Università degli studi di Roma "La Sapienza"
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

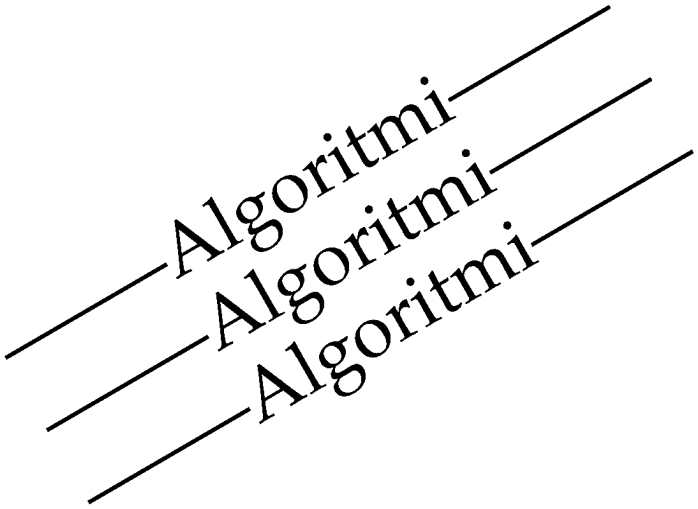
Appunti del corso di Elaborazione dell'Informazione Non Numerica

Professoressa Rossella Petreschi

anno accademico 1993/1994

versione 1.00

A cura di Fabio Lombardino, Massimiliano Tarquini e Valeria Vicinanza



NOTA PER IL LETTORE

Si fa presente che le seguenti dispense si riferiscono ad appunti presi durante le lezioni e sono tuttora in fase di revisione; è, quindi, lasciata al lettore la responsabilità di verificare la correttezza di quanto esposto e di porsi in atteggiamento critico nei confronti di tali appunti.

1 Parallelismo

1.1 Introduzione

La difficoltà nel rispondere alla richiesta di maggiore potenza computazionale con il passaggio a nuove tecnologie sembra essere il motivo principale che ha portato al progetto ed allo sviluppo di architetture parallele. Infatti, le tecnologie attuali hanno sostanzialmente permesso di raggiungere i limiti fisici dovuti alla velocità degli elettroni all'interno dei circuiti (la quale tende alla velocità della luce); pertanto ulteriori incrementi delle prestazioni di calcolo si potranno cercare, con maggior profitto, in campi diversi da quello del perfezionamento delle tecnologie attuali. La strada della connessione parallela tra più processori sembra essere la più praticabile nel prossimo futuro, anche perché il drastico abbassamento dei costi dell'elettronica ha consentito un buon risparmio economico rispetto all'utilizzo di un solo processore di pari potenza complessiva. Sebbene gli attuali monoprocessori raggiungano potenze di calcolo molto significative, circa 10^9 operazioni al secondo, diamo di seguito un elenco di applicazioni che richiedono potenze di calcolo molto maggiori.

Nel campo della metereologia la simulazione di temporali e correnti marine tridimensionali richiede calcoli non lineari che, con le attuali architetture, limitano la portata delle previsioni del tempo a 5 giorni. Lo sviluppo di sistemi di calcolo parallelo, che consentono di acquisire dati con una maggiore risoluzione utilizzando anche i satelliti geostazionari, potrebbe permettere previsioni della portata di 8 o 10 giorni.

Nel campo dell'ingegneria, la simulazione del comportamento dei corpi immersi nei fluidi richiede attualmente notevole tempo di calcolo. Si ipotizza che con la prossima generazione di computers paralleli saremo in grado di sviluppare le teorie sulla simulazione del comportamento aerodinamico degli oggetti fino a poter progettare un aeroplano in grado di volare dagli Stati Uniti al Giappone in meno di un'ora.

La simulazione al computer si estende anche al mondo dei materiali innovativi con proprietà decise in base alle nostre esigenze; oppure al comportamento dei mercati finanziari per il quale non esistono teorie affermate e le previsioni vengono basate esclusivamente sullo studio di ingenti moli di dati sui quali si effettuano calcoli stocastici non lineari.

Elaborazioni complesse di grandi basi di dati sono frequenti nelle applicazioni di intelligenza artificiale, quali il riconoscimento degli oggetti, il disegno di circuiti logici o la dimostrazione di teoremi matematici o infine la scelta di strategie militari.

1.2 Definizione di macchina parallela

Una definizione astratta di macchina parallela può essere data come segue:

una macchina parallela è un insieme di processori, tipicamente dello stesso tipo, interconnessi in modo tale da consentire il coordinamento delle attività e lo scambio dei dati.

1.3 Modelli di macchine parallele

Le macchine parallele possono essere classificate in:

- SISD: Single Instruction, Single Data; è la tradizionale macchina sequenziale di Von Neumann in cui, ad ogni passo, l'unica unità di controllo esegue una sola istruzione che opera su di un singolo dato.

- **MISD**: Multiple Instructions, Single Data; (si tratta di un'estensione della macchina sequenziale SISD) in cui ogni processore esegue un'operazione differente da quella svolta dagli altri su di un singolo dato comune a tutti i processori. Questo tipo di macchine viene spesso utilizzato per scopi specifici come l'elaborazione di segnali in cui un unico flusso di dati passa per diversi processori che effettuano ciascuno una trasformazione degli stessi, per la quale il processore è appositamente progettato. Ad esempio nel riconoscimento di diverse categorie di oggetti, possiamo utilizzare più processori ciascuno dei quali esegue il programma per analizzare tutti gli oggetti e riconoscere quelli appartenenti alla sua categoria; un oggetto è riconosciuto da un processore se appartiene alla categoria assegnata al processore stesso.

- **SIMD**: Single Instruction, Multiple Data; macchina in cui ogni processore esegue la stessa istruzione su dati differenti; essa costituisce il modello base per la **PRAM** (Parallel Random Access Machine). Questo modello può essere ulteriormente suddiviso a seconda del modo in cui i processori comunicano tra di loro:

1) Shared memory, in cui la comunicazione tra processori avviene passando attraverso zone di memoria condivisa. Ulteriori distinzioni sono necessarie a seconda delle differenti modalità di accesso consentite:

- EREW: Exclusive Read, Exclusive Write;
- CREW: Concurrent Read, Exclusive Write;
- ERCW: Exclusive Read, Concurrent Write;
- CRCW: Concurrent Read, Concurrent Write;

Nei modelli a scrittura concorrente, qualora più processori accedano in scrittura ad una stessa locazione di memoria condivisa, è possibile adottare uno dei seguenti criteri di accesso:

si permette la scrittura ad uno qualunque dei processori se i dati da inserire sono uguali fra loro;

si scrive nella cella di memoria una funzione di tutti i dati (ad esempio la somma);

si può stabilire una scala di priorità tra i processori.

Ecco un esempio di algoritmo di lettura su macchina a lettura concorrente:

```
for i:=1 to n in parallel do
      Pi:  carica  il dato D nella sua memoria locale
```

2) Reti di interconnessione, in cui le informazioni passano attraverso opportuni canali direttamente dalla memoria locale di un processore a quella di uno o più processori adiacenti.

La comunicazione tra processori è necessaria in quanto occorre che ciascun processore possa conoscere i dati parziali per coordinare il flusso di istruzioni

fino al risultato globale, dato che ogni processore elabora dati differenti da quelli utilizzati dagli altri.

-MIMD: Multiple Instructions, Multiple Data; ogni processore ha un suo insieme di dati ed un suo insieme di istruzioni per elaborarli; i processori sono collegati o attraverso memoria condivisa (nei calcolatori multiprocessore) o attraverso reti di interconnessione (nei multicalcolatori). Talvolta parlando di multicalcolatori si fa riferimento a sistemi distribuiti.

***Nota**: nel seguito ignoriamo il problema della sincronizzazione e ignoriamo i tempi di trasmissione sui canali fisici; tempi e costi nulli in realtà sono irrealizzabili.*

1.3.1 Alcuni modelli di reti di interconnessione

- Modello a connessione vettoriale (figura 1.1).

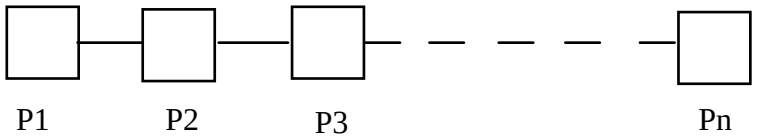


Figura 1.1

Ogni processore comunica con i suoi due adiacenti (ad eccezione del primo e dell'ultimo che comunicano rispettivamente solo con il successivo e solo con il precedente).

- Modello a connessione matriciale (figura 1.2).

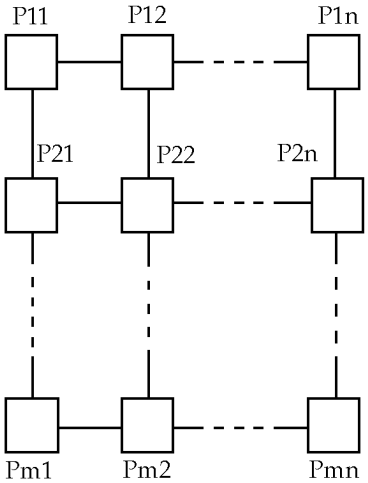


Figura 1.2

Ogni processore ha quattro adiacenti con cui comunicare (esclusi i processori ai bordi).

- Modello a ipercubo, cubo n-dimensionale (figura 1.3).

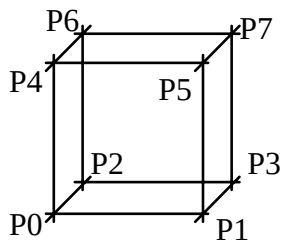


Figura 1.3

Ogni processore ha 2n adiacenti, dove n è il numero delle dimensioni. Nella figura n è uguale a 3; i processori sono 8 e ciascuno ha solo 3 processori adiacenti perchè tutti i processori si trovano ai confini della struttura, quindi gli altri processori adiacenti sarebbero fuori della struttura stessa come indicano i tre segmenti uscenti da ciascun vertice.

- Modello ad albero binario (figura 1.4).

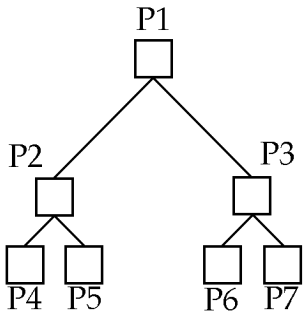


Figura 1.4

Ciascun processore comunica con i suoi figli sinistro, destro e con il padre.

***Nota**: nel seguito faremo sempre riferimento alle numerazioni dei processori nelle strutture così come riportate dalle figure a meno di esplicita negazione.*

1.4 Trasportabilità degli algoritmi

Le distinzioni fatte per le macchine a memoria condivisa permettono di trasportare i vari algoritmi da macchine ad alto livello in altre a basso livello, pagando in tempo ed occupazione della memoria. Ad esempio l'algoritmo di accesso in memoria concorrente ha tempo unitario su macchine CRCW, mentre ha tempo logaritmico se simulato (***broadcasting***) su macchine EREW.

1.4.1 Simulazione della lettura concorrente

Simuliamo una macchina CRXW su di una macchina ERXW (con X indichiamo che non occorre specificare se si tratta di un accesso esclusivo o concorrente; ossia in questo caso lavoriamo su di una macchina a lettura concorrente e scrittura concorrente o esclusiva). Abbiamo un dato D in memoria centrale che deve essere letto da n processori. Utilizziamo in memoria centrale un vettore A di dimensione n

che contiene una cella riservata a ciascun processore. Prevedere una locazione di memoria per ciascun processore non comporta seri problemi pratici in quanto il numero di processori presenti nei calcolatori paralleli è in genere molto limitato. Nella fase di inizializzazione, un processore (noi utilizzeremo ragionevolmente il primo) legge il dato D e lo carica in A[1], cella del vettore a lui riservata, e in una cella B_1 contenuta nella sua memoria locale. A questo punto inizia la fase ciclica: il secondo processore memorizza il contenuto di A[1] nella sua memoria locale e lo carica in A[2]; il terzo ed il quarto processore, contemporaneamente, recuperano il dato da A[1] ed A[2] rispettivamente, lo trascrivono nella propria memoria locale, quindi lo memorizzano in A[3] ed A[4] rispettivamente e così via, raddoppiando ad ogni passo il numero di processori che operano. La durata del ciclo di trasmissione sarà logaritmica nella dimensione n.

Nota: *negli algoritmi di seguito riportati, vengono trascurati fattori di arrotondamento come parti intere o approssimazioni in caso si abbiano numeri che non sono potenze intere di 2. Le valutazioni della complessità restano comunque valide perché effettuate per ordini di grandezza, quindi indipendenti da fattori costanti.*

```
P1:   B1:=D;
P1:   A[1]:=B1;
for i:=0 to log2(n)-1 do
      for j:=2^i+1 to 2^(i+1) in parallel do
            Pj:   begin
                        Bj:=A[j-2^i];
                        A[j]:=Bj;
                  end;
```

1.4.2 Simulazione della scrittura concorrente

Simuliamo una macchina XRCW su di un modello XREW. Utilizziamo una politica di scrittura concorrente che permette di memorizzare l'informazione solo al primo processore (quello di indice minore) nel caso in cui tutti i processori vogliono scrivere lo stesso dato.

L'idea è quella di confrontare l'informazione contenuta nel processore P_i con quella del processore $P_{i+n/2}$ per i=1,…,n/2 in parallelo. Utilizzeremo un flag inizializzato a TRUE nella memoria locale di ciascun processore per indicare se il confronto tra i dati ha avuto esito positivo o meno. Se il processore P_i trova nel processore $P_{i+n/2}$ la stessa sua informazione e i due flag sono TRUE, lascia il valore TRUE nel suo flag, scrive FALSE altrimenti. Al secondo passo si ripete la stessa operazione sulla prima metà dei processori utilizzati al passo precedente e così via fino all'ultimo passo in cui, se è stato generato un FALSE nel flag di qualcuno dei processori durante i passi precedenti, esso verrà propagato sino al primo processore.

```
for i:=1 to n in parallel do
      Pi: bi:=TRUE;
for i:=1 to log2(n) do
      for j:=1 to n/2^i in parallel do
            Pj: if (bj=FALSE) or (bj+n/2^i=FALSE) or (aj≠aj+n/2^i) then
                  Pj: bj:=FALSE;
P1: if b1=TRUE then M:=a1;
```

1.4.3 Somma di n elementi

Vogliamo sommare n elementi ciascuno contenuto nella memoria locale di un processore, utilizzando alcuni modelli di macchine parallele evidenziando la differenza di costo (prodotto tra complessità temporale e numero di processori) e le modifiche necessarie al codice.

- Modello a reti di interconnessione ad albero binario completo.

Le informazioni sono memorizzate nella memoria locale dei processori foglia e vogliamo trovare la somma nella radice.

```
for i:=1 to log2(n) do
   for j:=n/2^i to (n/2^(i-1))-1 in parallel do
         Pj: aj:=a2j+a2j+1;
```

La complessità di tale algoritmo è dominata dal ciclo seriale che porta ad un tempo di O(log(n)); poiché utilizziamo O(n) processori, il costo è O(nlog(n)).

- Modello a rete di interconnessione a matrice.

Abbiamo una matrice quadrata bidimensionale di lato $k=\sqrt{n}$ in cui ogni processore contiene un dato e vogliamo trovare la somma nel processore P_{kk}.

```
for h:=1 to k in parallel do
   for i:=2 to k do
         Phi:  ahi:=ahi+ah(i-1);
for i:=2 to k do
   Pik:  aik:=aik+a(i-1)k;
```

Anche in questo caso il tempo di esecuzione dell'algoritmo è determinato da cicli seriali. L'algoritmo avrà una complessità di $O(k=\sqrt{n})$. Per quanto riguarda il costo, utilizzando n processori si avrà $O(n^{3/2})$.

- Modello a reti di interconnessione a ipercubo.

Utilizziamo un ipercubo di dimensione d con 2^d processori. Associamo a ciascun processore un indice compreso tra 0 e 2^d -1 codificato in binario. Connettiamo tra loro i processori il cui indice differisce per una sola cifra (figura 1.5).

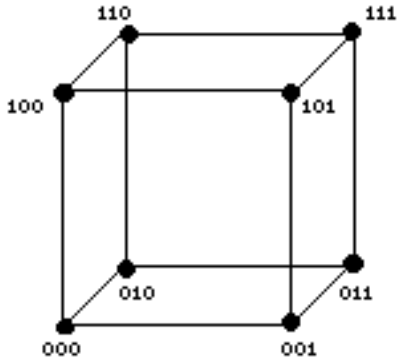


Figura 1.5

La struttura ad ipercubo è di tipo iterativo: possiamo passare ad una struttura con d+1 dimensioni semplicemente duplicando la struttura esistente, aggiungendo un bit alla codifica e rispettando le regole di connessione specificate (figura 1.6). La figura 1.5 mostra un ipercubo di dimensione 3, mentre la figura 1.6 ne mostra uno di dimensione 4.

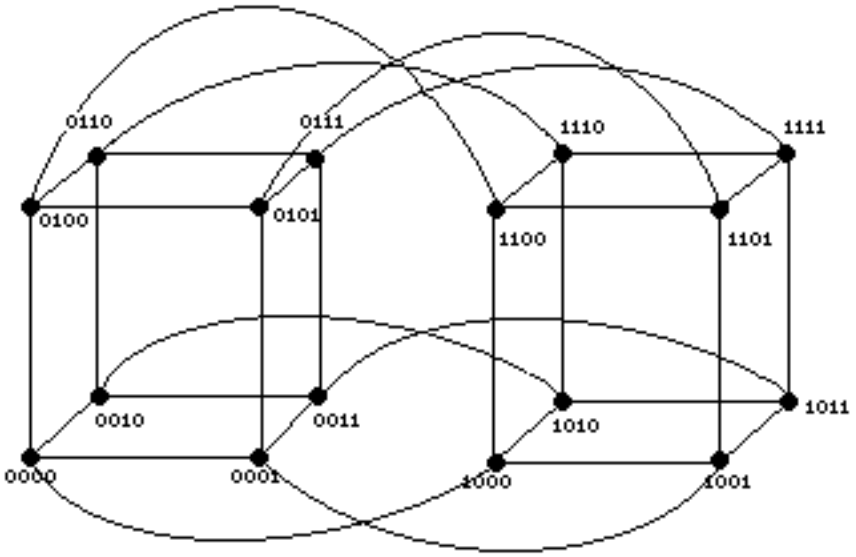


Figura 1.6

Riferendoci al caso particolare della figura 1.5 l'idea è che ogni processore contiene nella propria memoria locale uno dei dati da sommare e, al passo generico, si opera la somma parallela nel modo seguente:
ad ogni iterazione si memorizza il risultato parziale nella memoria del processore con indice minore in ciascuna delle coppie di processori operanti in parallelo;

dimezzando le dimensioni del problema ad ogni passo si ottiene un tempo logaritmico (figura 1.7).

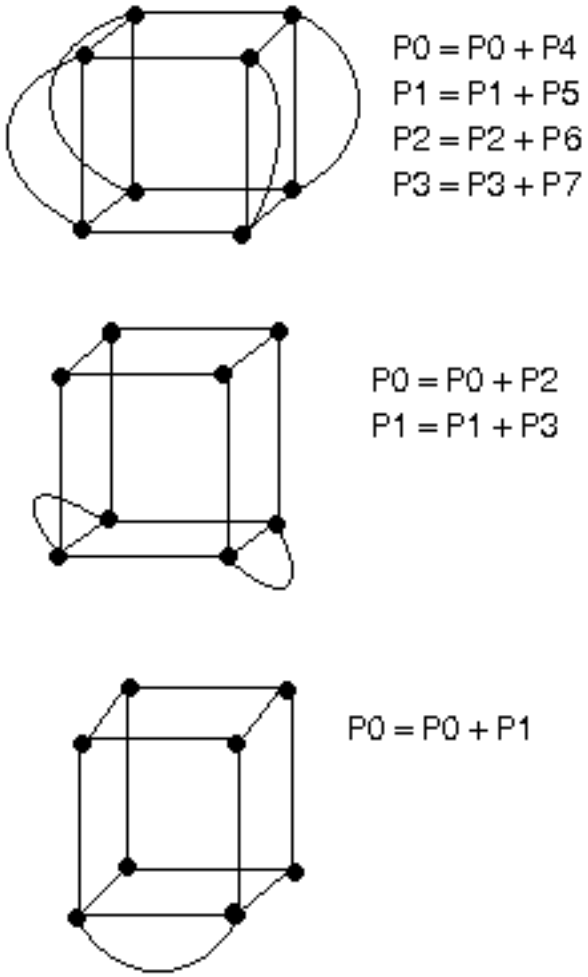


Figura 1.7

L'algoritmo si riferisce al caso generale di dimensione d.

```
for h:=0 to 2^d-1 in parallel do
   P_h:   for L:=d-1 to 0 do
              if(0≤h≤2^L-1) then
                   a_h:=a_h+a_(2^L+h);
```

La complessità dell'algoritmo è lineare in d, quindi logaritmico in n. Il costo è O(nlog(n)).

- Modello EREW a memoria condivisa.

