La visita in preordine permette in linea di principio di associare ad ogni nodo *v* dell'albero il numero d'ordine *N(v)* di visita del nodo.

Per ottenere un algoritmo parallelo per il calcolo di *N(v)* si consideri il grafo orientato $T_e$ associato a *T* e si pesino con 1 i lati di $T_e$ che "allontanano" dalla radice, con 0 quelli che "avvicinano" alla radice. Fissato un cammino euleriano che parte dalla radice, si consideri un qualsiasi sottocammino che parte dalla radice e arriva a un vertice v: la somma dei pesi degli archi di questo sottocammino è esattamente *N(v)*. Questo dà luogo al seguente algoritmo parallelo di attraversamento:

ALGORITMO ATTRAVERSAMENTO ( un albero binario *T*)

1. *Dato l'albero T, costruisci la lista L che denota un ciclo euleriano.*
2. *Associa 1 a un vertice (v,x) tale che succ[(v,x)]= (v',x') e v=padre(v'); associa 0 a tutti gli altri vertici.*
3. *Applica alla lista L coi pesi così stabiliti l'algoritmo per SOMMEPREFISSE.*

La correttezza è provata osservando che la somma prefissa fino al vertice (v,s) è il numero d'ordine N(v) del vertice v in una visita di T in preordine.

Poiché il calcolo di L e la definizione dei pesi dei suoi elementi richiede tempo parallelo O(1) e l'esecuzione dell'algoritmo per le somme prefisse richiede tempo parallelo $O(\log_2 n)$, concludiamo che l'attraversamento di un albero in preordine può messere fatto con un algoritmo parallelo in tempo $O(\log_2 n)$.

**Esempio 4.3: Profondità dei nodi di un albero binario**.

Ricordiamo che la profondità di un nodo *v* in un albero binario T è la lunghezza del cammino che va dalla radice a *v*.

Per ottenere un algoritmo parallelo per il calcolo della profondità, si consideri il grafo orientato $T_e$ e si pesano con *+1* il lati di $T_e$ che "avvicinano" alla radice, con *–1* i lati di $T_e$ che "allontanano" dalla radice: si osservi che in tal modo la somma dei pesi di lati che formano un cammino chiuso è 0. La profondità dei vari vertici può essere ottenuto semplicemente costruendo in parallelo in tempo O(1) la lista L associata al cammino euleriano di $T_e$ , calcolando poi in tempo $O(\log_2 n)$ le somme prefisse.

## 5. RETI DI INTERCONNESSIONE

Nel modello P-RAM precedentemente analizzato ogni processore può scambiare informazione con ogni altro in tempo O(1) grazie alla condivisione della memoria. Questa velocità di comunicazione si riflette nella velocità con cui molti problemi possono essere risolti su tale modello. D'altro lato, è importante ricordare che il modello P-RAM non è fisicamente realizzabile, con le attuali tecnologie, se non quando il numero di processori è modesto.

Modelli alternativi sono le macchine a memoria distribuita. In questo caso, ogni processore può accedere soltanto ad una sua memoria locale e i processori possono scambiarsi informazioni solo attraverso una opportuna rete di interconnessione; supporremo qui che la sincronizzazione avvenga grazie ad un orologio globale.

Una rete di interconnessione può essere astrattamente descritta da un grafo, i cui nodi sono i processori e i lati sono coppie di processori che possono comunicare direttamente. Per semplicità, supponiamo qui che la comunicazione sia bidirezionale: se A comunica direttamente con B allora anche B comunica direttamente con A. Il grafo che descrive l'interconnessione è allora un grafo non orientato G = (V,E).

Importanti parametri di una rete di interconnessione sono il suo *grado*, il suo *diametro* e la sua *ampiezza di bisezione*.

- *Grado della rete* G = (V,E).
  Il grado di un vertice $i \in V$ è il numero d(i) di lati uscenti da i. Il grado d(G) della rete è il massimo dei gradi dei vertici:
  $$d(G) = Max_i\, d(i)$$
  Reti di grado elevato sono di difficile costruzione: un basso grado risulta allora una caratteristica desiderabile.

- *Diametro di comunicazione* della rete G = (V,E). La distanza tra due vertici $i \in V$ e $j \in V$ è la lunghezza D(i,j) del più breve cammino da i a j. Il diametro di comunicazione D(G) è la massima distanza tra vertici:
  $$D(G) = Max_{i,j}\, D(i,j)$$
  Un basso diametro è una caratteristica desiderabile, poiché permette un trasferimento veloce di informazione tra una qualsiasi coppia di processori.

- *.Ampiezza di bisezione* della rete G = (V,E).La ampiezza di bisezione B(G) del grafo G è il minimo numero di lati che si devono togliere in modo che il nuovo grafo ottenuto sia formato da due componenti disconnesse contenenti (approssimativamente) lo stesso numero di vertici. La ampiezza di bisezione è un limite inferiore alla quantità di informazione che metà dei processori possono scambiare con i rimanenti in un passo di calcolo. Reti con alta

ampiezza di bisezione permettono di smistare tra i processori in poco tempo grandi quantità di dati; per contro, tali reti risultano di difficile realizzazione.

Un ulteriore parametro che permette di valutare le caratteristiche di una rete è il tempo di esecuzione del miglior algoritmo per la soluzione di un dato problema. In questa breve rassegna utilizzeremo due problemi di riferimento per il confronto di reti:

- Determinazione del MASSIMO in un vettore
- ORDINAMENTO.

Questi problemi possiedono infatti caratteristiche diverse. La capacità di risolvere efficientemente la determinazione del MASSIMO, simile al problema SOMMATORIA, risiede infatti nella semplice possibilità di far comunicare rapidamente una qualsiasi coppia di processori. Una soluzione parallela efficiente al problema ORDINAMENTO richiede invece di poter trasportare velocemente grandi masse di dati in un singolo passo. Queste caratteristiche portano alle seguenti stime di limiti inferiori al tempo necessario a risolvere MASSIMO e ORDINAMENTO con algoritmi paralleli implementati su reti di interconnessione.

**Fatto 5.1:** per risolvere il problema MASSIMO di n elementi con una rete di diametro D occorre almeno un tempo D.

Se infatti due processori hanno distanza maggiore di D, sarà impossibile confrontare tra loro gli elementi contenuti in questi processori in D passi.

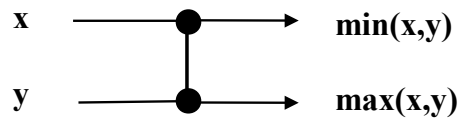
**Fatto 5.2:** per risolvere il problema ORDINAMENTO di n elementi con una rete di ampiezza di bisezione B occorre almeno un tempo n/2B.

Supponiamo infatti che ogni processore contenga all'inizio un elemento da ordinare e che alla fine il processore k debba contenere il k° elemento in ordine crescente. Se i primi n/2 processori contengono gli n/2 elementi più grandi, alla fine del processo essi devono contenere gli n/2 elementi più piccoli. Ad ogni passo, possiamo tuttavia trasferire in questi processori al più B elementi dagli altri processori, quindi i passi richiesti sono almeno n/2B.

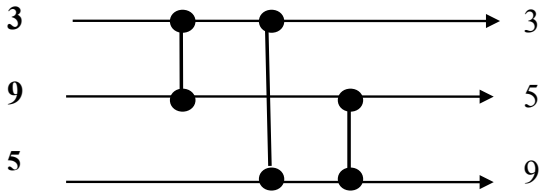
Prima di descrivere algoritmi per l'ordinamento su diverse reti di interconnessione, presentiamo un risultato sulle *reti di ordinamento* (sorting network), che permette di semplificare grandemente le prove di correttezza degli algoritmi.

Molti algoritmi di ordinamento basati su confronti sono "oblivious", cioè le sequenze di confronti per qualsiasi ingresso di data dimensione. In questo caso è semplice descrivere gli algoritmi come circuiti, la cui componente elementare è l'elemento *confronto-scambio*, che riceve in ingresso la coppia di interi (x,y) e dà in uscita la

coppia, ordinata in modo crescente, ( min(x,y), max (x,y) ):




Una *rete di ordinamento* di n interi può essere graficamente rappresentata da n linee orizzontali, aventi in ingresso gli n interi; ogni operazione di confronto-scambio è rappresentata da una connessione verticale tra due di queste linee. Una rete per l'ordinamento di 3 interi è rappresentata nella figura seguente:




In generale una rete R, avendo in ingresso un vettore di interi ($x_1$, …, $x_n$), dà in uscita un vettore di interi ($y_1$, …, $y_n$) che denoteremo R($x_1$, …, $x_n$). La rete R è *corretta* se, avendo in ingresso un qualsiasi vettore di interi ($x_1$, …, $x_n$), dà in uscita il vettore ordinato ($y_1$, …, $y_n$), tale cioè che $y_1 \leq y_2 \leq \ldots \leq y_n$ .

Uno strumento che grandemente semplifica le dimostrazioni di correttezza delle procedure di ordinamento su reti è un risultato detto "*principio 0,1*"(Knuth 1972):

**Fatto 5.3** ***(principio 0,1)*****:** se una rete di ordinamento R lavora correttamente su vettori con componenti in {0,1}, allora lavora correttamente per qualsiasi vettore di interi.