Al primo passo ciascun processore copia il proprio dato nella memoria condivisa in un vettore di dimensione n in modo da garantire la corrispondenza univoca tra

locazioni di memoria del vettore e processori. Utilizziamo il vettore come rappresentazione di un albero binario completo e quindi applichiamo un algoritmo analogo a quello relativo alla simulazione di scrittura concorrente. L'algoritmo si basa sulle proprietà associativa e commutativa della somma, raggruppando i dati, nel caso di 8 elementi, come segue:

$$\{[(x_1+x_5)+(x_2+x_6)]+[(x_3+x_7)+(x_4+x_8)]\}$$

- Modello EREW a memoria condivisa con struttura a matrice.

Ogni processore può essere considerato connesso a ciascuno degli altri.

```
for h:=1 to n in parallel do
   for i:=log2(n) to 1 do
         for j:=1 to 2^(i-1) in parallel do
               Phj:  ahj:=ahj+ah(j+2^(i-1));
for i:=log2(n) to 1 do
   for j:=1 to 2^(i-1) in parallel do
         Pj1:  aj1:=aj1+a(i+2^(i-1))1;
```

La complessità è logaritmica in n ed il costo O(nlog(n)).

1.4.4 Un ulteriore esempio

Problema: ricercare un elemento di valore x in un vettore di dimensione n in una macchina a memoria condivisa con M processori.

Si lavora sotto le ipotesi che ogni processore cerca l'elemento in una porzione di dimensione n/M del vettore; inoltre si suppone che tutti gli elementi siano distinti (il valore x appare al più una volta). Vediamo i passi dell'algoritmo.

1) si trasmette ad ogni processore il dato "x" ed il valore n;

2) in memoria condivisa si utilizza un vettore di booleani B di dimensione M; il processore P_i inizializza la posizione relativa all'indice i con il valore FALSE e ricerca nella sua porzione del vettore il dato x. Se lo trova, pone la i-esima cella di B a TRUE;

3) per ottenere il risultato nel primo processore, ciascun processore controlla il proprio flag; se il flag è TRUE esso viene copiato nella prima cella di B (al più un solo processore cercherà di scrivere).

1.5 Complessità nel calcolo parallelo

Un approccio un po' più rigoroso allo studio della complessità di programmi paralleli ci suggerisce di fare continuo riferimento alle nostre conoscenze nel campo della programmazione sequenziale. Le misure sulle quali basiamo le nostre analisi sono sostanzialmente cinque:

- tempo di esecuzione;
- numero di processori;
- costo;
- speed-up;
- efficienza;

consideriamo anche misure legate alla tecnologia quali:

- area occupata;
- connessioni fisiche;
- periodo del circuito.

1.5.1 Misure indipendenti dalla tecnologia

In una macchina parallela il ***tempo di esecuzione*** si calcola come differenza tra il tempo in cui tutti i processori hanno terminato il loro lavoro e il tempo di inizio del processo. Il tempo di esecuzione di un algoritmo parallelo si compone generalmente di due fasi:

- fase parallela, in cui vengono effettuati i cicli di calcolo;
- fase di trasmissione, in cui avviene la comunicazione dei dati.

Per esempio nell'algoritmo di somma di n elementi su modello a reti di interconnessione a matrice visto nel paragrafo 1.4.3, la fase parallela è costituita dalle seguenti linee di codice:

```
for h:=1 to k in parallel do
   for i:=2 to k do
         Phi:  ahi:=ahi+ah(i-1);
```

mentre quella di trasmissione è:

```
for i:=2 to k do
   Pik:  aik:=aik+a(i-1)k;
```

Per quanto riguarda il ***numero di processori***, occorre precisare che viene considerato solamente il massimo numero di processori effettivamente utilizzati contemporaneamente durante le varie fasi del processo e non il numero totale di processori disponibili. Si approssimerà tale numero al numero totale di processori disponibili (quindi per eccesso) qualora non sia conveniente cercare una stima più precisa.

Il ***costo*** di un algoritmo è dato dal prodotto del tempo di esecuzione per il numero di processori. Se ho un algoritmo parallelo che sfrutta M processori e devo implementarlo su di una macchina con M'<M processori, dovrò pagare un costo dato dal lavoro seriale che ciascuno degli M' processori deve svolgere in più. Tale variazione del numero di processori è inversamente proporzionale alla variazione del tempo di esecuzione. Per qualunque algoritmo parallelo che abbia nel seriale l'algoritmo ottimo può essere calcolato il limite inferiore alla sua complessità:

$$\Omega(t_s/M),$$

ove t_s rappresenta la complessità seriale ottima.

Lo ***speed-up*** è una misura utile a determinare quanto un algoritmo parallelo sia veloce rispetto al migliore algoritmo seriale conosciuto per lo stesso problema. Si calcola come rapporto tra il tempo impiegato nel caso pessimo dal miglior algoritmo seriale e il tempo del nostro algoritmo parallelo. Se si considera lo speed-up a partire da un algoritmo seriale ottimo, lo speed-up ideale è uguale al numero di processori, poiché un algoritmo che lavora su M processori può essere al più M volte più veloce del corrispondente algoritmo seriale. Quando si raggiunge lo speed-up ottimo si ottiene l'algoritmo parallelo ottimo (se lo era il seriale). Nel caso in cui lo speed-up sia maggiore di M, l'algoritmo seriale utilizzato non è ottimo e se ne può trovare uno migliore eseguendo serialmente i passi dell'algoritmo parallelo utilizzato.
Indipendentemente da calcoli teorici bisogna ricordare che il numero M di processori risulta vincolato da considerazioni di carattere tecnologico ed economico (costo iniziale e di manutenzione); in pratica si cerca di rendere M indipendente dalla dimensione n del problema e comunque abbastanza piccolo. Volendo esprimere il rapporto esistente tra M ed n, possiamo utilizzare un'equazione del tipo:

$$M=f(n)=n^{1-x} \quad \text{con } 0 \leq x \leq 1.$$

L'***efficienza*** è definita come il rapporto tra la complessità nel caso pessimo dell'algoritmo seriale ottimo e il costo dell'algoritmo parallelo; quindi tale misura è generalmente minore o uguale a 1. Se risultasse maggiore di 1 allora l'algoritmo seriale non sarebbe ottimo e possiamo trovarne uno migliore simulando l'algoritmo parallelo con l'esecuzione seriale degli stessi calcoli dell'algoritmo parallelo su di un unico processore.

1.5.2 Misure legate alla tecnologia

Tutte le misure legate alla tecnologia sono riconducibili a meno di costanti a quelle finora viste, le quali dipendono esclusivamente dal numero di processori. Se ad esempio consideriamo la tecnologia VLSI che permette di assemblare in un chip di $1cm^2$ più o meno un milione di porte di commutazione, possiamo allora parlare di ***area occupata*** come occupazione fisica di spazio del dispositivo. Poiché il costo è anche legato al numero di processori, è necessario minimizzare l'area occupata dai processori per ridurne il costo. Di conseguenza nasce il problema dell'ottimizzazione della disposizione all'interno di un chip delle porte logiche che costituiscono i processori, dei chip nella piastra che li contiene e della lunghezza delle ***connessioni fisiche***, generalmente organizzate secondo una struttura a griglia. Questo comporta che alcune strutture vengono implementate in maniera naturale (reti di interconnessione a matrice) e sarà dunque semplice trovare una disposizione ottimale, mentre per altre sorge la necessità di ottimizzare la dimensione delle connessioni le quali potrebbero anche avere lunghezza non omogenea. Nel caso di reti di interconnessione ad albero binario completo, una possibile implementazione è data dallo schema di figura 1.8:

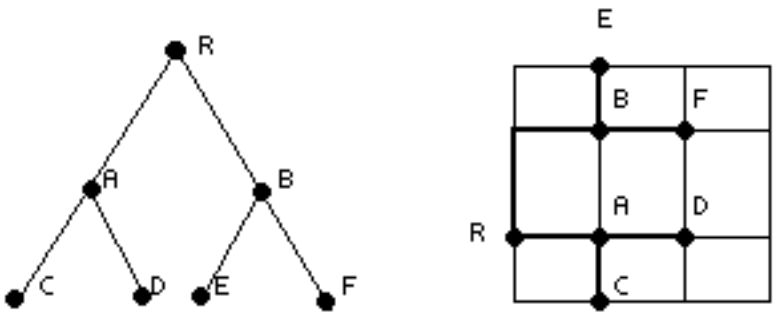


Figura 1.8

Poiché parlando di complessità di un algoritmo dobbiamo tener conto del tempo di trasmissione dei dati, risulta ovvio che tale misura risulti legata al tempo di esecuzione.
Consideriamo di nuovo la somma di elementi su struttura ad interconnessione ad albero binario completo, modificando leggermente il problema:
dati m blocchi di n elementi ciascuno, effettuare le m somme.
Utilizzando l'algoritmo risolutivo già visto, otteniamo una complessità O(mlog(n)) che può essere migliorata mediante un algoritmo di questo tipo:
non appena i processori di un dato livello hanno effettuato la somma dei dati contenuti nei figli, forniamo immediatamente nuovi dati ai figli senza aspettare che i precedenti arrivino alla radice (***pipeling***). Quindi ogni processore trasmette il dato al padre e, mentre il padre effettua la somma, recupera i dati dai figli per sommarli e memorizzarli. Questo algoritmo porta ad una complessità

$$O(m-1+\log(n))=O(m+\log(n))$$

Visto che ad ogni tempo unitario (ossia il tempo con il quale forniamo i blocchi di dati) abbiamo una somma in uscita, si dice che il ***periodo del circuito*** è unitario.

2 Tecniche di programmazione

2.1 Tecnica del salto del puntatore

Data una lista di n elementi con un processore associato ad ogni singolo record, l'idea è quella di "raddoppiare" il puntatore in modo che passo passo ogni puntatore indirizzerà il successivo del successivo. Poiché questo computo è fatto in parallelo, ad ogni passo la distanza da un singolo record al suo successivo sarà raddoppiata (figura 2.1).

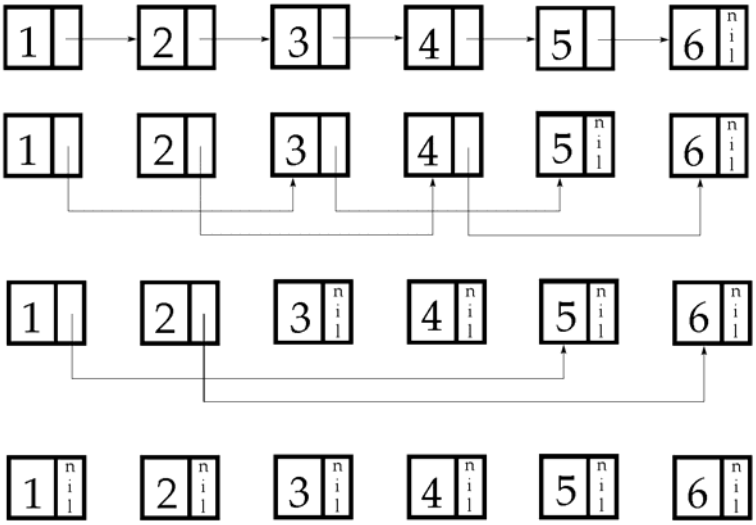


Figura 2.1

Per ovvie ragioni la configurazione iniziale della lista verrà persa e quindi se serve deve essere preventivamente duplicata.

2.2 Applicazioni del salto del puntatore

2.2.1 Calcolo della posizione all'interno di una lista

Data una lista di n elementi, calcolare per ciascuno di essi la distanza dall'ultimo elemento e memorizzarla nel relativo campo; supponendo di avere almeno n processori e di essere su di una SIMD EREW.

0) inizializzazione

```
for i:=1 to n in parallel do
      Pi: if succ(i)=nil then info(i):=0 else info(i):=1;
```

1) raddoppio del puntatore

```
for i:=1 to log2(n) do
      for j:=1 to n in parallel do
            Pj: if succ(j)≠nil then
            begin
                  info(j):=info(j)+info(succ(j));
                  succ(j):=succ(succ(j));
            end;
```

Per una lista di otto elementi l'algoritmo procede come in figura 2.2:

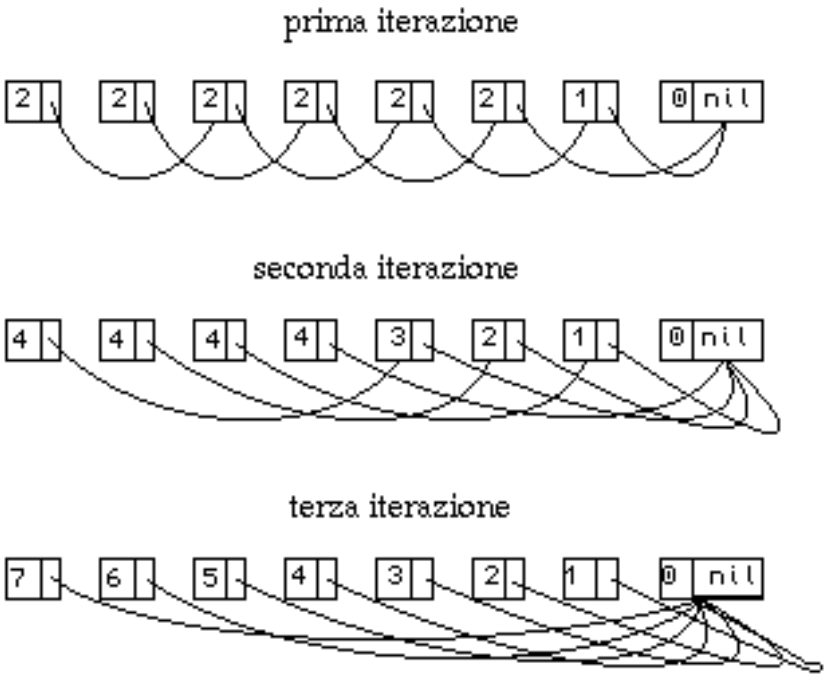


Figura 2.2

La lista viene inizializzata con complessità O(1) (passo 0), stessa complessità è ottenuta nell'aggiornamento dei campi info e succ (fine passo 1); quindi la complessità finale deriva dal ciclo seriale (O(log(n)) a dispetto di un O(n) di un algoritmo sequenziale).

2.2.2 Calcolo delle somme prefisse

Dato in input un vettore Y di n elementi vogliamo calcolare

$$\sum_{i=1}^{n} Y_i$$

mediante il calcolo iterativo delle somme parziali $S_k=S_{k-1}+Y_k$, k=1…n, con $S_0=0$.
Memorizziamo gli n elementi della sommatoria in una lista su cui operare con la tecnica del salto del puntatore.

Lavoriamo su di una macchina di tipo SIMD EREW a memoria condivisa con almeno n processori (vedere lo schema in figura 2.3).

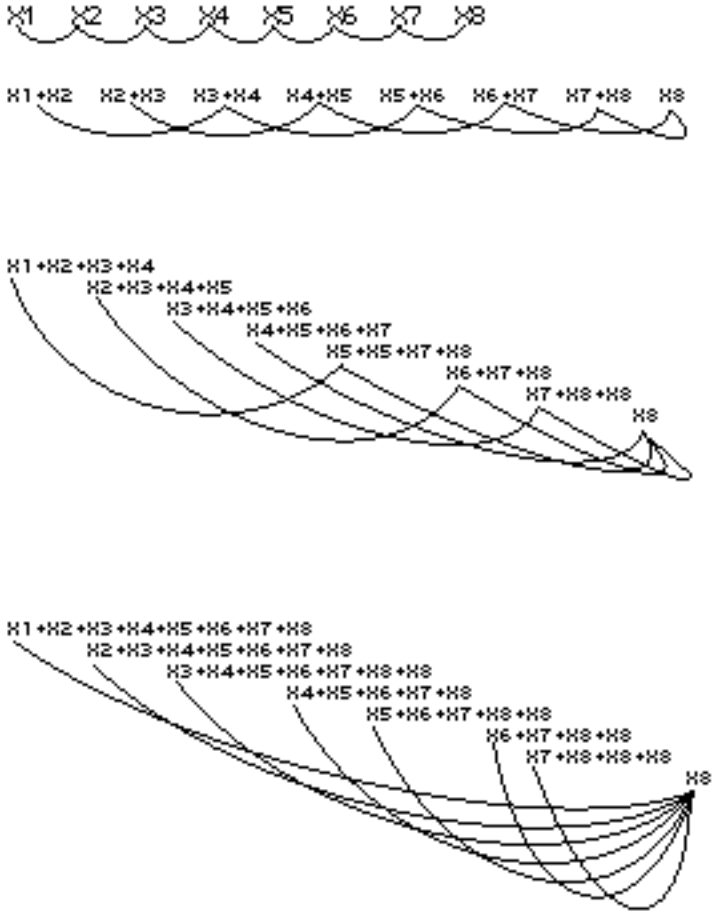


Figura 2.3

0) inizializzazione

```
for i:=1 to n in parallel do
     Pi: info(i):=Y[i];
```

1) incremento del puntatore

```
for i:=1 to log2(n) do
     for j:=1 to n in parallel do
          Pj: if succ(j)≠nil then
          begin
               info(j):=info(j)+info(succ(j));
               succ(j):=succ(succ(j));
          end;
```

Il metodo utilizzato per il calcolo delle somme prefisse continua a valere per ogni altra operazione aritmetica che goda della proprietà associativa (ad esempio la somma modulo due).

2.2.3 Calcolo delle radici in una foresta di alberi binari radicati

Data un foresta di alberi binari radicati vogliamo, per ogni nodo della foresta, determinare la radice dell'albero a cui appartiene.
Lavoriamo su un modello di macchina SIMD CREW ad n processori, uno per ogni nodo della foresta.
Rappresentiamo gli alberi mediante liste: ogni cella contiene un campo info che indica il nodo a cui è associata e un campo parent che inizialmente punta al padre del nodo.
Utilizziamo un vettore di appoggio Root con un numero n di elementi, la cui cella i-esima Root[i] conterrà al termine dell'esecuzione la radice dell'albero a cui il nodo i appartiene.

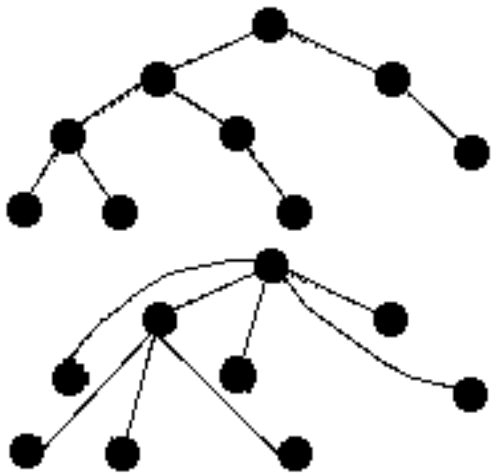
0) inizializzazione delle posizioni del vettore Root corrispondenti ai soli nodi che sono radici assegnando loro il valore i.

```
for i:=1 to n in parallel do
     Pi: if parent(i)=nil then Root[i]:=i;
```

1) ascesa di figlio in padre

```
while there exists a node i such that parent(i)≠nil do
     for i:= 1 to n in parallel do
          Pi: if parent(i)≠nil then
          begin
               Root[i]:=Root[parent(i)];
               parent(i):=parent(parent(i));
          end;
```

La necessità di un modello a lettura concorrente deriva dal possibile conflitto di lettura presente nel ciclo parallelo del passo 1.
La valutazione della complessità deve tener conto delle diverse profondità degli alberi della foresta, per cui il costo dell'intero algoritmo è logaritmico nella massima profondità.



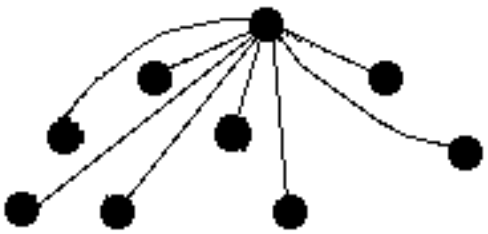


Figura 2.4

2.3 Confronto tra algoritmi CRCW e EREW

Teorema*: È possibile ordinare n elementi su di una macchina PRAM EREW in un tempo dell'ordine del logaritmo di n.*

La dimostrazione di questo teorema verrà omessa.

Teorema*: Un algoritmo CRCW a p processori non può essere più di O(log(p)) volte più veloce del miglior algoritmo EREW a p processori per lo stesso problema.*

Dimostrazione: supponiamo di avere una situazione di conflitto in scrittura del tipo di figura 2.5:

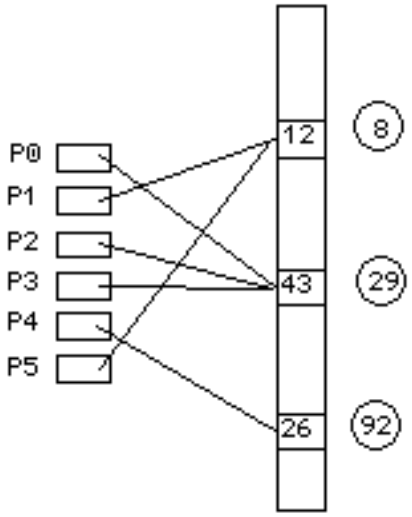


Figura 2.5

utilizziamo un vettore V contenuto nella memoria condivisa di dimensione p la cui locazione i, posta in corrispondenza col processore P_i, è un record a due campi di cui il primo contiene un indirizzo di memoria e il secondo un'informazione. Simuliamo la scrittura concorrente (sotto l'ipotesi di processori che vogliono scrivere informazioni identiche) su di una macchina a scrittura esclusiva.

1) Ogni processore scrive nella sua locazione all'interno del vettore V l'indirizzo della cella di memoria in cui vorrebbe scrivere, nel primo campo, e l'informazione nel secondo; tale accesso avviene ovviamente in maniera esclusiva (figura 2.6).

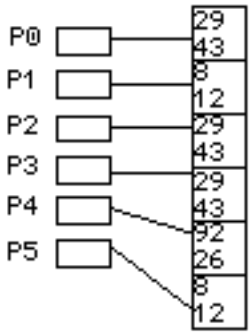


Figura 2.6

2) Ordiniamo gli elementi di V rispetto al primo campo, indirizzo di memoria, (figura 2.7).

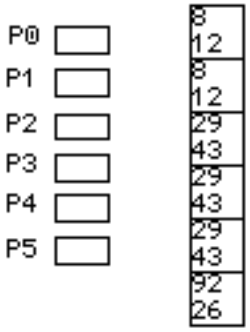


Figura 2.7

3)

3.1) Il primo processore scrive nell'indirizzo contenuto nel primo campo il dato indicato dal secondo.

3.2) Ogni processore, dal secondo in poi, <u>in parallelo</u> controlla se il suo record è uguale a quello del processore precedente; in tal caso non fa nulla, altrimenti scrive (figura 2.8).

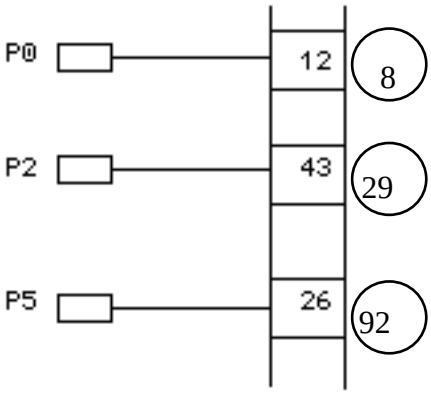


Figura 2.8

In definitiva poiché il primo ed il terzo passo richiedono tempo costante, la simulazione ha la complessità dell'ordinamento, che, per il teorema precedente, è dell'ordine del logaritmo di p.
Per il caso della lettura si utilizza il broadcast.

c.v.d.

2.4 Ordinamento

2.4.1 Premesse

Per questo problema non utilizzeremo il modello PRAM, ma un modello semplificato costruito appositamente. Utilizziamo il comparatore, ossia un circuito di confronto con due ingressi e due uscite il cui valore è la coppia di valori in input ordinati in modo ascendente (figura 2.9).

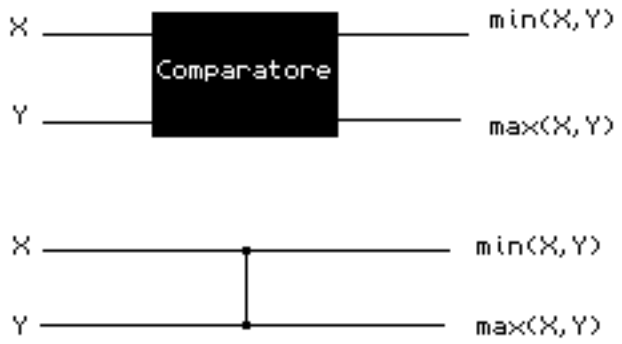


Figura 2.9

I circuiti formati da comparatori, se opportunamente combinati, costituiscono un'architettura in grado di ordinare n valori di input (figura 2.10).

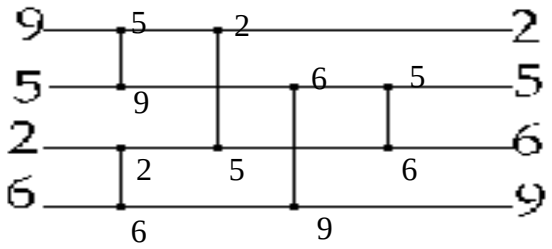


Figura 2.10

Ad esempio in figura 2.11 è mostrata un'architettura che implementa l'insertion sort:

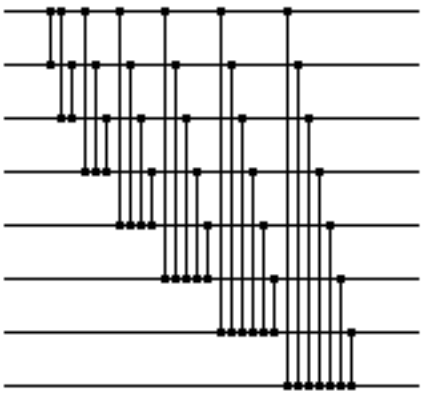


Figura 2.11

Questa macchina utilizza una versione seriale dell'algoritmo di complessità $O(n^2)$ (indicata dalla profondità del circuito, ossia il numero di porte che vengono attraversate in tempi differenti); mentre è possibile ottenere una complessità $O(n)$ ottimizzando lo scheduling dei comparatori e parallelizzando l'algoritmo (figura 2.12):

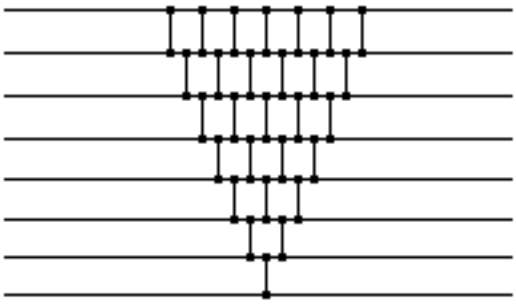


Figura 2.12

Lemma: *Se un circuito di ordinamento trasforma la sequenza di input* $a=(a_1,a_2,\ldots,a_n)$ *nella sequenza di output* $b=(b_1,b_2,\ldots,b_n)$*, allora per ogni funzione monotona crescente f, il circuito trasforma la sequenza di input* $f(a)=(f(a_1),f(a_2),\ldots,f(a_n))$ *nella sequenza di output* $f(b)=(f(b_1),f(b_2),\ldots,f(b_n))$.

Teorema (principio 0/1)*: se un circuito combinatorio di ordinamento lavora correttamente per qualunque input costruito sull'alfabeto {0,1} allora lavora correttamente per qualunque input costruito su di un qualsiasi alfabeto finito A.*

Dimostrazione: supponiamo per assurdo che il circuito ordini tutte le sequenze costruite sull'alfabeto {0,1} correttamente, ma che esista una sequenza di input di numeri arbitrari $a=(a_1,a_2,\ldots,a_n)$ contenente elementi a_i e a_j tali che $a_i<a_j$ mentre il circuito pone a_j prima di a_i nella sequenza di output. Definiamo f una funzione monotona crescente come:

$$f(x)=\begin{cases} 0 \text{ se } x\leq a_i \\ 1 \text{ se } x>a_i \end{cases}$$

dal lemma precedente segue che il circuito sistema $f(a_j)$ prima di $f(a_i)$ nella sequenza di output quando f(a) è l'input. Ma poiché $f(a_j)=1$ mentre $f(a_i)=0$, neghiamo l'ipotesi giungendo ad un assurdo.

c.v.d.

Mediante questo risultato, è possibile ragionare solo su stringhe binarie e utilizzare i risultati raggiunti per stringhe di qualsiasi tipo.

Definizione: *si definisce* <u>*sequenza bitonica*</u> *una sequenza che può essere divisa in due sottosequenze monotone, una crescente e l'altra decrescente o viceversa.*

Definizione: *data la sequenza* $S=s_1,\ldots,s_{2n}$ *definiamo le due sequenze:*

$$m(S)=(\min\{s_1,s_{n+1}\},\ \min\{s_2,s_{n+2}\},\ldots,\min\{s_n,s_{2n}\})$$
$$M(S)=(\max\{s_1,s_{n+1}\},\ \max\{s_2,s_{n+2}\},\ldots,\max\{s_n,s_{2n}\})$$

Proprietà: *m(S) ed M(S), ottenute da una sequenza S bitonica, sono bitoniche.*

Sfruttando la definizione di m(S) ed M(S) e le relative proprietà si può ottenere una definizione ricorsiva per le sequenze bitoniche che ci permette di realizzare un primo algoritmo di ordinamento che opera ricorsivamente secondo lo schema di figura 2.13:

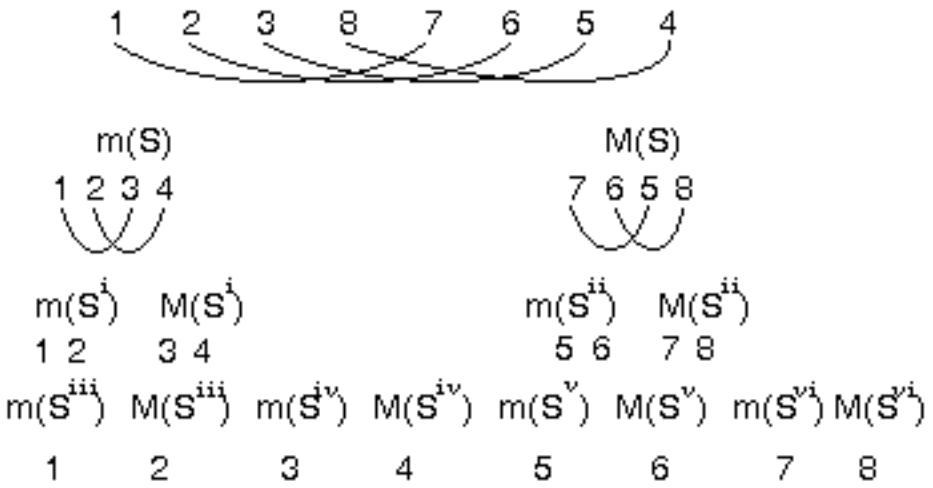


Figura 2.13

da cui deriva una complessità O(log(n)).

Il principio 0/1 ci permette d'ora in poi di utilizzare solo sequenze binarie.

Proprietà: *una sequenza 0/1 è bitonica se è nella forma* $0^i1^j0^k$ *oppure* $1^i0^j1^k$ *per* $i,j,k\geq 0$.

Definizione: *una* <u>*sequenza binaria*</u> *si dice* <u>*pulita*</u> *se è composta interamente da 0 o da 1.*

***Proprietà**: se S è bitonica almeno una delle due sottosequenze bitoniche m(S) e M(S) è pulita.*

Tale proprietà è diretta conseguenza del fatto che la cardinalità di {0,1} è due.
A partire da tali proprietà costruiamo un circuito di ordinamento applicabile a sequenze 0/1 bitoniche.
Parallelizzando adeguatamente i comparatori otteniamo un circuito di profondità logaritmica poiché ad ogni passo rendiamo pulita metà della sequenza (figura 2.14).

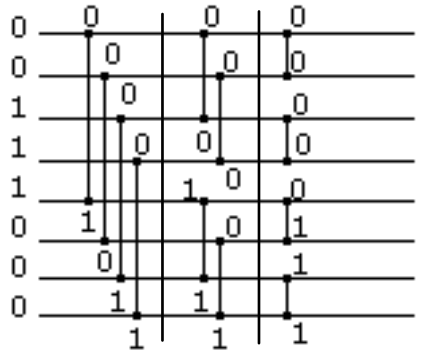


Figura 2.14

2.4.2 Circuiti di ordinamento
Generalizziamo il circuito di ordinamento bitonico per ottenere un circuito (detto di merge) che fonde due sequenze ordinate costruite sullo stesso alfabeto sfruttando il fatto che, date due sequenze ordinate entrambe crescenti (o decrescenti) x e y, la sequenza che si ottiene concatenando x con z=y rovesciata è bitonica. È da notare che l'inversione della stringa y è realizzata con connessioni opportune (figura 2.15). Successivamente sfruttiamo più circuiti di merge concatenati per ottenere un circuito di ordinamento applicabile a sequenze qualunque.

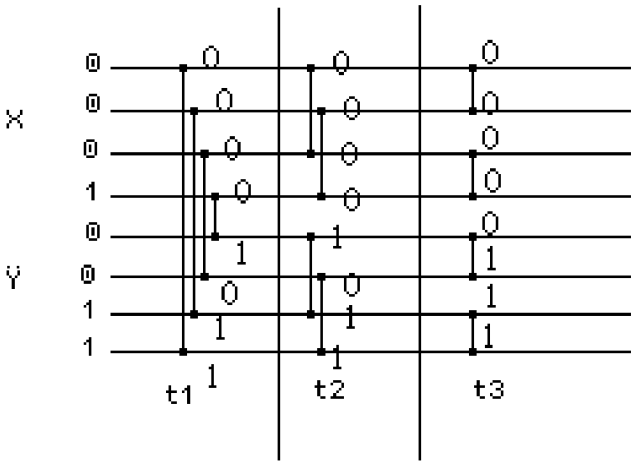


Figura 2.15

La profondità del circuito è logaritmica e il numero di comparatori ad ogni passo è n/2.

Visto che ogni sequenza di due elementi è bitonica, possiamo sfruttare il circuito di merge sulle coppie per ottenere stringhe ordinate di dimensione doppia e procedere iterativamente con altri circuiti di merge (figura 2.16).

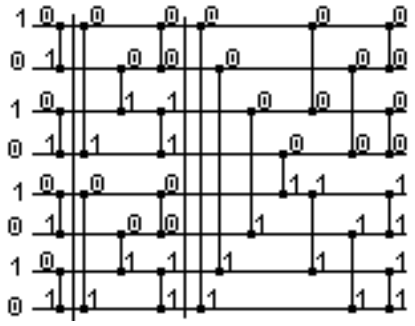


Figura 2.16

Poiché concateniamo un numero logaritmico di circuiti di merge, otteniamo una complessità $O(\log^2(n))$, vedere figura 2.16.

2.4.3 Teorema di Brent

Teorema: *ogni algoritmo che lavora su un circuito combinatorio di profondità d e dimensione n che sia a fan-in limitato, può essere simulato da un algoritmo che lavora su una PRAM CREW con p processori in O(n/p+d) tempo.*

Dimostrazione: prendiamo un circuito che soddisfa le ipotesi del teorema. Siano $p_1, p_2, \ldots p_p$ i processori e supponiamo di avere almeno tante celle di memoria condivisa quanti sono gli output degli elementi combinatori che formano il circuito. Supponiamo poi che p sia maggiore o uguale al massimo tra il numero di elementi combinatori utilizzati ad ogni istante di tempo. Inizialmente mettiamo l'input del circuito in una struttura in memoria condivisa in modo che ogni input sia a disposizione dei processori che dovranno utilizzarlo. Ogni processore simula un componente del circuito e fornisce l'output (limitato) che può essere immagazzinato nuovamente in memoria condivisa. Si ripete l'operazione per un numero di volte pari alla profondità del circuito utilizzando l'output di ogni livello come input per il livello successivo. In tal caso la complessità è pari alla profondità del circuito in quanto non è possibile parallelizzare ulteriormente l'algoritmo (trovandoci a livello k dobbiamo aspettare l'esecuzione al livello k-1 prima di eseguire i calcoli). Nel caso in cui p sia minore del massimo numero di elementi combinatori presenti in uno stesso livello, non potendo in qualche livello simulare contemporaneamente tutti i componenti di quel livello, dovrò effettuare un numero di passi seriali proporzionale al rapporto tra il numero di elementi del livello e p. In tal caso la complessità terrà conto della profondità del circuito e dei passi seriali necessari a simulare un livello; maggioreremo tale quantità con n/p+d. L'ipotesi del fan-in limitato ci serve per evitare che la memoria non sia sufficiente a memorizzare i risultati intermedi del calcolo, mentre la lettura concorrente ci permette di fornire uno stesso risultato parziale come input per più processori.

c.v.d.

Esistono due varianti al teorema di Brent:

***Teorema (seconda versione)**: ogni algoritmo che lavora su un circuito combinatorio di profondità d e dimensione n che sia a fan-in e fan-out limitato (fan-out uguale a uno), può essere simulato da un algoritmo che lavora su una PRAM EREW con p processori in O(n/p+d) tempo.*

***Dimostrazione**:* è analoga alla precedente con la differenza che il fan-out limitato a 1 ci permette di evitare che più processori debbano leggere dalla stessa posizione di memoria contemporaneamente.

c.v.d.

***Teorema (terza versione)**: ogni algoritmo che lavora su una PRAM a p processori può essere simulato da un algoritmo che lavora su una PRAM a p' processori (p'<p) in tempo O(tp/p') dove t è il tempo parallelo nella prima PRAM.*

***Dimostrazione**:* nella prima PRAM l'elaborazione avviene in t passi (ad ogni passo i p processori lavorano parallelamente); avendo invece a disposizione un numero inferiore (p'<p) di processori, ciascuno dei p' processori eseguirà un blocco seriale di p/p' operazioni. La complessità di ciascuno dei t passi è O(p/p') nella seconda PRAM, mentre era O(1) nella prima.

c.v.d.

Il costo dell'algoritmo simulato sulla seconda PRAM è uguale a quello dell'algoritmo originale sulla prima; infatti abbiamo un costo pari a tp che nel secondo caso viene dal prodotto di p' processori con la complessità O(tp/p'), mentre nel primo viene dal prodotto di p processori con la complessità O(t).

2.4.4 Applicazioni del teorema di Brent

Parlando di simulazione di macchina CRCW su EREW nel paragrafo 2.3 abbiamo fatto riferimento alla possibilità di effettuare l'ordinamento in tempo logaritmico con un teorema di cui segue una giustificazione; abbiamo visto un circuito combinatorio composto da log(n) circuiti di merge, ciascuno di profondità log(n). Considerando la seconda versione del teorema di Brent, sappiamo che possiamo ottenere l'ordinamento di n elementi in O(c/p+t) tempo dove c è il numero di comparatori e t è la profondità del circuito ($O(\log^2(n))$). Essendo c=n/2 per ogni blocco parallelo, otteniamo c=O(nlog(n)) e quindi una complessità dell'ordinamento pari a $O((n\log(n)/p)+(\log^2(n))$. Se supponiamo $p \geq n/\log(n)$ si ha che la complessità dell'ordinamento diventa $O(\log^2(n))$.
In realtà avevamo promesso una complessità logaritmica ed infatti è possibile raggiungere tale risultato mediante un circuito (che non verrà riportato) che utilizza solo differenti tecnologie ed ha per di più una costante moltiplicativa molto elevata che lo rende poco significativo.

2.4.5 Algoritmo parallelo di selezione (parallel select)

L'idea è quella di adattare il quicksort seriale ad una PRAM EREW. La tecnica del quicksort consiste nello scegliere un elemento separatore nel vettore di input e suddividere il vettore in tre sottovettori S_1, S_2 ed S_3 contenenti rispettivamente gli elementi strettamente minori, uguali e strettamente maggiori dell'elemento separatore, da riordinare separatamente e fondere successivamente mediante un algoritmo ricorsivo che sfrutta la tecnica divide et impera. In generale dovremo assegnare n^x elementi ad ognuno degli N processori a disposizione con $N=n^{1-x}$, $0<x<1$, dove n è la dimensione dell'input. Tale algoritmo necessita per ogni sottovettore di un elemento separatore; a tale scopo riporteremo una versione parallela dell'algoritmo di selezione seriale.
Utilizziamo una procedura ricorsiva nella dimensione del sottovettore in esame il cui "tappo di ricorsione" è il valore Q, dimensione che consente l'ordinamento (quindi la selezione dell'elemento mediano) in tempo costante. Fissato quindi Q (nel nostro caso n^x) lo schema dell'algoritmo è il seguente:

- se $|S|\leq Q$
    - ordina [1] S;
    - scrivi il mediano nel vettore S' dei mediani;
- altrimenti suddividi S in $|S|/Q$ sottovettori;
    - per ciascuno trova il mediano (ricorsione);
- trova il mediano dei mediani.

Il costo della ricorsione dipende dal numero di sottovettori individuati.
Sapendo trovare il mediano di un vettore possiamo generare S_1, S_2 ed S_3 e il generico elemento in posizione k nella sequenza ordinata sarà determinabile mediante il seguente schema:

- se $k\leq|S_1|$
    - prendi il k-mo elemento di S_1;
- se $|S_1|<k\leq|S_1|+|S_2|$
    - prendi il primo elemento di S_2;
- se $k>|S_1|+|S_2|$
    - prendi l'h-mo elemento di S_3 con $h=k-|S_1|-|S_2|$.

La relazione di ricorrenza che determina la complessità è la seguente:

$$T(n)=\text{costante} * n + T(3n/4) + T(n/Q)$$

la cui soluzione asintotica è O(n).
Per parallelizzare l'algoritmo lavoriamo su di una PRAM EREW ad N processori tali che $N=n^{1-x}$ ($0<x<1$). Ciascun processore riceve un blocco pari a n^x elementi estratti dal vettore di input S. In memoria condivisa abbiamo un vettore d'appoggio M di dimensione N in cui ogni posizione è in diretta corrispondenza con il processore di pari indice. L'algoritmo farà uso delle procedure di broadcast e delle somme parziali.

---

[1]) È possibile utilizzare un qualsiasi algoritmo di ordinamento senza alterare la complessità.