Per provare questo principio, consideriamo una qualsiasi funzione f:N→N monotona, cioè tale che, se x≤y, allora f(x)≤ f(y). Chiaramente f(max(x,y))=max(f(x),f(y)) e f(min(x,y))=min(f(x),f(y)); ne segue che, se ($y_1$, …, $y_n$)=R($x_1$, …, $x_n$), allora (f($y_1$), …,f($y_n$))=R(f($x_1$), …, f($x_n$)).

Supponiamo ora che la rete R non sia corretta, cioè che esista un vettore di interi ($x_1$, …,$x_n$) che tale che, dando in ingresso tale vettore alla rete R, R dia in uscita un vettore ($y_1$,…, $y_n$) non ordinato; si trovano allora due indici s,k per cui $y_s > y_k$ pur essendo s<k.

Consideriamo ora la funzione g:N→{0,1}, definita da:

g(x) = se x ≥ $y_s$ allora 1, altrimenti 0

Si osservi che g($y_s$)=1 mentre g($y_k$)=0.

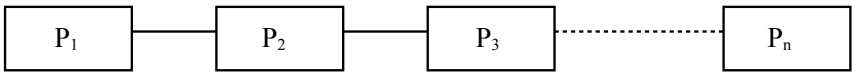
Poiché g è monotona, risulta (g($y_1$),…,g($y_n$))=R(g($x_1$), …, g($x_n$)): dando in ingresso a R il vettore (g($x_1$), …, g($x_n$)) a componenti 0,1 si ottiene il vettore (g($y_1$),…,g($y_n$)) che non è ordinato, poiché g($y_s$)=1 mentre g($y_k$)=0, con s<k.

Sulla base dei parametri che abbiamo introdotto, analizziamo e confrontiamo di seguito alcune topologie di reti di interconnessione:

- *Array lineare*
- *Mesh quadrata (o Array bi-dimensionale)*
- *Albero completo*
- *Ipercubo*

### *Array lineare*

Un array lineare di dimensione n consiste di n processori connessi in modo sequenziale, cioe` il processore $P_i$ e` collegato a $P_{i-1}$ e $P_{i+1}$, tranne i processori $P_1$ e $P_N$ che sono, rispettivamente, collegati soltanto a $P_2$ e $P_{N-1}$.




Array Lineare di dimensione n.

Il grado di un array lineare e` 2, poiché ogni processore è collegato al più ad altri due.

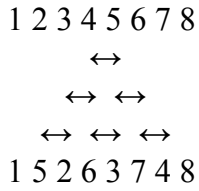
Il diametro di un array lineare è n-1, che rappresenta la distanza tra i processori più ”lontani” $P_1$ ed $P_n$ . Eseguire un algoritmo che richiede di processare informazioni inizialmente memorizzate in una qualsiasi coppia di processori richiede allora tempo $\Omega(n)$: in un array lineare risulta allora $\theta(n)$ il tempo necessario per determinare il massimo degli elementi di un vettore (A[1],…,A[n]), quando l’elemento A[i] è originariamente memorizzato nel processore $P_i$.

Se si elimina da un array lineare il lato che collega il processore $P_{n/2}$ al processore $P_{1+n/2}$, si ottengono due grafi disconnessi con lo stesso numero di vertici: l’ampiezza di bisezione di un array lineare è dunque 1: in un array lineare non è in generale possibile, in modo veloce, smistare agli altri processori informazioni originariamente memorizzate nei primi n/2 processori.

Prendiamo ora in considerazione due algoritmi paralleli realizzati su array lineare. Il primo, avendo in ingresso un vettore, ne calcola lo SHUFFLE, il secondo ordina il vettore stesso.

**Esempio 5.1: algoritmo per il calcolo dello SHUFFLE**

L'operazione SHUFFLE associa al vettore *(a[1], a[2], a[3], ..., a[2s]),* il vettore *(a[1], a[s+1], a[2], a[s+2], ...,a[s], a[2s]),* ottenuto alternando le componenti dei vettori *(a[1], a[2], ..., a[s])* e *(a[s+1], a[s+2], ..., a[2s]).*
Su un array lineare di n elementi, lo SHUFFLE di *(a[1], a[2], a[3], ..., a[2s])* può essere ottenuto in tempo O(s) da un procedura *SHUFFLE* con scambi a "triangolo" illustrata dalla seguente figura:




**Esempio 5.2: algoritmo *P-D-ORD* su array lineare**

Proponiamo ora un semplice algoritmo di ordinamento per confronti, che permette di ordinare con un array lineare un vettore di n elementi in tempo O(n): poiché per le considerazioni fatte sopra ogni algoritmo di ordinamento richiede tempo Ω(n), tale algoritmo è asintoticamente ottimo. Osserviamo che questo algoritmo è più veloce, sia pur di poco, di qualsiasi algoritmo per macchine a 1 processore, visto il tempo di un qualsiasi algoritmo sequenziale di ordinamento è Ω(n.log n).

Consideriamo il problema di ordinare n elementi *a[1],..., a[n]* utilizzando n processori $P_1$,..,$P_n$ con rete di interconnessione ad array lineare. Per ogni i, il processore $P_i$ ha una variabile locale *A[i]* che prima dell'esecuzione dell'algoritmo contiene l'elemento *a[i]*; al termine dell'esecuzione le variabili *A[i]* , .., *A[n]* dovranno contenere gli elementi in ordine crescente.

Presentiamo ora l'algoritmo *P-D-ORD* . L'idea su cui si basa l'algoritmo è quella di dividere i passi di calcolo in pari e dispari; in un passo pari, gli elementi memorizzati nei processori di posto pari sono confrontati con quelli nel loro vicino sinistro ed eventualmente scambiati, e similmente nei passi dispari.

Una semplice sottoprocedura che utilizzeremo è *SCAMBIO(i,i+1)* , il cui risultato è lo scambio dei dati presenti in *A[i]* e *A[i+1]* tra i processori $P_i$ e $P_{i+1}$ .
Lo schema ad alto livello dell'algoritmo è il seguente:

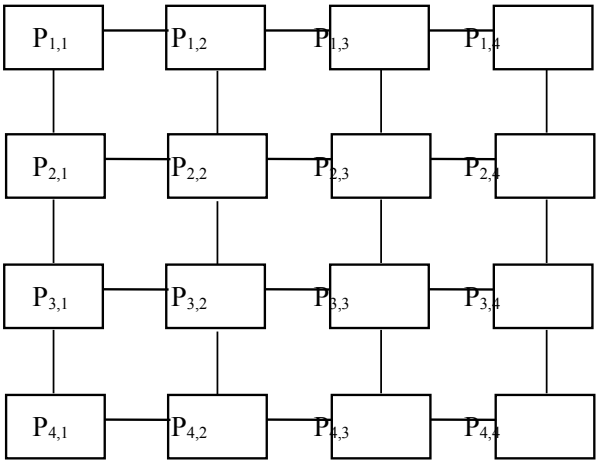
*ALGORITMO P-D ORD (a[k] memorizzato in A[k] (1≤k≤n ))*
*For 1≤t≤ n do*
*If t pari then for 1≤k≤n/2 do in parallel*
*If A[2k-1]<A[2k] then SCAMBIA(2k-1,2k)*
*If t dispari then for 1≤k<n/2 do in parallel*
*If A[2k]<A[2k+1] then SCAMBIA(2k,2k+1)*

L'idea della procedura è di fare in modo che, nei passi pari, i processori $P_1$ e $P_2$, $P_3$ e $P_4$, e così via, si scambino i dati presenti in *A[2i-1]* e *A[2i]* solo se *A[2i-1] <A[2i]*; nei passi dispari i processori $P_2$ e $P_3$, $P_4$ e $P_5$, e così via, fanno la stessa cosa. L'algoritmo è facilmente implementabile, usando un'altra variabile locale in ogni processore, perché i test sono effettuati su variabili di processori adiacenti.

Si osservi che il dato minore, ad ogni passo, si "muove" da un processore al precedente finchè non arriva alla posizione 1; il dato penultimo in ordine fa la stessa cosa, salvo in un eventuale passo in cui incontra il dato minore, e così via. Questo significa che, dopo al più n passi, tutti i dati si troveranno nella corretta posizione: il k° dato dal basso si troverà nel processore $P_k$ .

***Mesh quadrata***

E' la versione bidimensionale dell'array lineare. Consiste in $n = m^2$ processori inseriti in un griglia m× m, tale che il processore $P_{i,j}$ e` connesso ai processori $P_{i\pm 1,j}$ e $P_{i,j\pm 1}$ (salvo ai confini).



mesh 4x4

Il grado della mesh quadrata è dunque 4, che è il grado dei vertici interni.

I processori più "lontani" sono $P_{11}$ e $P_{mm}$:essi sono collegati da un cammino di 2(m-1) passi, quindi il diametro di una mesh è $2\cdot(\sqrt{n}-1)$.

Ogni algoritmo che richiede di processare informazioni originariamente memorizzate in una qualsiasi coppia di processori richiede allora un tempo di esecuzione $\Omega(\sqrt{n})$: si osservi che questo limite è inferiore a quello ottenuto per l'array lineare ( $\Omega(n)$ ), ma decisamente superiore a prestazioni ottenute per vari problemi su PRAM ( O(log n) ).