Scegliamo un valore $Q \leq n^x$ fissato in modo da consentire la soluzione diretta del problema da parte di un singolo processore. Ad ogni iterazione controlliamo se $|S|$ è diventata minore o uguale a Q. In tal caso risolviamo direttamente il problema; altrimenti suddividiamo S in sottosequenze S_i $(i=1,\ldots,n^{1-x})$ di lunghezza pari a n^x, ove n rappresenta l'attuale dimensione dell'input, ciascuna delle quali assegnata al processore p_i. Ogni processore riceve l'informazione n e la dimensione di ogni blocco attraverso il broadcast e, conoscendo il proprio indice, calcola gli estremi del sottovettore su cui lavorare che saranno rispettivamente:

$$n*(i-1)+1 \qquad e \qquad n*i+[n*(i-1)+1].$$

La complessità del broadcast è $\log(N=n^{1-x})$; ciascun processore in parallelo effettua la ricerca del mediano fra i suoi elementi e lo memorizza nel vettore M nella posizione relativa al proprio indice. Chiamando la procedura ricorsivamente su M, troviamo il mediano fin tanto che $|M| \leq Q$. Al termine della ricorsione il mediano m si trova in memoria condivisa. Non abbiamo utilizzato S per evitare la perdita dell'input. Noto il mediano m, suddividiamo S in tre sottovettori Lower, Equal e Greater in cui rispettivamente vogliamo memorizzare gli elementi minori, uguali e maggiori di m. A tale scopo ogni processore p_i suddivide S_i nei tre sottovettori L_i, E_i e G_i eventualmente vuoti. Per ottenere L, E e G, applichiamo la procedura delle somme parziali richiamandola su ogni processore.
Siano:

$$a_i=|L_i|, \qquad b_i=|E_i|, \qquad c_i=|G_i|$$

e siano s_0, z_0 ed r_0 le somme parziali iniziali relative alle a_0, b_0 e c_0 di valore nullo.
A partire da queste, ricaviamo le somme successive:

$s_1=a_1+s_0$
$s_2=a_2+s_1$
…
$s_{n^{1-x}}=a_1+a_2+\ldots+a_{n^{1-x}}=|L|$

ed analogamente per E e G ottenuti con le somme parziali a partire da z_0 ed r_0.
In parallelo ogni processore esegue:

scrivi L_i da $s_{i-1}+1$ fino ad s_i;
scrivi E_i a partire da $z_{i-1}+1$ fino ad z_i;
scrivi G_i a partire $r_{i-1}+1$ fino ad r_i.

A questo punto un qualunque processore estrae il k–esimo elemento secondo l'algoritmo seriale visto in precedenza.
La relazione di ricorsione è la seguente:

$$T(n)=(c_{11}+c_{12})*\log(n)+(c_{21}+c_{22})*n^x+T(n^{1-x})+T(3n/4)=O(n^x) \qquad (n>4)$$

dove i contributi logaritmici sono dovuti al broadcast maggiorando n^{1-x} con n, $c_{21}*n^x$ deriva dalla suddivisione in blocchi, $c_{22}*n^x$ per la suddivisione in tre sottovettori e $T(n^{1-x})+T(3n/4)$ per le chiamate alle procedure del calcolo del mediano e dell'elemento k–esimo.
Il costo è dell'ordine di n $(n^{1-x}*n^x)$.

2.4.6 QuickSort parallelo

Per raggiungere un costo ottimo $O(n\log(n))$, utilizziamo una versione parallela del QuickSort. Consideriamo una PRAM EREW nella cui memoria condivisa abbiamo il

vettore S di dimensione n, con $N=n^{1-x}$ processori. Dividiamo in $2^{1/x}$ sequenze di dimensione $n/2^{1/x}$ ciascuna (l'aver diviso il vettore in un numero di sequenze pari ad una potenza di due ci risulterà utile ai fini dell'algoritmo). Calcoliamo gli $m_1, m_2, \ldots, m_{2^{1/x}}$ separatori (elementi finali di ciascuna sequenza) in cui m_i è l'elemento di S in posizione $i*n/2^{1/x}$. Fissiamo $k=2^{1/x}$ tappo della ricorsione.

```
Procedure erewsort(S);
begin
      if |S|≤k then
            sequentialsort(S);
      else
            for i:=1 to k-1 do
                  parallelselect(S,i*|S|/2^(1/x));
      costruisci:
            - S1={s in S / s≤m1}
            - for i:=2 to k-1 do
                  Si={s in S / mi-1≤s≤mi}
            - Sk={s in S / s≥mk-1}
      utilizzando algoritmi simili a quelli usati per generare L, E e
      G.
      for i:=1 to k/2 in parallel do
            erewsort(Si);
      for i:=k/2+1 to k in parallel do
            erewsort(Si);
end;
```

Gli ultimi due `for` vengono utilizzati per evitare che con un `for` unico si cerchi di utilizzare più processori di quanti se ne hanno a disposizione.
La relazione di ricorrenza è:

$$T(n)=c_1*n^x+c_2*\log(n^{1-x}=2^{1/x})+2T(n/2^{1/k})=O(n^x\log(n))$$

in cui i contributi sono dovuti rispettivamente alla parallelselect, al broadcast degli S_i e alle due chiamate ricorsive. Dato il numero dei processori utilizzati, il costo risulta ottimo.

2.5 Tour di Eulero

La tecnica del Tour di Eulero è stata introdotta da Viskin e Tarjan.

Definizione: *si dice <u>circuito euleriano</u> un circuito di un grafo che a partire da un nodo iniziale percorre tutti gli archi una ed una sola volta.*

Supponiamo di avere un albero T=(V,E) e costruiamo a partire da questo un grafo T'=(V,E') in cui ad ogni arco (u,v) di T nel grafo T' associamo due archi orientati (u,v)

e (v,u). T' verificherà la proprietà che per ogni nodo il numero di archi entranti è uguale al numero di archi uscenti; un grafo come T' ovviamente conterrà almeno un circuito euleriano. Vogliamo trovarne uno nel grafo T' specificando una funzione "successore" che mappa ogni arco e di E' nell'arco successivo all'interno del circuito euleriano.
Definiamo $s(e)=s(<u_i,v>)=<v,u_{(i+1) \bmod d}>$, $0 \le i \le d-1$, dove $adj(v)=\{u_0,\ldots,u_{d-1}\}$ è l'insieme dei nodi adiacenti a v per il quale è stato scelto un ordinamento arbitrario; come ad esempio nel grafo di figura 2.17:

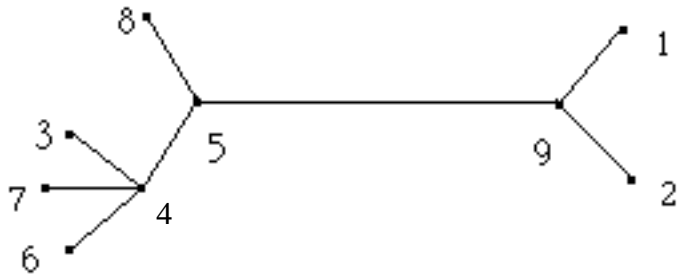


Figura 2.17

abbiamo una funzione successore rappresentata dalla seguente tabella:

e	s(e)
<9,1>	<1,9>
<9,2>	<2,9>
<4,3>	<3,4>
<5,4>	<4,3>
<3,4>	<4,7>
<7,4>	<4,6>
<6,4>	<4,5>
<8,5>	<5,4>
<4,5>	<5,9>
<9,5>	<5,8>
<4,6>	<6,4>
<4,7>	<7,4>
<5,8>	<8,5>
<5,9>	<9,2>
<2,9>	<9,1>
<1,9>	<9,5>

Lemma: *dato l'albero T ed un ordinamento dei vertici adiacenti ad ogni vertice v appartenente a V, la funzione s definita in tabella, determina un circuito euleriano nel grafo T'.*
Dimostrazione: verifichiamo prima che il successore di un arco sia univocamente determinato. Tale affermazione è evidente dalla definizione di successore la quale si basa sull'ordinamento dei nodi adiacenti ad un nodo dato. Dimostriamo quindi che attraverso la funzione successore ci muoviamo su di un circuito euleriano. Utilizziamo l'induzione sul numero n di vertici. Per n=0 o n=1 è banale, per n=2 la situazione è quella illustrata in figura 2.18.

Figura 2.18

Supponiamo il teorema vero per n-1 e dimostriamo che resta vero per n. Consideriamo in T', ricavato da T, l'(n-1)-esimo nodo u a cui ora aggiungiamo un nodo v (l'n-esimo nodo) tramite una coppia di archi orientati (figura 2.19).

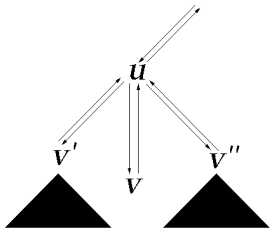


Figura 2.19

Poiché adj(v)={u} allora s(<u,v>)=<v,u>. Adj(u) sarà ora {…,v',v,v",…} e quindi s(<v',u>)=<u,v> e s(<v,u>)=<u,v"> il che implica che abbiamo spezzato il ciclo in T' aggiungendo un doppio arco e ottenendo quindi un nuovo circuito euleriano.

c.v.d.

2.6 Strutture dati per il tour di Eulero
Rappresentiamo T' tramite liste di adiacenza circolari e correlate come in figura 2.20:

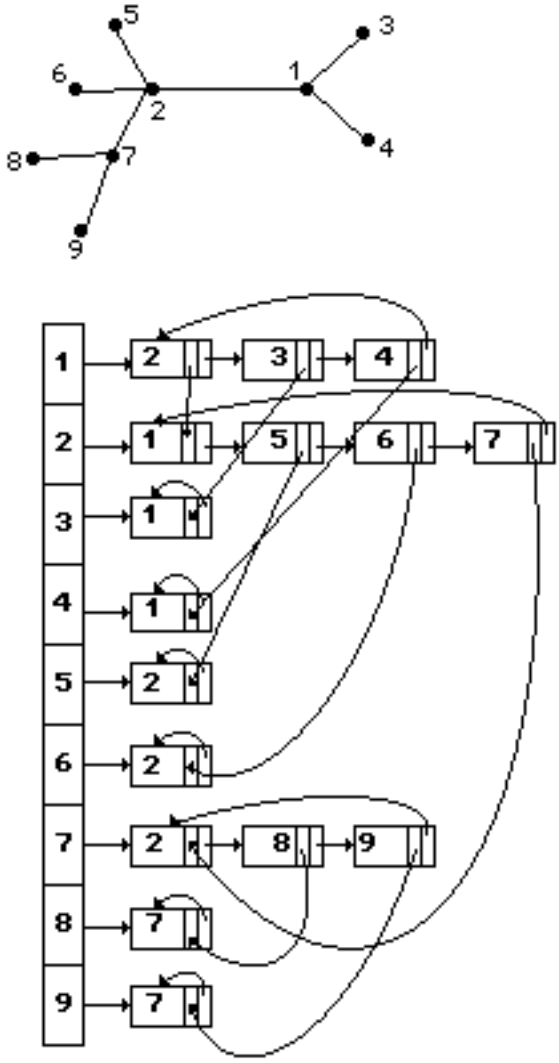


Figura 2.20

È sufficiente un modello EREW con n processori nel quale a ciascun processore associamo un arco ed il suo speculare. Ciò permette di costruire il circuito di Eulero con una complessità O(1) ed un costo C(n).

2.7 Applicazioni del tour di Eulero
Partiamo da un albero non radicato T di cui costruiamo il grafo euleriano T'. Per radicare l'albero nel nodo r, costruiamo un circuito euleriano a partire da r, orientiamo gli archi seguendo la sequenza dei nodi toccati, senza considerare gli archi di ritorno a nodi già visitati. Spezziamo quindi il circuito abolendo l'arco che torna in r.

Nota: *se analizziamo l'orientamento dato, vediamo che esso segue una visita di tipo DFS, ma ciò non vuol dire che abbiamo realizzato una DFS in parallelo che è anzi uno di quei problemi che restano inerentemente seriali.*

L'utilizzo di tale tecnica, combinata con la procedura delle somme parziali ci permette di ottenere in tempo logaritmico le seguenti operazioni:

- numerazione in preordine;
- numerazione in postordine;
- calcolo del livello di ogni vertice;
- numero dei discendenti.

Per realizzare le prime due procediamo come segue:

1) assegniamo ad ogni arco un peso secondo la seguente tabella:

	Preordine	Postordine
Peso 0	<v,padre(v)>	<padre(v),v>
Peso 1	<padre(v),v>	<v,padre(v)>

per evitare accessi concorrenti in memoria, ad ogni processore viene associato lo stesso arco ascendente e discendente.

2) operiamo con la procedura delle somme prefisse applicata ai pesi sulla lista degli archi relativi al tour di Eulero;

3) per ogni nodo v≠r la posizione di v nella postvisita (previsita) è uguale alla somma prefissa relativa all'arco <v,padre(v)> (<padre(v),v>); per v=r la posizione è n (1).

La complessità O(log(n)) è dovuta solamente alla valutazione delle somme prefisse.

Per quanto riguarda il calcolo del livello, supponiamo che il livello della radice sia 0. I pesi assegnati sono +1 per ogni arco discendente e -1 per ogni arco ascendente; l'informazione relativa al nodo v è la somma prefissa relativa all'arco <padre(v),v>. La complessità resta O(log(n)).

Il numero di discendenti rispetto ad un nodo v si calcola assegnando +1 ad ogni arco discendente e 0 ad ogni arco ascendente. Anche in questo caso la complessità è dominata dal calcolo delle somme prefisse.

2.8 Tecnica del rake

L'operazione di rake si applica ad alberi binari i cui nodi interni (ossia che non sono foglie) hanno esattamente due figli. Essa consiste nell'eliminare una foglia ed il padre, che viene sostituito dal fratello della foglia eliminata, mantenendo così la proprietà che ciascun nodo abbia zero o due figli.
L'esigenza di parallelizzare al massimo gli algoritmi ci obbliga ad analizzare in base al modello di macchina su cui si opera il maggior numero di nodi sul quale è possibile effettuare il rake contemporaneamente.

Supponiamo di voler operare in parallelo su tutte le foglie; possiamo incappare in due tipi di problemi; il primo (figura 2.21) è risolvibile utilizzando un modello CRCW in quanto si tratta di un accesso concorrente in lettura e scrittura, il secondo (figura 2.22) non si può risolvere con tecniche analoghe in quanto contiene una contraddizione logica (si cerca di utilizzare come radice di un sottoalbero un nodo cancellato).

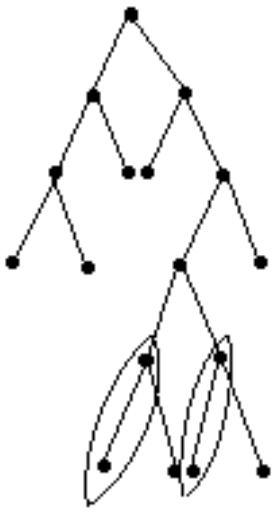


Figura 2.21

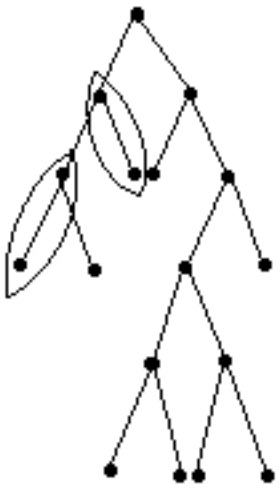


Figura 2.22

Non potendo quindi operare indiscriminatamente su tutte le foglie, studiamo quali limitazioni occorrono per utilizzare l'algoritmo su di un modello EREW.
Vogliamo arrivare ad un albero costruito da uno dato, mantenendo solo la radice e le foglie estremali (la più a destra come figlio destro e la più a sinistra come figlio sinistro).

1) Enumeriamo tutte le foglie in ordine crescente da sinistra verso destra e i rimanenti nodi dell'albero in maniera casuale (enumerazione effettuabile utilizzando la tecnica del tour di Eulero). Utilizziamo un vettore di appoggio

in cui vengono memorizzati in maniera ordinata gli indici di tutte le foglie eccetto quelle estremali.

2)
2.1) Applichiamo il rake a tutte le foglie di posizione dispari (rispetto alla posizione occupata nel vettore) che sono figli sinistri cancellando i dati relativi alle loro posizioni all'interno del vettore (lasciando le posizioni vuote).
2.2) Applichiamo il rake al resto delle foglie in posizione dispari ripetendo l'eliminazione delle posizioni e, successivamente, eliminando tutte le caselle vuote, dimezzando il vettore.

Poiché ad ogni esecuzione del passo 2 il vettore si dimezza, la struttura dell'algoritmo sarà la seguente:

```
for log2(n+1) volte do
     begin
          2.1;
          2.2;
     end;
```

Poiché la complessità, sia del passo 1 che dell'iterazione del passo 2 su modello EREW, è logaritmica, anche l'intero algoritmo è di complessità logaritmica.

2.8.1 Applicazione del rake alla valutazione delle espressioni

Analizziamo l'algoritmo seriale: consideriamo un'espressione qualunque ad esempio 3+2*5-1. Al fine di rendere inequivocabile il significato dell'espressione, utilizziamo una forma ridondante parentesizzata detta ***forma semplice*** che si ottiene applicando le regole di riscrittura descritte nella tabella seguente a partire dall'espressione iniziale racchiusa globalmente da parentesi:

Regole per ottenere la forma semplice a partire dalla forma normale	
Forma normale	**Forma semplice**
(	**(((**
)	**)))**
+	**))+((**
-	**))-((**
*****	**)*(**
/	**)/(**

Nel nostro caso l'espressione in forma semplice diventa (((3))+((2)*(5))-((1))).
Occorre verificare che l'espressione così ottenuta sia corretta, ossia le parentesi siano correttamente bilanciate; a tal scopo si scandisce la stringa utilizzando un contatore incrementato ad ogni apertura di parentesi e decrementato ad ogni chiusura: il contatore dovrà raggiungere il valore 0 senza assumere mai valori negativi. Tali operazioni nel seriale hanno complessità lineare.
Al fine di valutare l'espressione, facciamo uso di una pila di appoggio nella quale duplichiamo le informazioni scandendo l'espressione ed operando come segue:
estraiamo dalla pila una parentesi aperta per ogni parentesi chiusa incontrata nella stringa e sostituiamo il risultato dell'operazione ad ogni occorrenza di una sequenza del tipo "operando", "operatore", "operando". Poiché anche tale operazione nel seriale ha complessità lineare, la complessità di tutto l'algoritmo nel seriale è lineare.
L'estensione al parallelo dell'algoritmo opera su di un albero E-Tree, albero delle espressioni, costruito dalla rappresentazione in forma semplice (figura 2.23):

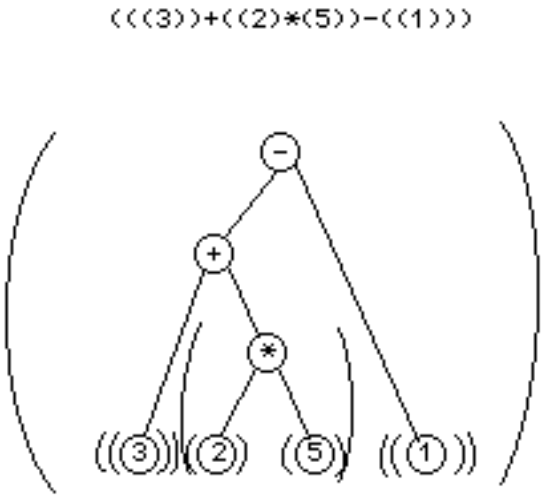


Figura 2.23

L'albero ottenuto è binario in quanto sono binari gli operatori utilizzati (+ e *), ed ogni nodo ha 0 o 2 figli per lo stesso motivo. Essendo la struttura dell'albero strettamente correlata alla parentesizzazione, dobbiamo, per ogni parentesi, conoscere la gemella. Supponiamo di avere una funzione match(i), che implementeremo in seguito, che ci fornisce la gemella di una parentesi; supponiamo inoltre di avere tanti processori quanti sono i caratteri della stringa.
Costruiamo l'albero lavorando sui processori relativi alle parentesi chiuse nella stringa risalendo dalle foglie alla radice dei vari sottoalberi, tenendo conto delle seguenti regole di costruzione: preso un generico processore associato ad un elemento nella lista.

1) Se il processore a sinistra è associato ad una costante, stiamo considerando una foglia. Tale processore dovrà, mediante match, determinare il processore associato alla parentesi gemella e avvertirlo di non fare nessun controllo.

2) Se il processore a sinistra è associato ad un'altra parentesi chiusa, stiamo considerando il sottoalbero che ha come radice il primo operatore alla sinistra della gemella aperta, fatta sempre eccezione per i processori ai bordi.
3) I processori associati agli operatori nella stringa devono individuare gli operandi (figlio sinistro e figlio destro nell'albero), che potranno essere costanti (foglie) o espressioni (sottoalberi).

Il costo di tale serie di operazioni è pari al costo dell'operatore match.
Per quel che riguarda la parallelizzazione dell'algoritmo per la verifica della esattezza della parentesizzazione consideriamo quanto segue.

Definizione: *Si dice sequenza irriducibile di parentesi una sequenza costituita da un certo numero i di parentesi chiuse seguite da un numero k≠i di parentesi aperte, e si indica*

$$)^i\,(^k$$

Introduciamo un operatore ® che, date due sequenze irriducibili, opera nel modo seguente:

$$)^i(^h ®)^k(^l = \begin{cases} k \geq h \Rightarrow)^{i+k-h}(^l \\ k < h \Rightarrow)^i(^{h-k+l} \end{cases}$$

ossia elide le parentesi di troppo a due a due e adopera un'albero con tante foglie quanti sono gli elementi nella stringa corrispondenti a parentesi; applicando l'operatore definito a partire dal basso verso l'alto via via sino alla radice (figura 2.24).

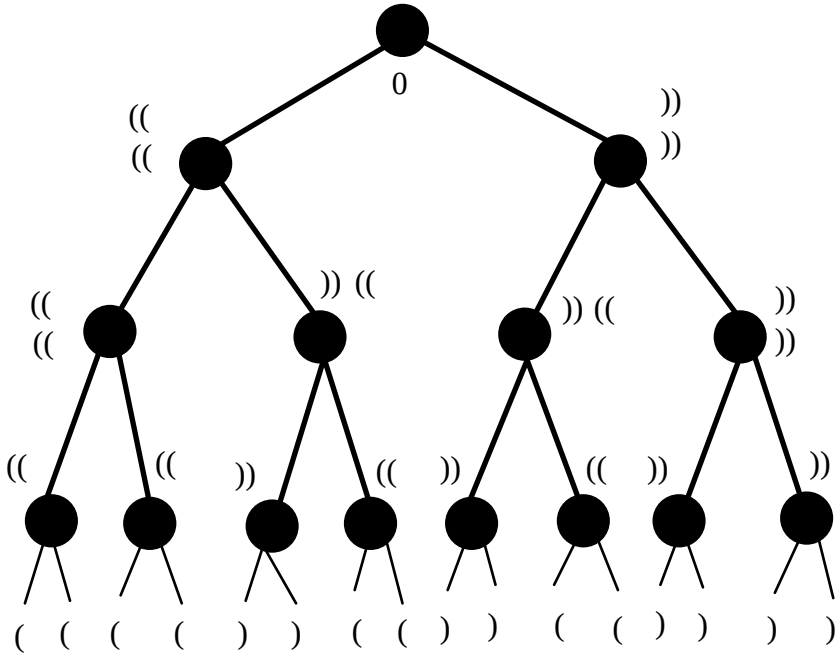


Figura 2.24

La parentesizzazione risulterà corretta solo se, arrivati alla radice, l'operatore fornito darà risultato nullo. La complessità di questa fase è logaritmica nel numero di parentesi.
L'algoritmo per la valutazione opera sull'E-TREE mediante l'operazione di rake contraendo l'albero, operando sui nodi interni che hanno almeno un figlio costante, in modo da ritrovare il risultato finale nella radice.
Un approccio intuitivo per la valutazione parallela dell'albero delle espressioni consiste nel valutare ogni sottoespressione di un nodo v i cui due figli sono foglie, parallelamente per ognuno di questi, ripetendo il processo fino a che rimarrà determinato il valore finale della espressione nella radice. Associamo ad ogni nodo v una coppia di etichette (a_v,b_v) attraverso le quali costruiamo l'espressione lineare a_v*X+b_v ove a_v e b_v sono costanti e X un indeterminata rappresentante il possibile valore della subespressione associata al sottoalbero con radice il nodo v. L'idea è quella di contrarre l'albero eliminandone i nodi, tenendo memoria delle loro informazioni e modificando opportunamente le etichette associate ad ogni nodo restante. La regola generale su cui opera l'algoritmo è basata sulla seguente proprietà: sia u un nodo interno all'expression tree al passo corrente, contenente l'operatore op_u appartenente all'insieme {+,*}, ed avente figli i nodi v e w con etichette rispettivamente (a_v,b_v), (a_w,b_w); allora il valore della subespressione di u sarà dato da:

$$val(u)=(a_v*val(v)+b_v)op_u(a_w*val(w)+b_w).$$

Inizialmente assegnamo alle etichette di ogni nodo i valori (1,0) in modo da non alterare il valore iniziale delle foglie e utilizziamo l'algoritmo di rake per contrarre l'albero (figura 2.25).

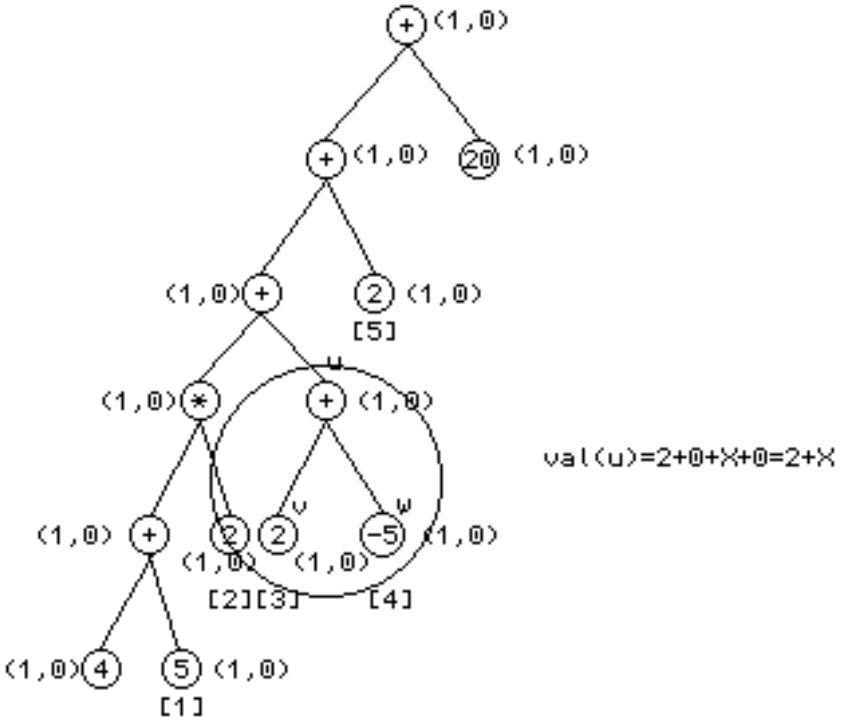


Figura 2.25

Come descritto nell'algoritmo del rake operiamo in parallelo: prima sulle foglie di numerazione dispari che sono figli sinistri, per ognuna delle quali valutiamo val(u) (figura 2.25) e aggiorniamo le etichette relative al nodo u (figura 2.26),

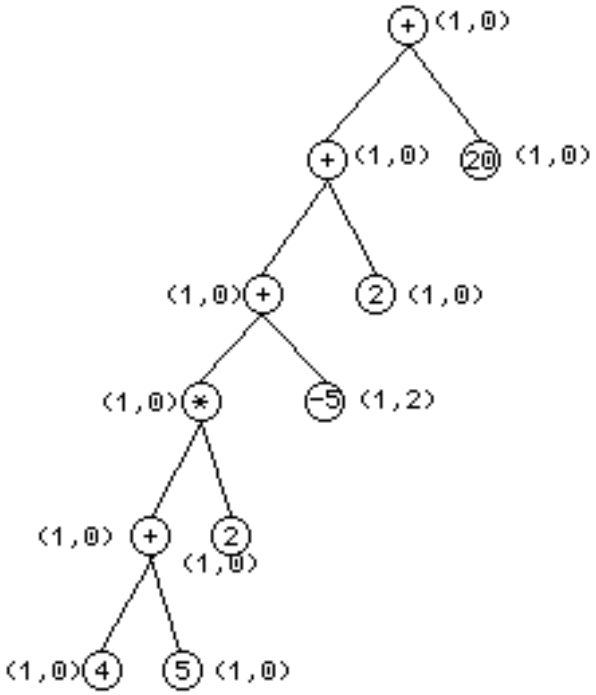


Figura 2.26

poi operiamo sulle restanti di numerazione dispari come in precedenza (figura 2.27 e 2.28),

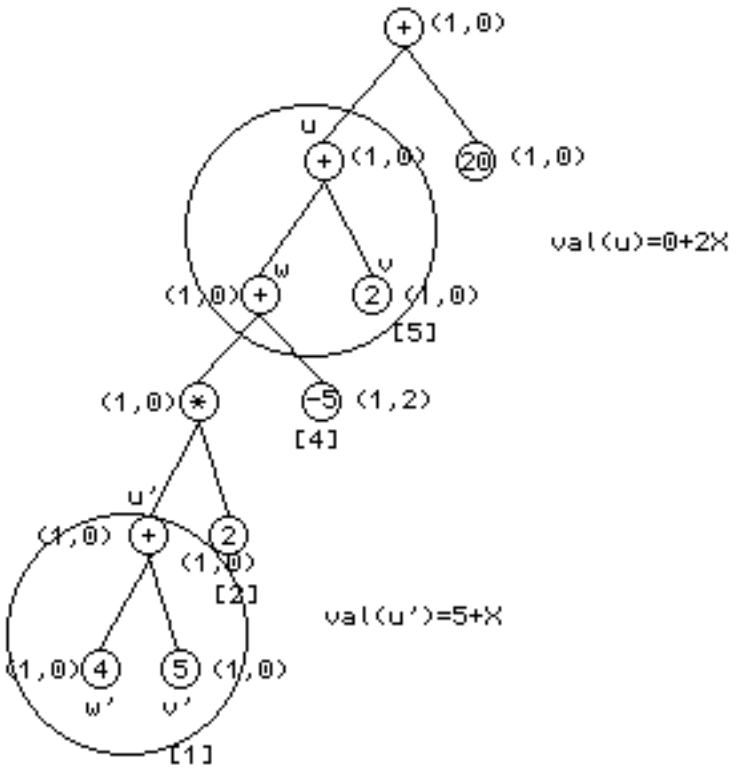


Figura 2.27

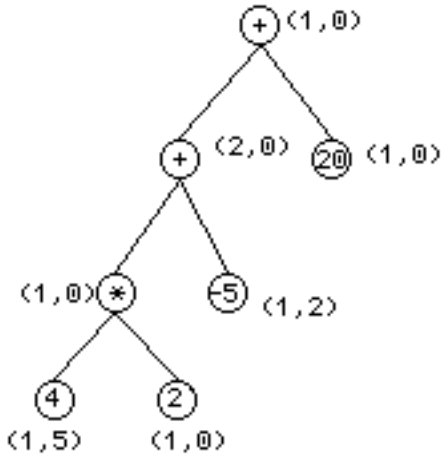


Figura 2.28

infine sulle foglie rimanenti (figura 2.29 e 2.30).

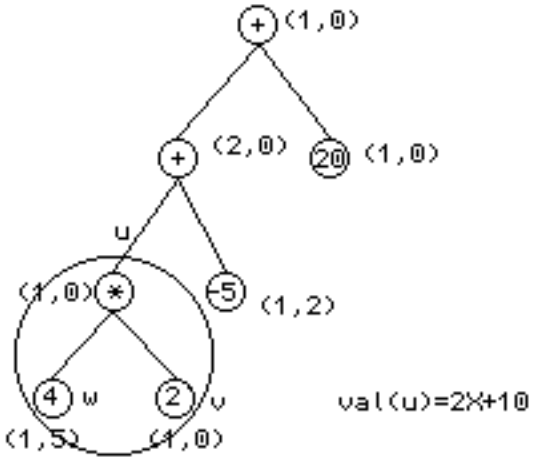


Figura 2.29

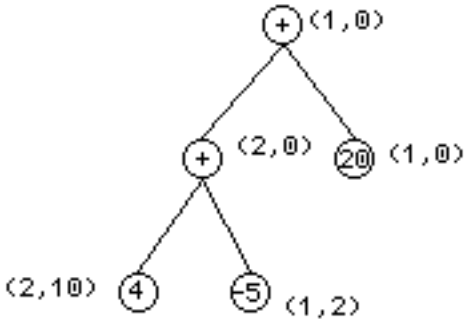


Figura 2.30

Ripetendo tali operazioni come specificato nell'algoritmo di rake otteniamo per passi successivi i due alberi di figura 2.31 e 2.32:

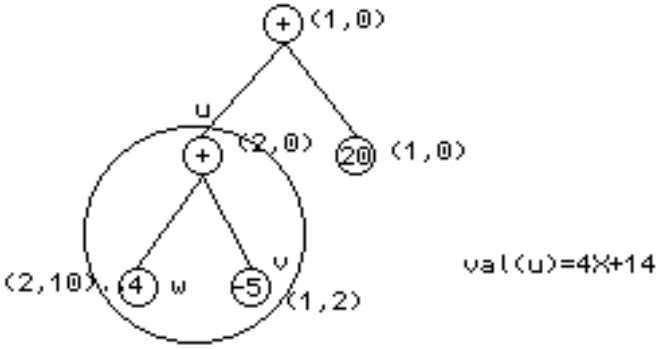


Figura 2.31

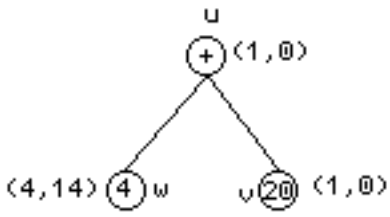


Figura 2.32

La valutazione di val(u) secondo la regola data ci fornisce quindi il valore dell'espressione nella radice dell'expression tree che sarà ottenibile secondo la regola fornita come $val(u)=(a_w*c_w+b_w)op_u(a_v*c_v+b_v)=[(4*4+14)+(20*1+0)]=50$, ove c_w e c_v rappresentano rispettivamente i valori dei due nodi v e w, valori da assegnare all'indeterminata X ora nota. L'algoritmo di valutazione delle espressioni algebriche fornito consiste quindi in un assegnamento iniziale delle etichette (1,0) ad ogni nodo dell'albero e nella applicazione dell'algoritmo di contrazione mediante rake, richiedendo quindi un tempo logaritmico nei nodi dell'expression tree.

2.8.2 Match

Al fine di completare la trattazione del problema della valutazione delle espressioni, andiamo ad analizzare l'algoritmo che realizza l'operatore ***match***.
Tale operatore, prendendo in input una parentesi aperta, ne trova la gemella operando in due fasi distinte: una fase in cui scandisce l'albero delle parentesi dal basso verso l'alto (fase bottom-up) ed una fase in cui scandisce tale albero dall'alto verso il basso (fase top-down).
Sia s il nodo attuale e sia z una stringa inizializzata a stringa vuota:

fase bottom-up

```
If s è figlio destro
      then ci spostiamo sul padre
      else if z ® (valore del fratello destro di s) inizia
      con ")"
            then ci spostiamo sul fratello destro e iniziamo
                  la fase top-down
            else begin
                    z:=z ® (valore del fratello destro di s);
                    ci spostiamo sul padre;
                  end;
```

tale procedura è contenuta all'interno di un ciclo che termina con l'inizio della fase top-down.

fase top-down

```
if z ® (valore del figlio sinistro di s) comincia con ")"
      then ci spostiamo sul figlio sinistro
      else begin
              z:= z ® (valore del figlio sinistro di s);
              ci spostiamo sul figlio destro;
            end;
```

La complessità di tale operatore è quindi logaritmica nel numero di parentesi della stringa.
Alcune complicazioni possono insorgere se modifichiamo il numero di processori. La situazione attuale ci ha permesso di determinare un algoritmo in grado di risolvere il problema della ricerca della parentesi gemella con complessità logaritmica, avendo un numero di processori sufficenti a coprire tutte le parentesi contenute nella stringa da valutare.
Supponiamo che k sia il numero di parentesi e che k/log(k) sia il numero di processori a nostra disposizione; applicando il teorema di Brent, la riscrittura dell'algoritmo basata sull'attuale numero di processori fornirebbe una complessità dell'ordine del logaritmo al quadrato.
Esiste però un'algoritmo, di cui daremo solamente un'idea, che ci permette di mantenere la stessa complessità: suddividiamo la stringa di parentesi in blocchi di log(k) elementi, quindi:

1) in ogni blocco eliminiamo tutte le coppie parentesi aperta-chiusa;
2) marchiamo la prima parentesi aperta e l'ultima parentesi chiusa all'interno di ogni blocco;

2.1) per ogni parentesi marcata, determiniamo la gemella e la marchiamo.

2.9 Operazioni su matrici

2.9.1 Matrice trasposta

Data una matrice quadrata di dimensione n l'algoritmo seriale ottimo di trasposizione ha complessità $O(n^2)$. Trasportiamo l'algoritmo nel parallelo su di una macchina a rete di interconnessione matriciale, supponendo di avere tanti processori quanti sono gli elementi della matrice. L'idea dell'algoritmo si basa sulle seguenti considerazioni:

- poiché gli elementi sulla diagonale principale non devono essere toccati durante la trasposizione, tali dati restano fermi;
- gli elementi sotto la diagonale vengono spediti ad occupare la posizione simmetrica sopra la diagonale e, simultaneamente, gli elementi sopra le diagonali vengono spediti ad occupare le posizioni simmetriche sotto la diagonale secondo lo schema di figura 2.33:

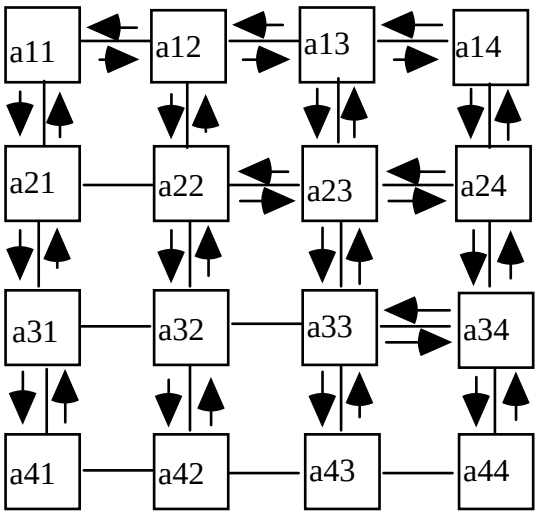


Figura 2.33

Supponiamo che ogni processore abbia tre registri: uno per un flusso orizzontale di dati, uno per un flusso verticale e uno per il dato definitivo. Lo schema dell'algoritmo sarà il seguente:

1) parallelamente tutte le informazioni della parte triangolare inferiore e superiore prendono posto nei rispettivi registri e vengono accompagnati da un campo in cui è presente il nome del processore da cui è partito il dato, eseguendo un passo del flusso;
2) ogni processore analizza l'informazione iniziale e quelle in arrivo;
   2.1) tutti i processori sulla diagonale confermano il verso di percorrenza del dato in arrivo verso il processore di indici trasposti rispetto agli indici del processore che ha inviato il dato;

2.2) gli altri processori verificano mediante confronto degli opportuni registri l'informazione in transito: se l'informazione non è quella aspettata ne confermano il verso di percorrenza.

Nonostante il modello di macchina utilizzato possa sembrare il più consono alla manipolazione di una tale struttura dati, in realtà si dimostra essere assai inefficente rispetto a un modello di macchina a memoria condivisa.
Supponiamo di utilizzare un tale modello di macchina con un numero di processori dell'ordine $O(n^2)$. Associamo ad ogni elemento della matrice che appartenga alla triangolare inferiore ed al proprio trasposto nella triangolare superiore un processore che provvederà, mediante memoria condivisa, alla trasposizione di tali elementi.
La complessità totale si riduce quindi a O(1), complessità che risulta ottima. Considerate inoltre le caratteristiche di tale algoritmo, esso sarà trasportabile anche su un modello EREW con uguale complessità.

2.9.2 Prodotto tra matrici

L'algoritmo seriale di prodotto righe per colonne tra due matrici (m*n) e (n*k), nella sua forma basata sulla definizione matematica di tale operazione, ha complessità dell'ordine $O(h^3)$, dove $h=\max\{m,n,k\}$.
Il limite inferiore alla complessità di tale algoritmo è stato dimostrato essere $O(h^2)$, esistono degli algoritmi seriali (Algoritmo di Strassen, il quale lavora con matrici h*h ed ha una complessità di $O(h^{2,37})$) che si avvicinano a tale limite, ma non è stato ancora trovato alcun algoritmo di complessità ottima.
Supponiamo di lavorare su un modello di macchina a rete di interconnessione matriciale con un numero di processori pari al numero di elementi della matrice finale (m*k). L'idea dell'algoritmo è la seguente:
siano A e B le due matrici in input, associamo ad ogni elemento della matrice finale C un processore nella cui memoria locale verrà riservata una cella per contenere il risultato finale $c_{i,j}$ inizializzata a 0.