Per dividere una mesh quadrata in due rettangoli disconnessi di ugual numero di vertici basta "tagliare" le connessioni tra i processori $P_{m/2\ k-1}$ e $P_{m/2\ k}$ (k=1,m/2): l'ampiezza di bisezione di una mesh mxm è allora $\sqrt{n}$, e il tempo di trasferimento di dati memorizzati in un rettangolo nell'altro non potrà essere inferiore a $\sqrt{n}/2$. Questo implica, in particolare, che ogni algoritmo di ordinamento su mesh richiede almeno $2\sqrt{n}$ passi di calcolo.

**Esempio 5.3: Determinazione del MASSIMO su mesh**

Consideriamo il problema MASSIMO su una mesh mxm. Supponiamo che ogni processore $P_{ij}$ abbia una variabile locale *A[i,j]* che inizialmente contiene la componente *a[i,j]* della matrice a= (*a[i,j]* ) di cui vogliamo calcolare il massimo. Al termine dell'esecuzione della procedura, si vuole che $M=\max_{ij} a[i,j]$ sia memorizzato nel processore $P_{11}$.

Per disegnare la procedura basta osservare che una mesh quadrata mxm può essere vista, dimenticando le connessioni verticali, come una collezione di m array lineari di m processori (righe). Analogamente, dimenticando le connessioni orizzontali, può essere vista come una collezione di m array lineari di m processori (colonne).

La procedura, ad alto livello, è la seguente:

*ALGORITMO "MAX Mesh"*

- *Trova il massimo di ogni riga i, mettendo il risultato in A[i1].*
- *Trova il massimo sulla prima colonna, mettendo il risultato in A[11].*

Entrambe i passi, agendo in parallelo sulle righe (colonne) viste come array lineari, richiedono tempo $O(\sqrt{n})$, che risulta asintoticamente ottimo.

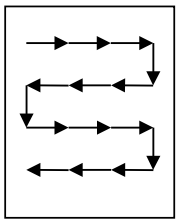
**Esempio 5.4: algoritmo *SortMergeLS3* su mesh**

Presentiamo qui un algoritmo che risolve su mesh il problema ORDINAMENTO in tempo ottimo $O(\sqrt{n})$; per semplicità, prenderemo in considerazione mesh mxm dove m è potenza di 2.

Supponiamo per prima cosa che il vettore da ordinare *(a[1],.., a[n]),* con $n=m^2$, si trovi memorizzato "a serpente" nella mesh mxm. Questo significa:

- se i è pari, nella variabile *A[ik],* locale al processore $P_{ik}$ , si trova memorizzato il numero *a[(i-1).m+k)]*.
- se i è dispari, nella variabile *A[ik],* locale al processore $P_{ik}$ , si trova memorizzato il numero *a[(i-1).m+m-k+1]*.

Qui sotto viene mostrata la memorizzazione a "serpente" di un vettore di 16 componenti in una mesh

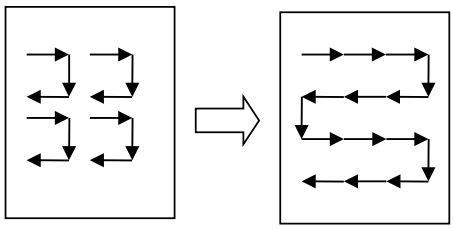


Vettore " a serpente" in mesh

Obbiettivo di un algoritmo di ordinamento su mesh è di fare in modo che, dopo l'esecuzione della procedura, il vettore si trovi memorizzato a "serpente" ordinato in modo crescente. Si osservi che i processori ordinati "a serpente" formano un array lineare, è quindi possibile applicare l'algoritmo *P-D-ORD* precedentemente visto; tale algoritmo tuttavia ha un tempo di calcolo O(n), non sfruttando altre potenziali connessioni tra i processori.

Discutiamo ora l'algoritmo LS3, dovuto a Lang, Schimmler, Schmeck e Schroeder (1985), che si basa sulla tecnica "divide et impera" e risolve (come nel caso del Bitonic Sort su P-RAM) il problema richiamando una procedura di Merge.

Ogni mesh mxm può essere suddivisa in 4 mesh m/2 x m/2 . La procedura di *MergeLS3* ordina a serpente, sulla mesh mxm, un vettore che in partenza è ordinato, a serpente, in ognuna delle 4 sottomesh, come rappresentato nella figura seguente (relativa al caso m=4):

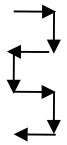


Effetto della procedura di Merge

Questo algoritmo richiama le procedure di *SHUFFLE* e *P-D-ORD,* descritte nella sezione dedicata all'array lineare, applicate ad array lineari che sono opportuni sotto-array della mesh.

Per prima cosa, ricordiamo che ogni riga della mesh mxm può essere vista come un array lineare di m processori. Chiameremo *SHUFFLE (i)* l'applicazione della procedura *SHUFFLE* alla riga i della mesh.

Osserviamo poi che due colonne consecutive di processori di una mesh possono essere visti come un array lineare se si ordinano i processori "a serpente", come illustrato della figura seguente



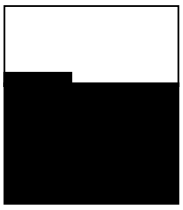
Chiameremo *DOPPIA-COLONNA(i,i+1)* la procedura che si ottiene applicando l'algoritmo *P-D-ORD* ai processori delle colonne i e i+1 della mesh, che formano un array lineare nell'ordinamento "a serpente".

La procedura *MergeLS3* può essere così descritta:

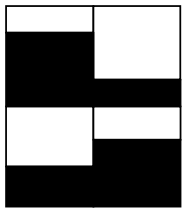
*Procedura MergeLS3 (m x m mesh in cui le 4 m/2 x m/2 sottomesh sono già ordinate a serpente)*

1. *Applica SHUFFLE(i) in parallelo per 1≤i≤m (sulle righe)*
2. *Applica DOPPIA-COLONNA(2i-1,2i) in parallelo per 1≤i≤m/2*
3. *Applica i primi 2m passi di P-D-ORD all'array lineare formato dal serpente della mesh.*

Per provare, sia pur a grandi linee, la correttezza della procedura *MergeLS3*, ricorriamo al "*principio 0,1*" . Prendiamo in considerazione quindi vettori memorizzati in mesh a componenti 0 o 1; un vettore ordinato "a serpente" darà luogo ad una mesh con righe superiori a componenti 0, righe inferiori a componenti 1, salvo eventualmente una riga di confine che può contenere sia 0 che 1. La seguente figura rappresenta a sinistra un vettore ordinato in mesh mxm, a destra un vettore che risulta ordinato in ognuna delle quattro sottomesh m/2 x m/2 (le componenti 0 sono rappresentate in bianco, le componenti 1 in nero, per semplicità non è rappresentata la riga di confine)

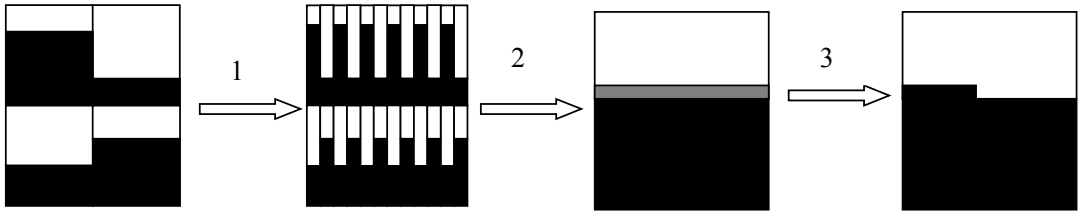


Vettore ordinato



Vettore ordinato nelle sottomesh

La procedura *MergeLS3* lavora nei tre passi come segue:




Il primo passo (*SHUFFLE* per riga) fa in modo che le coppie di colonne adiacenti siano uguali (a parte le righe di confine). Il secondo passo (*DOPPIA COLONNA*) ricostruisce il vettore ordinato, a parte la riga di confine. Bastano infine 2m passi per ordinare la riga di confine.

Ognuno dei tre passi della procedura *MergeLS3* lavora in tempo O(m), quindi l'intera procedura ha un tempo di calcolo al più $c\sqrt{n}$, per una opportuna costante c.

Possiamo ora descrivere la procedura *MergeSortLS3* che, nei suoi passi essenziali, risulta essere:

*Procedura MergSortLS3 (m; vettore (a[1],..,a[n]) memorizzato a serpente)*
*Se m=1 allora return a*
*altrimenti*

- *dividi la mesh mxm in 4 sottomesh m/2 x m/2*
- *applica ricorsivamente in parallelo la procedura di MergeSortLS3 a ognuna delle sottomesh m/2 x m/2.*
- *applica la procedura di MergeLS3 alla mesh mxm con la configurazione ottenuta.*

La correttezza è immediata poiché in ingresso alla procedura *MergeLS3* risulta un vettore ordinato nelle quattro sottomesh m/2 x m/2.