- Per ciascuna matrice da moltiplicare, impacchettiamo rispettivamente le righe e le colonne mettendole in input ai processori della prima riga e prima colonna secondo lo schema di figura 2.34:

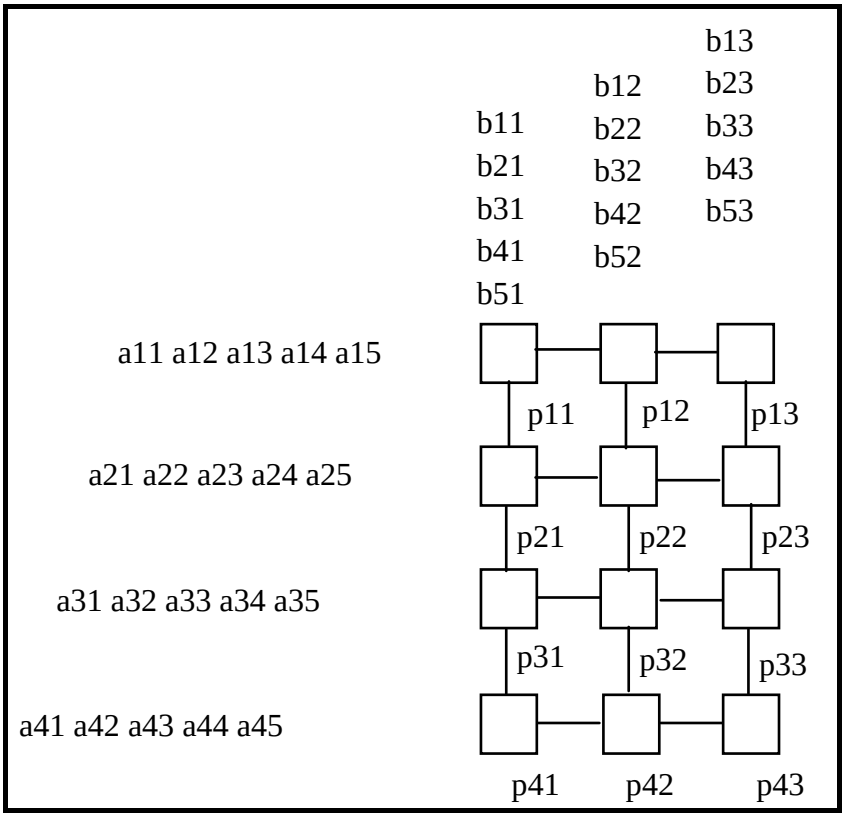


Figura 2.34

La riga i della matrice A arriva una unità di tempo dopo la riga i-1, e la colonna j della matrice B arriva una unità di tempo dopo la colonna j-1 in modo che a_{ik} incontra b_{kj} nel processore $p_{i,j}$ al momento giusto.

-Il processore $p_{i,j}$ riceve i singoli elementi dei due ingressi nel giusto sincronismo operando nel modo seguente:

- fino a che il processore $p_{i,j}$ riceve i due ingressi $a_{i,k}$, $b_{k,j}$:
  - moltiplica i due ingressi;
  - somma il risultato al contenuto della memoria locale;
  - se $i<m$ spedisce $b_{k,j}$ a $p_{i+1,j}$;
  - se $j<k$ spedisce $a_{i,k}$ a $p_{i,j+1}$.

Il valore della memoria locale di ciascun processore ad ogni passo per due matrici A e B di dimensioni (2*2) è evidenziato nella figura 2.35:

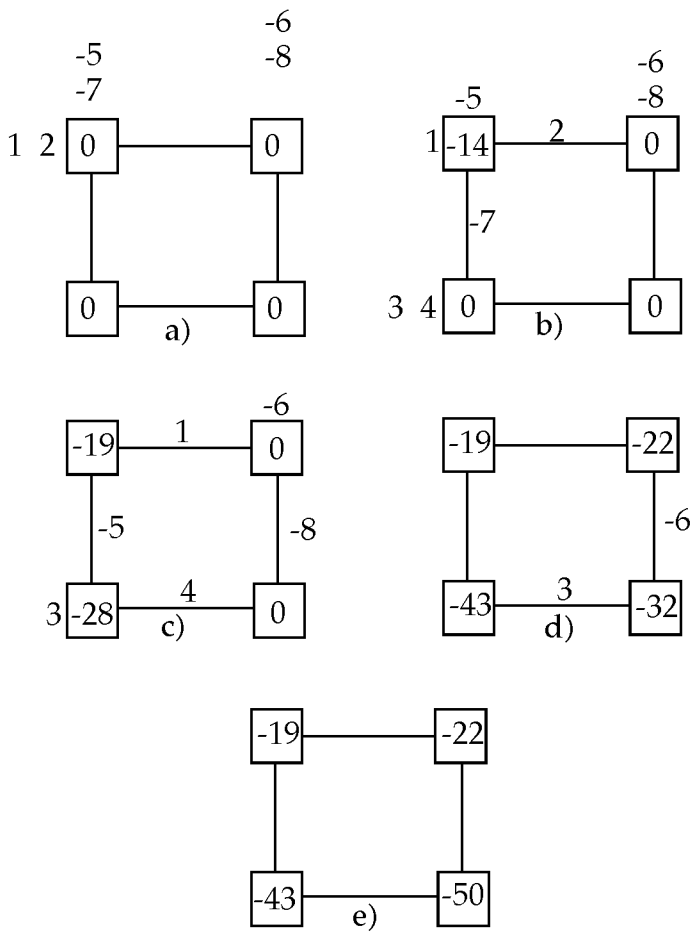


figura 2.35

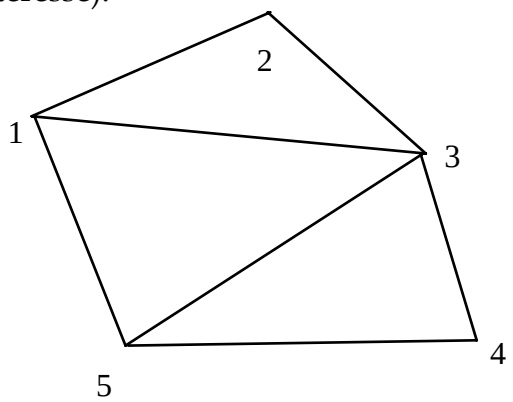
Con un numero di processori dell'ordine $O(h^2)$ l'algoritmo ha complessità lineare $O(h)$, tempo necessario ai dati per propagarsi lungo la rete e un costo $O(h^3)$. Sebbene il modello di macchina utilizzato possa sembrare il più appropriato a questo tipo di operazione, in realtà il costo può essere migliorato lavorando su un modello CRCW a memoria condivisa con un numero di processori dell'ordine $O(h^3)$, trasportando l'algoritmo dal seriale al parallelo con costo ottimo $O(h)$.

```
for i:=1 to m in parallel do
  for j:=1 to n in parallel do
     for t:=1 to k in parallel do
     begin
          ci,j:=ci,j+ci,k*ck,j;
     end;
```

2.9.2.1 Matrice chiusura transitiva di un grafo

Si tratta di un'applicazione dell'operazione precedente.
Dato un grafo G non orientato memorizzato mediante la sua matrice di adiacenza A_G (figura 2.36) vogliamo generare la matrice di chiusura transitiva di G, C_G avente valore 1 in $c_{i,j}$ se e solo se esiste un cammino di qualche lunghezza che connetta il

vertice v_i al vertice v_j (nel caso di grafi orientati, la complessità del problema è tale da renderlo di scarso interesse).



	1	2	3	4	5
1	**1**	**1**	**1**	**0**	**1**
2	**1**	**1**	**1**	**0**	**0**
3	**1**	**1**	**1**	**1**	**1**
4	**0**	**0**	**1**	**1**	**1**
5	**1**	**0**	**1**	**1**	**1**

Figura 2.36

Si consideri inizialmente la matrice di tutti i cammini di lunghezza unitaria di G, C_{G1} ottenuta da A_G aggiungendo tutti 1 sulla diagonale principale, vogliamo arrivare alla matrice di tutti i cammini di lunghezza al più n-1 costruendo una successione di matrici di tipo C_{Gi} AND $C_{Gi}=C_{G2i}$ contenente tutti i cammini di lunghezza al più i+1 nel modo seguente:

$$C_{G1} \text{ AND } C_{G1} = C_{G2}$$
$$C_{G2} \text{ AND } C_{G2} = C_{G4}$$
$$C_{G4} \text{ AND } C_{G4} = C_{G8}$$
$$\ldots$$
$$C_{Gm} \text{ AND } C_{Gm} = C_{G2m}$$

tenendo presente che C_{Gm} AND $C_{Gm}=C_{Gn-1}$ per ogni m>n-1.

La successione di matrici siffatta rappresenta cammini di lunghezza crescente secondo le potenze di due e si ferma al primo k=n-1 necessitando di $\log_2(n-1)$ prodotti di matrici.

La complessità dell'algoritmo ed il numero di processori sono quindi misure legate al prodotto tra matrici, quindi dipendenti dalla macchina e dall'algoritmo utilizzati.

2.9.3 Prodotto tra matrice e vettore

Vogliamo eseguire un prodotto tra una matrice con m righe ed n colonne ed un vettore (ovviamente di dimensione m). Utilizziamo una SIMD ad albero binario completo. Si tratta di un'applicazione della tecnica del pipeling. Carichiamo sulle foglie a tempi cadenzati prima il vettore e poi le righe della matrice (una per volta) in modo che ad ogni foglia complessivamente forniamo un'intera colonna della matrice. All'i–esimo passo nella foglia j eseguiamo il prodotto $a_{ij}u_j$. Lo schema è il seguente:

all'istante t_1, a_{1j} viene caricato nel processore foglia P_j (ove già risiede u_j) e viene eseguito il prodotto $a_{1j}u_j$ parallelamente su ogni foglia. Forniamo questo prodotto al livello superiore il cui processore farà (all'istante t_2) la somma delle informazioni fornite dai figli; all'istante t_2, a_{2j} viene caricato nel processore foglia P_j e viene eseguito il prodotto $a_{2j}u_j$ parallelamente su ogni foglia. Contemporaneamente ai livelli superiori avvengono le somme dei prodotti; all'istante $t_{\log2(m)}$, nella radice abbiamo ottenuto $\sum a_{1j}u_j=v_1$ (j=1,…,n).
L'ultimo output viene ottenuto dopo n passi a partire dall'istante $t_{\log2(m)}$; in questo modo la complessità risulta O(log(m)+n).

2.10 Componenti connesse

Considereremo due algoritmi paralleli efficienti per l'identificazione delle componenti connesse di un grafo.
Sia G=(V,E) un grafo non orientato, con |V|=n e |E|=m. Due vertici u e v di V si dicono connessi, se u=v o esiste un cammino $C=(u=x_1,x_2,\ldots,x_k=v)$ tale che l'arco (x_i, x_{i+1}) appartiene ad E, $1\leq i\leq k-1$. Quest'ultima è una relazione di equivalenza su V che partiziona V stesso in classi di equivalenza $\{V_j\}$ (j=1,…,t). I sottografi $G_j=(V_j, E_j)$, con $E_j=\{(x,y)$ di E | x,y appartengono a $V_j\}$, sono le componenti connesse del grafo G.
La matrice di adiacenza C(n*n) è definita come:

$$C(i,j)=\begin{cases} 1 \text{ se esiste in E l'arco } (v_i,v_j) \\ 0 \text{ altrimenti.} \end{cases} \quad (1\leq i,j\leq n)$$

Un algoritmo sequenziale per la risoluzione del nostro problema consiste in una semplice visita o di tipo Breadth-First-Search oppure di tipo Depth-First-Search; in entrambi i casi la complessità è O(m+n).
Poiché la visita di tipo DFS è, come abbiamo già osservato, inerentemente sequenziale, mentre la BFS può essere effettuata in $\log^2(n)$ passi con $O(n^{2,376})$ processori su di una CREW, se pensassimo di parallelizzare uno di questi algoritmi otterremmo solamente un algoritmo molto complicato e di complessità elevata. Dobbiamo cercare perciò un approccio differente al problema, che sfrutti altre proprietà correlate alla struttura dei grafi.
Iniziamo col dare la definizione di pseudoforesta.
Una pseudoforesta è un grafo orientato in cui ogni vertice ha un grado minore od uguale ad uno; equivalentemente si può dire che, per ogni v appartenente a V, outdegree(v)≤1. Una pseudoforesta è allo stesso modo definita quando sia definita una funzione D:V—>V e l'insieme F={(v, D(v)) | v appartiene a V}, allora la pseudoforesta è denotata dalla coppia (V,F).
Diamo inoltre la definizione di un tipo particolare di albero, la stella radicata: essa è un albero orientato radicato in cui ogni nodo è direttamente collegato all'unica radice dell'albero.
Torniamo allora all'esame del nostro problema: dato un grafo G non orientato, con n nodi, lo si vuole scomporre nel più piccolo numero di componenti connesse.
A questo scopo, iniziamo col calcolare la matrice chiusura transitiva (o di connettività) C^* del grafo, a partire dal dato di input che è la matrice di adiacenza C del grafo stesso:

$$\hat{C}(i,j)=\begin{cases} 1 \text{ se } v_i \text{ è connesso a } v_j \\ 0 \text{ altrimenti.} \end{cases} \quad (1 \le i,j \le n)$$

Successivamente determiniamo una nuova matrice quadrata D(n*n) così definita:

$$D(i,j)=\begin{cases} v_j \text{ se } \hat{C}(i,j)=1 \\ 0 \text{ altrimenti} \end{cases} \quad (1 \le i,j \le n)$$

in questo modo la matrice D per ogni riga i contiene i nomi dei vertici v_j connessi tramite un cammino di lunghezza maggiore o uguale a zero al vertice v_i.
Poiché da tale matrice risulta che ad ogni riga i gli elementi non nulli sono i vertici appartenenti alla stessa componente connessa di v_i, è possibile individuare il più piccolo numero di componenti connesse di G in questo modo:
il vertice v_i è assegnato alla componente a, se a è il più piccolo indice tale che $d_{ia} \neq 0$, ovvero, fra tutti gli elementi non nulli della riga i, viene scelto quello di numerazione minima.
Per calcolare la matrice C^* operiamo la moltiplicazione tra matrici booleane (sia C che C^* contengono elementi sull'insieme {0,1}). Il procedimento è perfettamente identico a quello della moltiplicazione regolare tra matrici a patto di sostituire l'operazione di moltiplicazione con l'operazione di AND logico e l'operazione di somma con l'operazione di OR logico.
É da notare, infatti, che se C rappresenta tutti i percorsi di lunghezza minore o uguale ad uno allora C^2 rappresenta tutti i percorsi di lunghezza minore o uguale a due, C^4 quelli di lunghezza minore o uguale a quattro e così via fino a percorsi di lunghezza massima, la quale per n nodi vale n-1; quindi $C^*=C^{n-1}$.
La matrice di connettività per il grafo G viene quindi ottenuta in $\log_2(n)$ passi. É possibile scegliere un metodo qualsiasi di moltiplicazione tra matrici pur di adattarlo a matrici booleane, conseguentemente il numero di processori utilizzati dipenderà dal modello di macchina e dal metodo scelti per l'implementazione.
A questo punto abbiamo tutti gli strumenti per introdurre i due algoritmi.
Il primo è ottimo per grafi densi ed ha complessità $O(\log^2(n))$, il secondo, invece, risulta essere ottimo per grafi sparsi ed ha complessità $O((m+n)*\log(n))$.

2.10.1 Algoritmo ottimo per grafi densi

Sia A(n*n) la matrice di adiacenza del grafo non orientato G=(V,E), con V={1,2,…,n} ed |E|=m (m>n), risulterà ovviamente A(i,j)=1 se e solo se (i,j) è un arco di E ($1 \le i,j \le n$).
Si costruisce la funzione C su V in modo tale che:

$$C(v)=\begin{cases} \min\{u \mid A(u,v)=1\} & \text{se } v \text{ è isolato} \\ v & \text{altrimenti.} \end{cases}$$

Con questa funzione si riesce ad identificare le componenti connesse.
Conseguentemente la funzione C(v) definisce una pseudoforesta F che partiziona i vertici di V in $V_1,V_2,\ldots,V_S$.
Inoltre:

1) i vertici di ciascun V_i appartengono alla stessa componente connessa;
2) ogni ciclo in F o è un loop o contiene due archi;

3) in ogni albero T_i (relativo a V_i) il ciclo contiene il vertice di enumerazione minima.

Spieghiamo le precedenti affermazioni:

1) poiché il grafo di partenza è non orientato, l'appartenenza di due vertici allo stesso albero T_i significa l'appartenenza alla stessa componente connessa;

2) in T_i c'è un loop se il nodo è isolato, altrimenti l'unico ciclo in T_i è determinato dal nodo di enumerazione minima r e da un solo altro nodo u ad esso connesso; essendo il grafo non orientato, il ciclo è dato dai due archi (u, r) e (r, u);

3) chiaramente in ogni T_i il ciclo contiene il vertice di enumerazione minima, per quanto visto sopra e tale vertice r è detto radice dell'alberoT_i.

Vediamo come si svolge l'algoritmo:

una volta determinati gli insiemi V_i attraverso l'applicazione della funzione C all'insieme dei vertici V, si accorpa ciascun V_i in singoli vertici a formare dei meganodi. A questo punto la procedura si ripete considerando come grafo l'insieme dei meganodi $\{V_i\}$ e dei megaarchi (V_i,V_j) tali che esiste v appartenente a V_i e w appartenente a V_j con la proprietà che (v,w) appartiene ad E. Una volta accorpati i T_i, i nuovi meganodi vengono rinumerati; l'algoritmo termina quando ogni meganodo (dell'ultima rinumerazione) diventa isolato.

Vediamo più nel dettagio l'operazione di accorpamento.

Per ciascun insieme V_i individuiamo l'elemento r_i di V_i con indice minimo, esso sarà il rappresentante della classe V_i, quindi etichettiamo gli altri vertici di V_i con la stessa etichetta di quello rappresentativo.

Ora per ogni vertice v appartenente a V_i poniamo:

$$C(v)=\begin{cases} C(r) & \text{se } v\neq r_i \\ C(u) & \text{se } v=r_i \end{cases} \text{ (essendo u il vertice che forma il ciclo con } r_i\text{)}$$

Prendendo il minimo tra C(v) e C(C(v)) e ponendolo come radice dell'albero cui appartiene, la pseudoforesta di partenza viene trasformata in una foresta di stelle radicate.

La radice r_i di T_i così trovata, sarà l'elemento rappresentativo per l'insieme V_i e ad esso ci riferiremo come al meganodo r_i.

Siano allora r_i e r_j due meganodi rappresentativi per gli alberi T_i e T_j rispettivamente. Essi sono adiacenti nel grafo dei meganodi se e solo se esiste v appartente a V_i e w appartente a V_j tali che l'arco (v,w) appartiene ad E. In caso di risposta affermativa è possibile accorpare ulteriormente T_i e T_j inserendo l'arco (r_i,r_j) in questo modo:

per ogni arco (x,y) appartenente ad E tale che $C(x)\neq C(y)$, poniamo (C(x),C(y)) come arco nel grafo costituito dai meganodi.

Terminato l'algoritmo (la situazione finale è di soli meganodi isolati), per determinare la funzione D applicata a ciascun vertice, inizialmente poniamo D(r)=r per ogni meganodo r. A questo punto si procede al contrario espandendo ogni radice r nell'albero ad essa corrispondente nella precedente iterazione ed assegnando il valore D(r) a tutti i vertici del suo albero. Si continua ad espandere ogni meganodo fino ad ottenere il grafo di partenza, cosicché al termine di questa fase avremo individuato le componenti connesse.

L'algoritmo è il seguente:

come input abbiamo A(n*n) la matrice di adiacenza del grafo G non orientato; come output il vettore D(n), tale che D(i) contiene il più piccolo vertice nella componente connessa di i.

```
begin
    1. A0:=A; n0:=n; k:=0.
    2. while nk>0 do
        2.1 k:=k+1.
        2.2  C(v):=min{u|Ak-1(u,v)=1,  u≠v},  se  v  è  isolato,
        altrimenti C(v)=v.
        2.3 Accorpa ogni albero della pseudoforesta definita da C
        in una stella radicata, con la tecnica del salto del
        puntatore. La radice di ogni stella contenente più di un
        vertice definisce un nuovo meganodo.
        2.4  Poni  nk  uguale  al  numero  dei  nuovi  meganodi,
        utilizzando l'algoritmo delle somme prefisse, e sia
        Ak(nk*nk)  la  matrice  di  adiacenza  del  grafo  dei  nuovi
        meganodi.
    3. Per ogni vertice v determina D(v) come segue: se alla fine
    del passo due C(v)=v, allora poni D(v)=v, altrimenti inverti
    il processo compiuto al passo due espandendo ogni meganodo r
    (tra i meganodi formati durante una iterazione) nell'insieme Vr
    dei vertici del suo albero orientato e costruisci gli
    assegnamenti D(v)=D(r), per ogni v appartenente a Vr.
end.
```

Passiamo adesso ad un'analisi dell'algoritmo.
Il passo più significativo è il secondo, vediamo le sue parti:

2.2 per il calcolo del minimo elemento ci occorrono $O(\log(n^{k-1}))=O(\log(n))$ passi;

2.3 trasformare un albero in una stella comporta una complessità $O(\log(n))$, utilizzando la tecnica del salto del puntatore;

2.4 per contare il numero di stelle significative su cui costruire il grafo dei meganodi ci avvaliamo dell'algoritmo delle somme prefisse, ciò comporta una complessità $O(\log(n))$; per calcolare A_k (una volta fissato n_k) occorrono solo operazioni costanti.

Notiamo inoltre che per il passo 2.3 possiamo utilizzare un modello di macchina PRAM CREW, mentre, per il passo 2.4 necessitiamo di una PRAM CRCW per l'inserimento dell'arco (C(x),C(y)) nel grafo dei meganodi.
Ma vediamo più nel dettaglio il ciclo while .
Nella fase di accorpamento possiamo pensare di dimezzare, almeno ogni volta, il numero dei meganodi da considerare successivamente; questo ragionamento è ammissibile in quanto, tra le stelle presenti all'iterazione precedente, alcune saranno diventate un meganodo isolato, cioè un loop, siano esse n'_{k-1}, allora tutte le restanti $n_k-n'_{k-1}$ saranno contenute in una delle stelle non banali n_k e risulterà $n_k \leq (n_k-n'_{k-1})/2$, contenendo ogni stella non banale almeno due vertici. Pertanto si può concludere

che $n_k \leq n'_{k-1}/2$ e quindi la complessità del ciclo while è data da $O(\log(n_k))$, mentre la complessità dell'intero algoritmo è maggiorata da $O(\log(n^2))$, per un totale di $O(n^2))$ operazioni.
Per un grafo denso la struttura della matrice di adiacenza non risulta quasi vuota ($m \approx n^2$), altrettanto vale per la struttura delle stelle radicate, e allora è possibile affermare che in questo caso l'algoritmo è ottimo.

2.10.2 Algoritmo ottimo per grafi sparsi

Questo algoritmo sfrutta la caratteristica dei grafi sparsi ($m << n^2$) per abbandonare talune restrizioni presenti nell'algoritmo precedente, come formare stelle radicate ad ogni iterazione per riuscire ad eseguire questa iterazione in tempo costante senza pagare il costo del calcolo della funzione C.
Anche in questo algoritmo la situazione di partenza è una foresta di alberi, in cui i vertici di ciascun albero appartengono alla stessa componente connessa e c'è un ciclo alle radici.
L'idea dell'algoritmo è quella di esaminare, durante ogni iterazione, ciascun arco (u,v) del grafo e, sotto opportune condizioni, fondere i due alberi contenenti i due nodi u e v, se essi sono distinti. Contestualmente alla suddetta iterazione, la tecnica del salto del puntatore viene applicata ad ogni vertice di un albero orientato.
Sia D la funzione su V che definisce una pseudoforesta alla prima iterazione. Inizialmente poniamo D(v)=v per ogni v appartenente a V (è inutile in questo caso partire da una struttura preaccorpata).
Definiamo la seguente operazione:
siano T_i e T_j due alberi distinti di una pseudoforesta definita da D; data la radice r_i di T_i e un vertice v di T_j, l'operazione $D(r_i)=v$ è chiamata innesto di T_i su T_j.
L'applicazione della tecnica del salto del puntatore si svolge in questo modo:
dato un vertice v dell'albero T, si pone D(v)=D(D(v)).
Alla luce di quanto appena detto vediamo un'analisi dei passi, le loro problematiche e la complessità globale dell'algoritmo.
Siano T_i e T_j due alberi della nostra pseudoforesta all'inizio di una iterazione generica. Supponiamo che esista un arco (u,v) di E tale che u appartenga a T_i e v a T_j. Il nostro obiettivo è fondere i due alberi in tempo costante, dopo aver verificato che i loro vertici appartengono alla stessa componente connessa. Operiamo in tal senso se il vertice x soddisfa la condizione D(x)=D(D(x)), che rappresenta la situazione in cui o x è la radice dell'albero o x è direttamente connesso alla radice dell'albero a cui appartiene. Sorge però il problema del caso in cui entrambi u e v soddisfino questa condizione, allora non si saprebbe quale albero innestare sull'altro. Per evitare questa situazione, si sceglie di agganciare l'albero il cui vertice in esame ha la numerazione minima.
Può sorgere anche un altro problema: per il tipo di struttura che abbiamo ora non è più possibile affermare che una stella non sia più accorpabile ad un'altra, ma occorrerà accertarsene. Può capitare che una stella contenga tutti i vertici di numerazione minore e i loro adiacenti, di numerazione più grande, siano in un altro albero ad un livello molto maggiore. L'unico mutamento che può avvenire in questo caso all'interno della pseudoforesta è che in tempo logaritmico il vertice v salga fino a diventare radice del suo albero (tecnica del salto del puntatore). Per ovviare a

quest'ultimo inconveniente, una volta terminato il processo d'innesto iniziale, deve esser fatto un tentativo per innestare una stella con radice r_i su un vertice v, qualora esista l'arco che connette la stella T_i a v. Quindi una volta applicata la tecnica del salto del puntatore ad ogni vertice v (passi 1 e 2), questa fa ridurre l'altezza di T_j almeno di un fattore 2/3, se T_j non è già una stella radicata.
La fase di accorpamento del passo 3 non richiede O(log(n)) tempo come nel precedente algoritmo, perché nel nostro caso viene eseguito un passo per volta della procedura del salto del puntatore, ma ciò rimane valido solo a condizione che l'altezza degli alberi non sia maggiore di n.
Vediamo l'input e l'output dell'algoritmo.
Input:

1) Un insieme di archi (i,j) dati in ordine qualsiasi; un arco (i,j) apparirà come (i,j) e (j,i). Inoltre per ogni vertice i, si introduce una copia di i, i', connessa ad i in modo tale che D(i)=D(i')=i, per ogni nodo i. Ciò è necessario per evitare di innestare una o più stelle su un'altra nella prima iterazione, (alla partenza ci saranno n stelle).
2) Una pseudoforesta definita da una funzione D, tale che tutti i vertici in ogni albero appartengano alla stessa componente connessa.

Output:

la pseudoforesta ottenuta dopo
1) l'innesto degli alberi in vertici di numerazione minore di altri alberi;
2) l'innesto di stelle radicate su altri alberi, se possibile;
3) l'applicazione del salto del puntatore ad ogni vertice.

Descrizione di una iterazione generale.

```
begin
     1. Operare un'operazione d'innesto di alberi in vertici di
     numerazione minore di altri alberi, come segue:
     for tutti gli archi (i,j) di E in parallel do
        if (D(i)=D(D(i))) and (D(i)≠D(j)) then D(D(i)):=D(j)
     2. Innestare le stelle radicate su altri alberi, se possibile,
     come segue:
     for tutti gli archi (i,j) di E in parallel do
        if (i appartiene alla stella radicata) and (D(i)≠D(j)) then
        D(D(i)):=D(j)
     3. Se tutti i vertici sono in una stella radicata, allora
     termina; altrimenti, opera il salto del puntatore su ogni
     vertice come segue:
     for tutti i vertici i in parallel do
        D(i):=D(D(i))
end.
```

Diamo di seguito la procedura utile a riconoscere quando un vertice i appartiene ad una stella:

```
begin
for all vertices i in parallel do
      star(i):=true;
      D(i)≠D(D(i)) then star(i), star(D(i)), star(D(D(i))):=false;
      star(i):=star(D(i));
end.
```

Alla fine della procedura star(i)= true se e solo se i appartiene ad una stella; la complessità è costante.
Una volta terminato l'algoritmo, ogni vertice appartenente ad una stessa componente connessa apparterrà ad una stella con radice o sarà egli stesso radice.
Possiamo concludere che, poiché la cardinalità di una componente connessa è, nei due casi limite, n o 1, rispettivamente quando si riesce a trovare un'unica componente connessa o quando ci sono n vertici isolati, al più è possibile trovare nel grafo un cammino lungo n.
La complessità totale dell'algoritmo risulta essere O(log(n)) (dovuta alle sole iterazioni, essendo i tre passi costanti) per un totale di O((m+n)log(n)) operazioni.
Appare chiaro a questo punto che per l'implementazione dell'algoritmo per grafi sparsi occorre un modello di macchina PRAM CRCW (il passo 3 richiede una macchina a lettura concorrente, i passi 1 e 2 necessitano di una macchina a scrittura concorrente).

2.11 Minimo albero di copertura

Dato il grafo G=(V,E) pesato e non orientato, cerchiamo un albero T, sottografo di G senza cicli, detto spanning tree, tale che T=(V,E_T), dove E_T è un sottinsieme di E. Fra tutti gli spanning tree possibili del nostro grafo G pesato, cerchiamo quello di costo minimo, MST (il costo dell'albero è dato dalla somma dei pesi degli archi).
Supponiamo, senza perdita di generalità, che tutti i pesi siano distinti; se così non fosse, sarebbe sempre possibile costruire una funzione che associ i pesi agli archi in modo tale che essi siano distinti, per esempio: w(e)=w(e)+s(e), il peso dell' arco è aumentato del numero seriale associato all'arco stesso. Altrimenti se si verifica ancora una coincidenza di pesi, ovvero w(x,y)=w(e)=w(e')=w(z,t), è possibile distinguerli nuovamente associandoli all'uno o all'altro vertice in questo modo: w'(e)=(w(e),x)≠(w(e'),z)=w'(e'). Se ci fosse ancora bisogno di differenziare i pesi si potrebbe trasformare le coppie in terne così: w''(e)=(w(e),x,y)≠(w(e'),z,t). In questo modo si riesce a generare un unico spanning tree.

Lemma 1: *sia G un grafo pesato connesso. Per ogni vertice u del grafo esiste un vertice C(u) appartenente a V tale che (u,C(u)) è l'arco di peso minimo incidente sul nodo u. Allora:*
1) tutti gli archi del tipo (u,C(u)) debbono appartenere al MST;
2) la funzione C definisce una pseudoforesta tale che ogni albero orientato ha esattamente un ciclo contenente esattamente due archi;

Dimostrazione:
1) Sia T un minimo albero di copertura per il grafo G e supponiamo che l'arco (v,C(v)) non gli appartenga, per qualche nodo v del grafo G. Allora, se non esiste in T un arco diretto tra v e C(v) esisterà un cammino $P=(v,x_1,\ldots,x_s,C(v))$, e, sostituendo in esso al primo arco l'arco (v,C(v)) e togliendo l'ultimo arco per non avere un ciclo, si ottiene uno spanning tree T' di costo inferiore a quello di T. Ma ciò è assurdo poiché, per le ipotesi fatte, già T é un MST, perciò (v,C(v)) deve appartenere a T.
2) Il punto 1 già contiene le proprietà che garantiscono la definizione di pseudoforesta.

Lemma 2: *sia $V=UV_i$ una partizione arbitraria di V con i corrispondenti sottografi $G_i=(V_i,E_i)$, $1\leq i\leq t$. Per ogni i, sia e_i l'arco di peso minimo che connette un vertice di V_i con un vertice di $V-V_i$. Allora tutti gli archi e_i appartengono al MST del grafo G.*

Dimostrazione: il lemma 2 è semplicemente la generalizzazione del lemma 1 ad una foresta di singoli alberi G_i: se per ciascuno di essi già conosco uno spanning tree, nel momento in cui li voglio unire cerco la connessione di costo minimo da un albero ad un altro.

Esistono tre diversi algoritmi che si basano sulla strategia greedy per risolvere il problema del MST e tutti derivano le loro procedure dai suddetti lemma.
Algoritmo di Prim:
si parte da un vertice qualunque e si applica il passo 1 del primo lemma; costruito così un albero iniziale, si aggiungono successivamente ad esso archi di peso minimo applicando serialmente il secondo lemma.
Algoritmo di Kruskal:
si parte da una foresta fatta di soli vertici e da un ordinamento degli archi in funzione del costo crescente e considerando quelli che non formano cicli; a questo punto vengono scelti via via gli archi di costo minimo i cui estremi appartengono ad alberi differenti, gli altri vengono scartati altrimenti formerebbero dei cicli.
Algoritmo di Sollin:
di questo algoritmo faremo un'analisi dettagliata e mostreremo l'implementazione parallela.

Si parte da una foresta di soli vertici, $F_0=(V,\emptyset)$, e applicando i punti 1 e 2 del lemma 1 si costruisce gradualmente una pseudoforesta, la prima fase di accorpamento è analoga a quella dell'algoritmo per i grafi densi. Il processo continua finché un singolo albero contiene tutti i vertici. È chiaro che il numero di alberi nella foresta F_s è al più la metà del numero di alberi in F_{s-1}, perciò questo algoritmo richiede un numero di iterazioni O(log(n)).
La parallelizzazione dell'algoritmo di Sollin ha complessità $O(\log^2(n))$, per un totale di $O(n^2)$ operazioni.
Sia W la matrice dei pesi del grafo connesso dato:

$$W(i,j)=\begin{cases} w(i,j) \text{ se l'arco } (i,j) \text{ esiste} \\ \infty \text{ altrimenti.} \end{cases} \qquad (1\leq i,j\leq n)$$

Vediamo la struttura dell'algoritmo.
Input:
W(n*n) matrice dei pesi tutti distinti del grafo connesso.
Output:
un'etichetta per ciascuno degli archi appartenenti al MST.

```
begin
    1. W0:=W; n0:=n; k:=0;
    2. while (nk>1) do
        2.1 k:=k+1;
        2.2 C(v):=u, dove Wk-1(v,u)=min{Wk-1(v,x) | x≠v}; marca
        (v,C(v));
        2.3 accorpa ogni albero orientato della pseudoforesta
        definita da C in una stella con radice;
        2.4 nk:=numero delle stelle radicate; sia Wk-1(nk*nk) la
        matrice dei pesi indotta dal nuovo insieme dei meganodi;
    3. restituisci il nome originario ad ogni arco marcato;
end.
```

Il ciclo while viene eseguito al più O(log(n)) volte, ogni volta si dimezza almeno il numero di vertici. Ci occorrono $O(n^2)$ processori e O(log(n)) passi, quindi la complessità totale è $O(\log^2(n))$.

2.12 Ear Decomposition

Si tratta di un metodo per partizionare gli archi di un grafo in cammini semplici, metodo che risulta efficiente per ottenere, a partire da un grafo, strutture più semplici. Dato il grafo non orientato G=(V,E) con |V|=n e |E|=m, sia P_0 un cammino o un ciclo semplice arbitrario in G. Un'ear decomposition che parte da P_0 è una partizione ordinata dell'insieme degli archi $E=P_0 \cup P_1 \cup \ldots \cup P_k$, tale che per ogni $1 \le i \le k$ P_i è un cammino semplice i cui estremi (e nessuno dei punti interni) appartengono a $P_0 \cup P_1 \cup \ldots \cup P_{i-1}$. Se nessuno dei P_i è un ciclo, la decomposizione è detta aperta. In figura 2.37a è mostrata una decomposizione normale, mentre in figura 2.37b ne è mostrata una aperta.

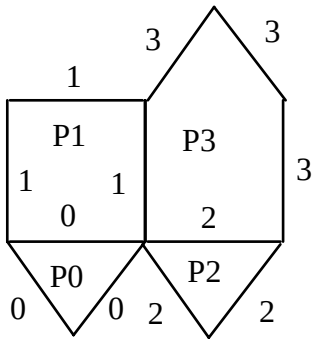


Figura 2.37a

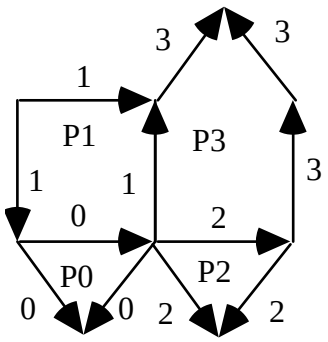


Figura 2.37b

Teorema: *un grafo non orientato G=(V,E) ha un ear decomposition se e solo se è privo di ponti, ossia è biconnesso (un ponte è uno arco la cui eliminazione sconnette il grafo).*

Un'ear decomposition genera un insieme indipendente di cicli ossia, togliendo uno arco arbitrario a ciascun cammino semplice P_i (ear) si ottiene uno spanning tree di G, perciò il numero di ear è pari al numero di archi di G non nell'albero: m-n+1.
Una regola per la costruzione di un'ear decomposition può essere la seguente: iniziamo col calcolare un albero di copertura radicato T associato al grafo G. Associamo ad ogni arco e non di T un ear P_e. Sia C_e un cammino o ciclo base indotto dall'arco e. Non è corretto far coincidere banalmente C_e con P_e, in generale non otterremmo infatti una partizione, ma un ricoprimento. Allora diventa necessario interrompere i cicli in cammini tali che ciascuno di essi appartenga ad un singolo ear. A questo scopo etichettiamo ogni arco e=(u,v) non in T come segue:

label(e)=(level(e),s(e))

dove level(e)=level(lca(u,v)) (lca=lowest common ancestor) e s(e) è il numero seriale corrispondente all'arco e ($1 \leq s(e) \leq m$). Per ciascun arco g di T label(g) è la più piccola etichetta di qualsiasi arco non di T nel cui ciclo è contenuto g (figura 2.38 e 2.39).

Lemma: *sia T uno spanning tree con radice del grafo G=(V,E), allora per ogni arco g di E sia label(g) la funzione precedentemente definita. Dato e, arco non dell'albero, sia P_e={e}U{g appartenente a T | label(g)=label(e)}. Allora P_e è un cammino semplice o ciclo. Inoltre ogni arco dell'albero appartiene esattamente ad uno di questi cammini.*

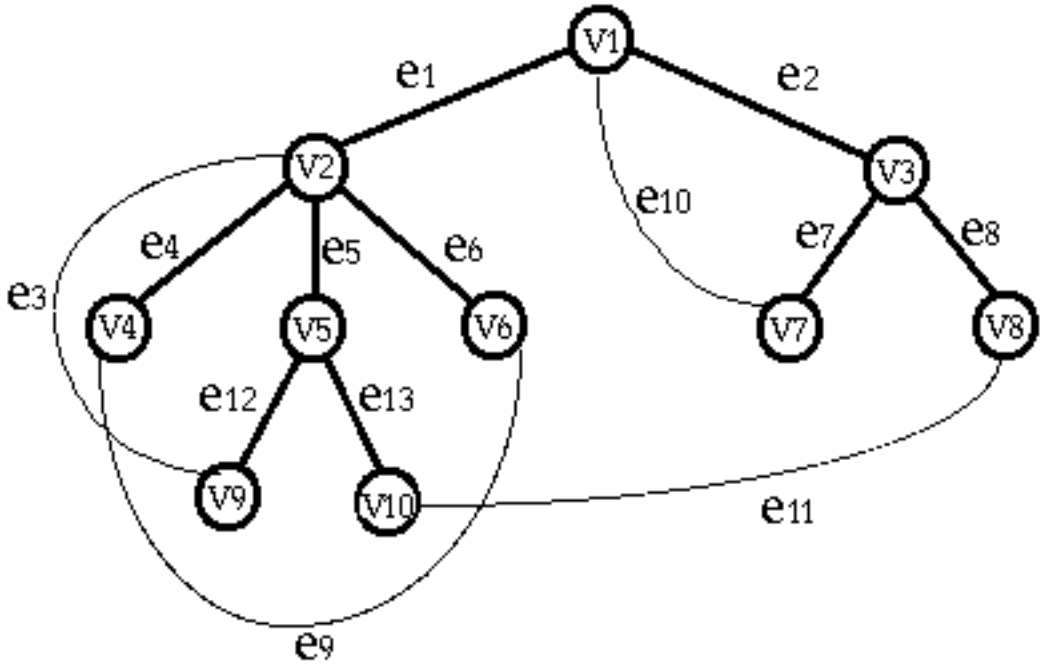


Figura 2.38

arco	label
e_1	(0,11)
e_2	(0,10)
e_3	(1,3)
e_4	(1,9)
e_5	(0,11)
e_6	(1,9)
e_7	(0,10)
e_8	(0,11)
e_9	(1,9)
e_{10}	(0,10)
e_{11}	(0,11)
e_{12}	(1,3)
e_{13}	(0,11)

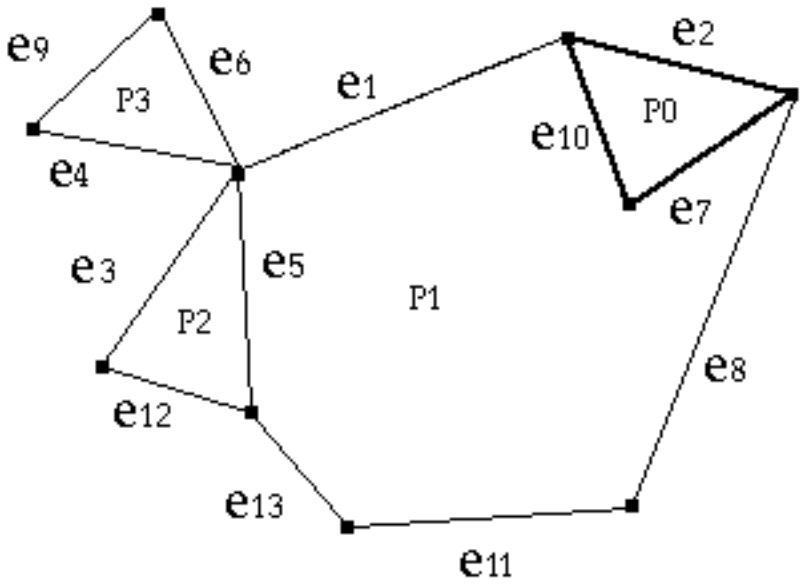


Figura 2.39

Input:

G grafo biconnesso non orientato rappresentato come sequenza di archi.