Analizziamo ora il tempo di calcolo. Detto $T_{MS}(n)$ il tempo per ordinare un vettore di n elementi con la procedura *MergeSortLS3* e $T_M(n)= c\sqrt{n}$ il tempo necessario a ordinare con la procedura *MergeLS3* un vettore parzialmente ordinato di n elementi, vale:

$$T_{MS}(n) \leq T_{MS}(n/4)+ T_M(n) = c\ \sqrt{n}\ + T_{MS}(n/4)$$

Pertanto:

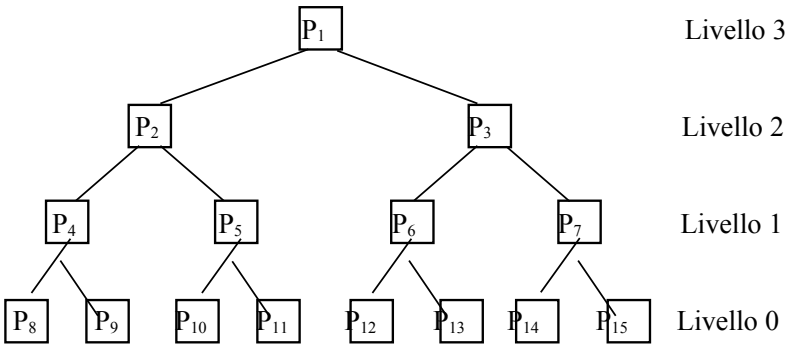
$$T_{MS}(n) \leq c\ \sqrt{n} + c\sqrt{n/4} + c\sqrt{n/16} + \ldots \leq 2c\ \sqrt{n}$$

Concludiamo che la procedura *MergSortLS3* ordina un vettore di n elementi in tempo asintoticamente ottimale $O(\sqrt{n})$.

***Albero di base n***

Un albero di base n è un albero binario completo con n foglie; i processori sono posti ai nodi dell'albero. L'albero e` composto da *d* livelli, numerati da 0 a d-1, quindi $n=2^{d-1}$ e il numero di processori è 2n-1.

Albero di base 8




Ogni processore $P_k$ , che non sia radice o foglia, e` connesso sia al processore padre (al livello d+1) che ai due figli (al livello d-1), mentre la radice non ha padre e le foglie non hanno figli: l'albero di base n ha quindi grado 3.

Osserviamo ora che il cammino da una foglia alla radice richiede d-1 passi: il diametro e` allora pari a 2(d-1) = θ(log n). Questo basso diametro permette di risolvere in maniera efficiente il problema Massimo

**Esempio 5.5: determinazione del MASSIMO**

Il seguente algoritmo risolve in tempo asintoticamente ottimo O(log n) il problema di determinare il massimo elemento in un vettore $(A_1,\ldots, A_n)$, le cui componenti sono inizialmente memorizzate nelle foglie dell'albero di base n.

*For all k (k=1,n) do in parallel*
*"il processore $P_k$ posto alla foglia k invia $A_k$ al proprio padre"*
*For t=2,log n –1 do*
*For all "nodo interno v" do in parallel*
*If " $P_v$ posto al nodo v riceve a dal figlio sinistro e b dal figlio destro" then*
*$c_v$ = Max{a,b}*
*$P_v$ invia $c_v$ al proprio padre*
*If " $P_r$ posto alla radice r riceve a dal figlio sinistro e b dal figlio destro" then*
*$c_r$ = Max{a,b}*

L'esecuzione termina dopo O(log n) passi di calcolo e al termine dell'esecuzione il valore del massimo è memorizzato nella radice.

Analizziamo ora l'ampiezza di bisezione dell'albero di base n. Tagliando le due connessioni tra la radice e i suoi figli, si ottengono due alberi disconnessi di base n/2: l'ampiezza di bisezione dell'albero è dunque 2.

Questa bassa ampiezza di bisezione è causa di colli di bottiglia nell'esecuzione di algoritmi che richiedono di trasferire grandi quantità di dati. Per esempio, in un qualsiasi algoritmo di ordinamento si devono poter trasferire gli n/2 dati memorizzati nel sottoalbero di sinistra nel sottoalbero di destra: questi dati devono passare per la radice e il processore $P_r$ può trasferire al più un dato alla volta. Concludiamo col seguente risultato negativo:

**Fatto 5.4:** ogni algoritmo di ordinamento su un albero di base n richiede tempo $\Omega(n)$.

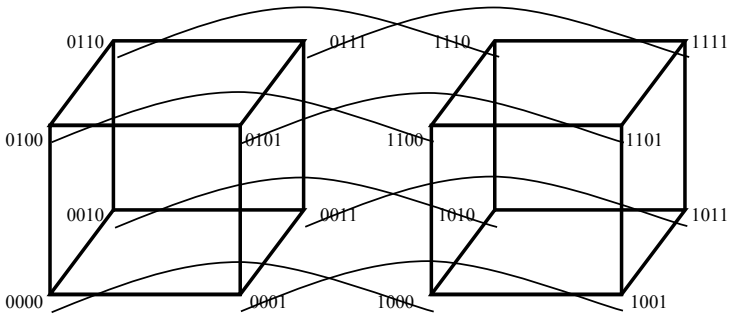
In conclusione, il basso diametro dell'albero permette la soluzione efficiente di alcuni problemi che richiedono di combinare informazioni distribuite nell'albero, a patto che non vengano smistate da una parte all'altra grandi quantità di dati: in tal caso la bassa ampiezza di bisezione è responsabile di prestazioni povere.

### *Ipercubo*

Per $n = 2^d$, un ipercubo (o d-cubo) è un grafo i cui vertici sono elementi $c_1 \ldots c_d \in \{0,1\}^d$ e due vertici x e y sono estremi di un lato se le parole x e y differiscono in una sola posizione: in una architettura a ipercubo i processori $P_x$ e $P_y$ sono quindi collegati se x e y differiscono in una sola posizione. Per esempio, il processore $P_{1001}$ e` collegato a $P_{1000}$ ma non a $P_{1111}$

L'ipercubo e` una struttura ricorsiva: un d+1-cubo può essere ottenuto connettendo due d-cubi. Uno dei due cubi possiede i processori i cui indici hanno il bit piu` significativo pari a 0, mentre l'altro cubo quelli i cui indici hanno il bit piu` significativo uguale ad a 1.

Un 4-cubo




Ogni vertice $c = c_1 \ldots c_d$ è collegato direttamente ai d vertici ottenuti modificando un bit di c: il grado di un d-cubo è dunque $d = \log n$.

I vettori più distanti di un cubo sono 00..0 e 11..1: essi sono collegati da un cammino di lunghezza d, quindi il diametro di un d-cubo è $d = \log n$. Il basso diametro di un d-cubo permette di risolvere in modo efficiente problemi che richiedono di combinare informazioni distribuite. In particolare, non è difficile progettare algoritmi in tempo $O(\log n)$ per la soluzione del problema MASSIMO. Presentiamo qui un semplice algoritmo per tale compito.

**Esempio 5.6: Determinazione del MASSIMO.**

Dato il vettore $(A_1,\ldots, A_n)$, supporremo che la componente $A_k$ sia memorizzata, inizialmente, nella memoria locale A[k] del processore $P_k$ (k=1,n) (qui con abuso di notazione identifichiamo l'intero k con la sua rappresentazione binaria); il risultato della computazione sarà il massimo $\text{Max}_k A_k$ memorizzato in A[0] nel processore $P_0$ (si noti che gli indici dell'array iniziano da 0). Nell'algoritmo che segue, $i^{(b)}$ è ottenuto complementando il *b-esimo* bit dell'indice *i* in notazione binaria; ad esempio, $1000110^{(3)}$ è 1010110. Si osservi in particolare che il processore $P_i$ e $P_i^{(b)}$ sono collegati.

*ALGORITMO ( Massimo su un ipercubo)*
*Input: Un array $(A_1,\ldots, A_n)$ dove A(i) e` memorizzato nella memoria locale A[i] del processore $P_i$ (i=1,n).*

*For b= d-1, 0 do in parallel*
*if $(0 \le i \le 2^b-1)$*
- *$P_i^{(b)}$ invia a $P_i$ il contenuto di $A[i^{(b)}]$*
- *$A[i] = Max \{A[i], A[i^{(b)}]$*

La correttezza si prova osservando che, posto $M= \text{Max}_k A_k$, quando b assume valore s, con $0 \le s<d$, il massimo dei valori contenuti in A[i] con $0 \le i \le 2^b-1$ è M. In particolare, all'ultimo passo vale b=0, ne segue che $0 \le i \le 2^b-1$ implica i=0 e quindi il processore $P_0$ contiene M in A[0].

Per disconnettere i due d-1-cubi con vertici 0x e 1y è necessario tagliare n/2 lati di un d-cubo. Più in generale, per disconnettere due sottonisiemi disgiunti A e B del d-cubo, ognuno di cardinalità n/2, è necessario in generale "tagliare" almeno n/2 lati: l'ampiezza di bisezione di un d-cubo risulta allora n/2. Se l'alta ampiezza di bisezione rende difficile la realizzazione h/w di una rete a ipercubo, la stessa permette di smistare in parallelo grandi quantità di dati. Ad esempio, contrariamente a quanto visto per l'albero di base n, sull'ipercubo il problema dell'ordinamento può essere risolto efficientemente in tempo $O(\log^2 n)$ implementando opportunamente l'algoritmo bitonico, presentato per le P-RAM.