Output:

insieme ordinato di archi che rappresenta un'ear decomposition.

```
begin
  1. trovare T spanning tree di G;
  2. fissare r radice di T e calcolare per ogni v≠r level(v) e p(v),
  il padre di v;
  3. per ogni arco e=(u,v) calcolare lca(e)=lca(u,v) e
  level(e)=level(lca(e)); poni label(e)=(level(e),s(e)) dove s(e) è
  il numero seriale corrispondente ad e;
  4. per ogni arco g di T calcolare label(g);
  5. per ogni arco e non di T poni Pe={e}U{g appartenente a T |
  label(g)=label(e)}; ordina i Pe in base a label(e);
end.
```

Le figure 2.38 e 2.39 mostrano un esempio di applicazione di tale algoritmo. La sua complessità è data dalla somma delle complessità dei singoli passi; il passo 1 richiede $O(\log^2(n))$ tempo per la costruzione dello spanning tree utilizzando un modello di macchina CREW con $O(n^2)$ processori. Il passo 2 si realizza mediante il tour di Eulero su di una macchina EREW ad n processori in tempo costante. Il terzo passo è dominato dal calcolo dell'lca che è logaritmico su una EREW ad n processori. La complessità del quarto passo è data dal costo logaritmico della ricerca del minimo tra la varie etichette. Il passo 5 è suddiviso in un ordinamento (logaritmico) e un'operazione analoga a quella della scrittura concorrente la cui complessità è $O(\log(n))$. Tutto l'algoritmo risulta di complessità $O(\log^2(n))$.
Dato un cammino semplice qualunque P_e, resta da mostrare che i suoi estremi cadono in qualche cammino $P_{e'}$ tale che label(e')<label(e) e che i suoi nodi interni non appartengano a nessuno di tali ear.
Sia e_0 l'arco non di T di etichetta minima tra tutte le etichette dei nodi non dell'albero e sia e un arco non dell'albero tale che $label(e)>label(e_0)=(0,s(e_0))$ dove $s(e_0)$ è il minimo tra tutti gli archi e ha la radice come lca. L'ear P_{e_0} corrispondente ad e_0 è il ciclo base C_{e_0}.
Il precedente lemma ci dice che o P_e è il ciclo base C_e oppure ciascuno degli estremi appartiene a qualche arco g tale che label(g)<label(e). In tal caso abbiamo ottenuto il risultato perseguito. Se $P_e=C_e$ sia v=lca(e). Se v è la radice non c'è nulla da dimostrare poiché C_e interseca C_{e_0} alla radice. Altrimenti consideriamo l'arco (v,p(v)) che deve appartenere a qualche C_f, per qualche f non dell'albero (altrimenti f sarebbe un ponte, ma ciò è impossibile). Level(lca(f)) deve essere più piccolo di level(v); quindi v appartiene ad un ear la cui etichetta è minore di label(e). Da ciò segue la tesi.

2.12.1 Calcolo del lowest common ancestor

Per il calcolo del lowest common ancestor tra u e v, è possibile identificare due casi distinti per i quali possiamo risolvere immediatamente il problema dato. Il primo caso è quello in cui T è un cammino semplice per cui basta confrontare le distanze di u e v dalla radice per stabilire quale dei due precede l'altro.

Il secondo caso è quello in cui T è un albero binario completo (figura 2.40). Mediante il tour di Eulero, numeriamo i nodi dell'albero. Per ciascuno di essi consideriamo la codifica binaria del numero associato. Il lowest common ancestor tra u e v si calcola tenendo presente che le codifiche binarie dei numeri associati ad u e v sono composte da cifre uguali a partire dalla più significativa fino ad incontrarne una distinta; è facile notare che il numero binario associato all'lca richiesto si ottiene copiando le cifre più significative in comune tra i due numeri, seguite dal massimo tra le due cifre binarie differenti (ovviamente uno) e da zeri per tutte le cifre restanti (figura 2.40).

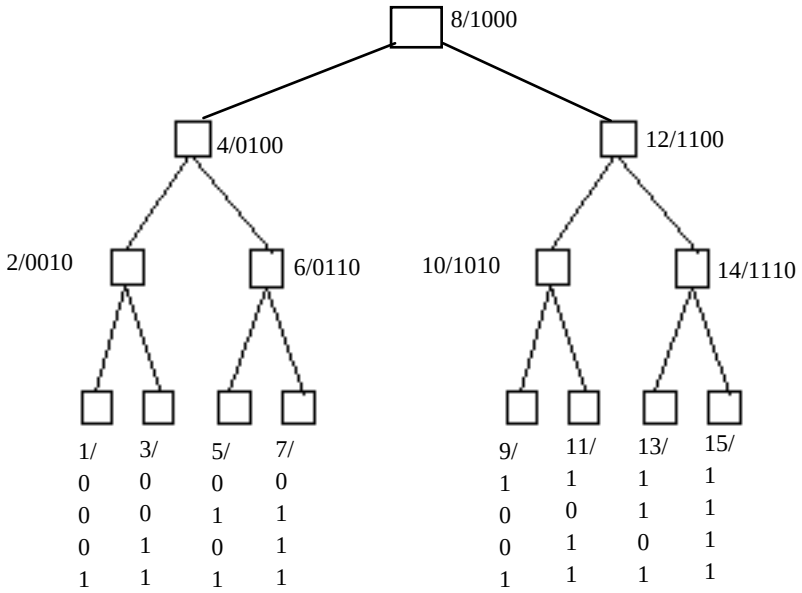


Figura 2.40

Ciò deriva dal fatto che, essendo stati numerati progressivamente i nodi secondo una visita in postordine o in preordine, il numero associato al lowest common ancestor è sempre compreso tra quelli associati ad u e v; quindi i bit più significativi tra loro in comune dovranno essere in comune anche all'lca.
Nel caso di alberi qualunque, possiamo generalizzare questa tecnica trasformando T in un albero binario (in O(log(n)) tempo) eventualmente non completo come segue: ogni nodo u che abbia più di due figli viene spezzato nel numero di nodi necessari u',u''… che diventeranno suoi discendenti e che erediteranno i figli originari di u (figura 2.41 e 2.42). Il procedimento è logaritmico poiché ogni nodo diventa radice del sottoalbero di altezza al più logaritmica rispetto al numero di figli di u che al massimo è n.

Teorema: *dato un albero T=(V,E) con radice, siano A, level(v), l(v) e r(v) per ogni nodo v appartenente a V rispettivamente il vettore euleriano che contiene il nome dei nodi nell'ordine di visita secondo il tour di Eulero, il livello di v, il più piccolo e il più grande indice di A in cui appare il nodo v nella visita. Siano u e v due vertici arbitrari distinti di T, allora vale quanto segue:*

1. u è antenato di v <=> l(u)<l(v)<r(u).
2. u e v sono scorrelati, ossia nessuno è discendente dell'altro, <=> r(u)<l(v) o r(v)<l(u).
3. se r(u)<l(v) allora lca(u,v) è il vertice di minimo livello nell'intervallo [r(u),l(v)]

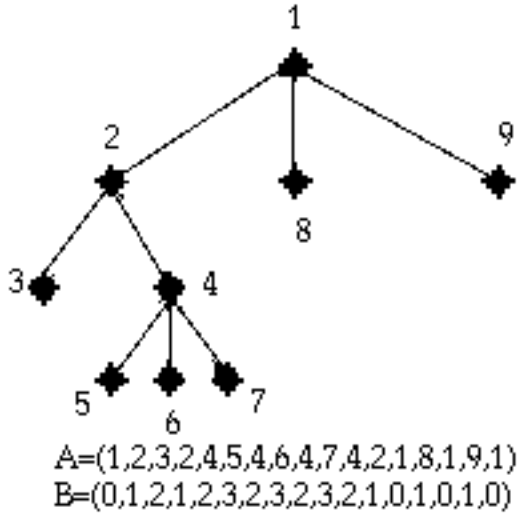


Figura 2.41

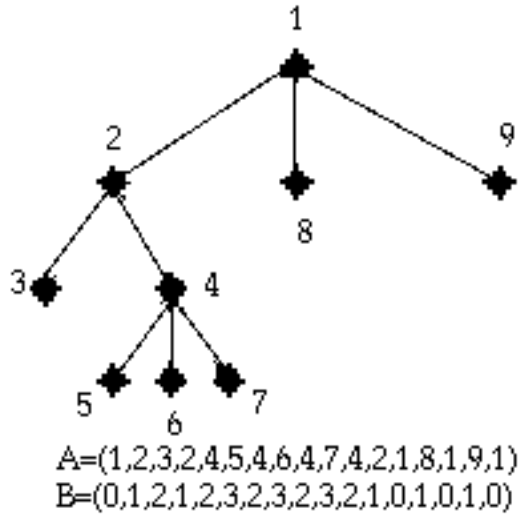


Figura 2.42

Costruito il vettore euleriano, definiamo B il vettore dei livelli nelle cui posizioni, in corrispondenza con gli elementi del vettore A, vengono memorizzati i livelli di tali elementi. Per ogni vertice v≠r, dove r è la radice di T, possiamo calcolare l(v) ed r(v) come segue. Dato il vettore A, l'elemento a_i=v è l'apparizione più a sinistra di v in A se e solo se level(a_{i-1})=level(v)-1. Inoltre a_i=v è l'apparizione più a destra di v in A se e solo se level(a_{i+1})=level(v)-1.Il calcolo di l(v) ed r(v) richiede tempo costante quindi il calcolo del lowest common ancestor richiede tempo logaritmico su di una macchina EREW.

2.13 Geometria computazionale

Oggetto della geometria computazionale è lo studio di problemi di tipo geometrico e la ricerca di algoritmi efficienti per la loro risoluzione. In questo paragrafo ci occuperemo di due casi particolarmente interessanti:

1) problemi di inclusione: dato un insieme di punti del piano, vedere quali di essi appartengono o meno ad un dominio specificato;
2) problemi di intersezione: trovare le intersezioni tra linee, poligoni, rettangoli, poliedri, semispazi etc;
3) problemi di costruzione: dato un insieme di punti del piano, determinare il minimo poligono convesso che li contiene tutti.

2.14 Un problema di inclusione

Si dice che un grafo è planare, se può essere disegnato in un piano in modo tale che nessuna coppia dei suoi archi si intersechi. Una sottodivisione planare di un grafo planare è costituita da un insieme di poligoni adiacenti.
Il problema che ci viene posto è il seguente:
data una sottodivisione planare ed un punto p nel piano, stabilire quale poligono (se esiste) contiene p (figura 2.43).

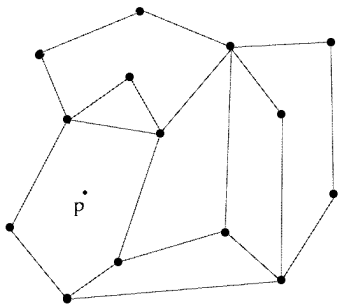


Figura 2.43

Per risolvere questo problema, poniamoci prima l'obiettivo di risolvere il sottoproblema: dato un poligono ed un punto p nel piano, determinare se p cade all'interno del poligono oppure no.
Stabiliamo di implementare il nostro algoritmo su di un modello di macchina SIMD a rete di interconnessione ad albero binario completo. Dato il poligono Q avente n lati, si considera un albero binario B di dimensioni tali da contenere almeno gli n lati; quindi si assegna un arco e_j ad ogni processore P_j, dandogli le coordinate dei due punti corrispondenti ai due nodi che sono gli estremi dell'arco.
L'idea dell'algoritmo è la seguente:
si disegna una linea verticale L_p passante per il punto p all'interno del piano cartesiano contenente l'insieme di punti che costituiscono il poligono Q; quindi si determinano le intersezioni tra questa linea e i lati di Q. Se il numero di intersezioni al di sopra del punto p è dispari, allora p cade all'interno di Q; altrimenti p è fuori Q. Queste affermazioni sono motivate dal fatto che un numero pari di intersezioni al di sopra di p formano una spezzata chiusa che esclude p stesso, mentre un numero dispari di esse può richiudersi soltanto al di sotto di p, includendolo (figura 2.44).

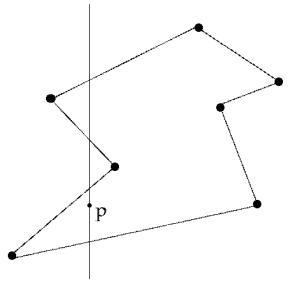


Figura 2.44

Ciascun processore deve testare due condizioni entrambe di tipo binario, presenza dell'intersezione o meno e posizione di essa al di sopra di p oppure no, l'uscita che il processore quindi produce è l'AND logico dei due test. Poi le risposte di ogni processore vengono sommate fra loro e portate fino alla radice, la quale al termine di questa operazione conterrà la risposta definitiva al problema: somma dispari, p è all'interno del poligono; somma pari, p è esterno a Q.
Analizziamo nel dettaglio l'algoritmo. Esso presenta una fase di inizializzazione costituita dal broadcast dalla radice alle foglie delle coordinate del punto p, (x_p,y_p); una seconda fase costituita da una coppia di operazioni costanti eseguite in parallelo ed una somma lungo un percorso foglia–radice. Globalmente, quindi, la complessità è O(log(n)) ed il costo O(nlog(n)). Purtroppo questo risultato non è ottimale, poiché sappiamo che esiste un algoritmo seriale che risolve questo problema con complessità lineare, ma non è comunque difficile modificarlo per renderlo ottimo.
Possiamo generalizzare inoltre la procedura allo scopo di testare una sequenza di punti seguendo uno schema di tipo pipeline. Più precisamente ciò viene realizzato in questo modo: al livello zero il processore P_1 esegue i due test, quindi trasmette il risultato al suo figlio destro o sinistro; quando il computo è passato ai processori al livello uno, i processori al livello zero sono pronti a ricevere nuovi dati. Basta introdurre un ciclo ed a partire dalle foglie si propagherà un flusso ascendente di informazioni.
Diamo infine l'algoritmo:

```
procedure PuntoNelPoligono(xp,yp,risposta);
  1. (1.1) P1 legge (xp,yp);
     (1.2) if Lp ha intersezione con ei sopra p then
                s1:=1;
           else
                s1:=0;
     (1.3) P1 spedisce (xp,yp,s1) a P2 e (xp,yp,0) a P3;
  2. for i:=log2(n+1)-2 downto 1 do
           for j:=(n+1)/2^(i+1) to ((n+1)/2^i)-1 in parallel do
           begin
                (2.1) Pj riceve (xp,yp,s) dal padre
                (2.2) if Lp ha intersezione con ei sopra p then
```

s_j:=1;

else

s_j:=0;

(2.3) P_j manda $(x_p, y_p, s+s_j)$ a P_{2j} e $(x_p, y_p, 0)$ a P_{2j+1};

end;

```
3. for j:=(n+1)/2 to n in parallel do
   begin
         (3.1) P_j riceve (x_p,y_p,s) dal padre
         (3.2) if L_p ha intersezione con e_j sopra p then
                         a_j:=s+1;
                   else
                         a_j:=s;
   end;
4. for i:=1 to log_2(n+1)-1 do
         for j:=(n+1)/2^(i+1) to ((n+1)/2^i)-1 in parallel do
               a_j:=a_2j+a_2j+1;
5. risposta:=a_1 mod 2;
```

Ora si può passare al problema più generale di individuare un punto in una sottodivisione planare. Supponendo che i poligoni di cui la sottodivisione è composta siano m, con al più n lati ciascuno, utilizziamo un'architettura parallela a maglia di alberi m*n (figura 2.45).

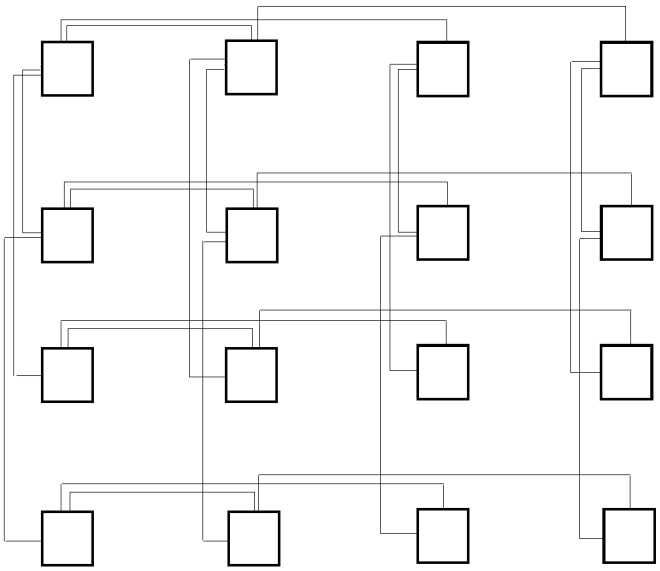


Figura 2.45

Su di ognuna delle m righe è memorizzato l'albero binario contenente i lati di un determinato poligono; analogamente alle n colonne corrispondono altrettanti alberi binari; ne risulta una fitta rete di connessioni in modo particolare nei punti estremali.
È logico pensare che la precedente procedura debba subire qualche modifica:

1) all'inizio dell'algoritmo ogni processore radice delle m righe deve contenere le coordinate del punto p;

2) il risultato che al termine dell'algoritmo si otterrà dalla radice alla riga i sarà del tipo (1,i) o (0,i);
3) una volta terminata la procedura per ciascuna riga, stabilito perciò se il punto p cade nel poligono corrispondente a quella riga, viene effettuato l'OR logico tra le informazioni contenute nelle m radici così individuate; il risultato binario di questa somma, in posizione (1,1) nella matrice dà la risposta al problema di inclusione dato.

Per quanto riguarda l'analisi della complessità dell'algoritmo, occorre considerare il computo parallelo per ogni riga O(log(n)) e l'operazione di somma sulla prima colonna O(log(m)); considerando inoltre che m=O(n), si ottiene una complessità totale logaritmica in n ed un costo $O(n^2 \log(n))$. Anche in questo caso l'algoritmo risultante non è ottimale rispetto alla soluzione seriale per lo stesso problema che ha complessità solo $O(n^2)$, ma può essere modificato così come nel caso precedente. L'implementazione in pipeline per un insieme di k punti produce una risposta in O(k+log(n)) tempo.

2.15 Un problema di intersezione

Si dice che due poligoni Q ed R si intersecano se un lato di Q attraversa un lato di R. Il nostro problema è determinare appunto, se due poligoni si intersecano.
L'approccio fornito da questa soluzione parallela è piuttosto semplice: per ciascun lato di Q si verifica se esso interseca un lato di R (figura 2.46).

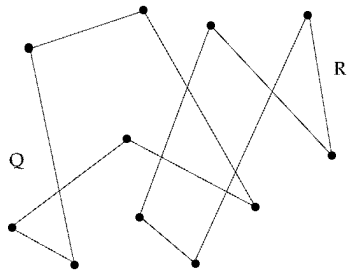


Figura 2.46

Si sceglie di utilizzare una macchina SIMD a maglia di alberi con m righe, quanti sono i lati di Q, e n/log(n) colonne, con n i lati di R. Nella fase di caricamento a ciascun processore radice per ogni colonna viene assegnato un blocco di log(n) lati di R, che verranno propagati lungo la struttura ad albero avente quel processore come radice. Ad ogni processore riga è assegnato un lato di Q che viene poi propagato ai processori figli di quella riga. Nel complesso occorrerà un tempo pari a O(log(m))+O(log(n)) per caricare in parallelo le colonne e un tempo pari a O(n/log(n)) per caricare in parallelo le righe.
A questo punto l'algoritmo prosegue con il test di intersezione che ogni processore opera conoscendo le coordinate degli estremi del lato di Q ad esso assegnato e le coordinate relative al blocco di log(n) lati di R; il tempo richiesto a questa operazione è O(log(n)), poiché i processori operano tutti contemporaneamente. La risposta binaria a questa verifica, che viene prodotta da ogni processore, viene sommata alle risposte ricevute dai figli secondo un OR logico fino a giungere alla radice della riga relativa al lato di Q considerato. Il risultato finale perviene dopo

O(n/log(n)) passi alle radici delle righe e in altri O(log(m)) passi viene portato in posizione (1,1) nella maglia, sempre con un'operazione di OR logico.
Considerando infine che $m \leq n$, si ottiene per l'intero algoritmo una complessità O(log(n)). Avendo inoltre utilizzato $O(n^2/\log(n))$ processori, il costo dell'algoritmo risulta quadratico.

2.16 Un problema di costruzione

Prendiamo in considerazione il problema dell'involucro convesso ossia, dato un insieme $S=\{p_1, p_2,..., p_n\}$ di punti nel piano, determinare il poligono convesso più piccolo che include tutti i punti di S (figura 2.47).

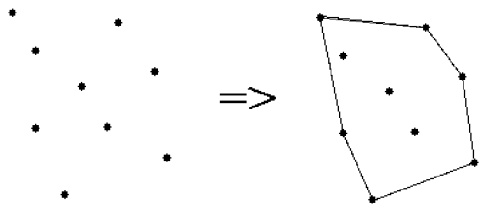


Figura 2.47

Sia quindi dato un insieme S di punti del piano, dove ciascun punto è rappresentato dalle sue coordinate cartesiane, cioé $p_i=(x_i,y_i)$: un primo approccio al problema dell'inviluppo convesso è progettato per girare su un calcolatore SIMD con una maglia di alberi e fa appoggio alle due seguenti ipotesi semplificative:

- nessun punto ha le stesse coordinate;
- nessuna terna di punti cade sulla stessa retta.

Consideriamo i quattro punti con coordinata x minima, x massima, y minima, y massima, e assumiamo che tali punti siano già noti; come appare da figura 2.48 tali punti apparterranno sicuramente all'involucro convesso.

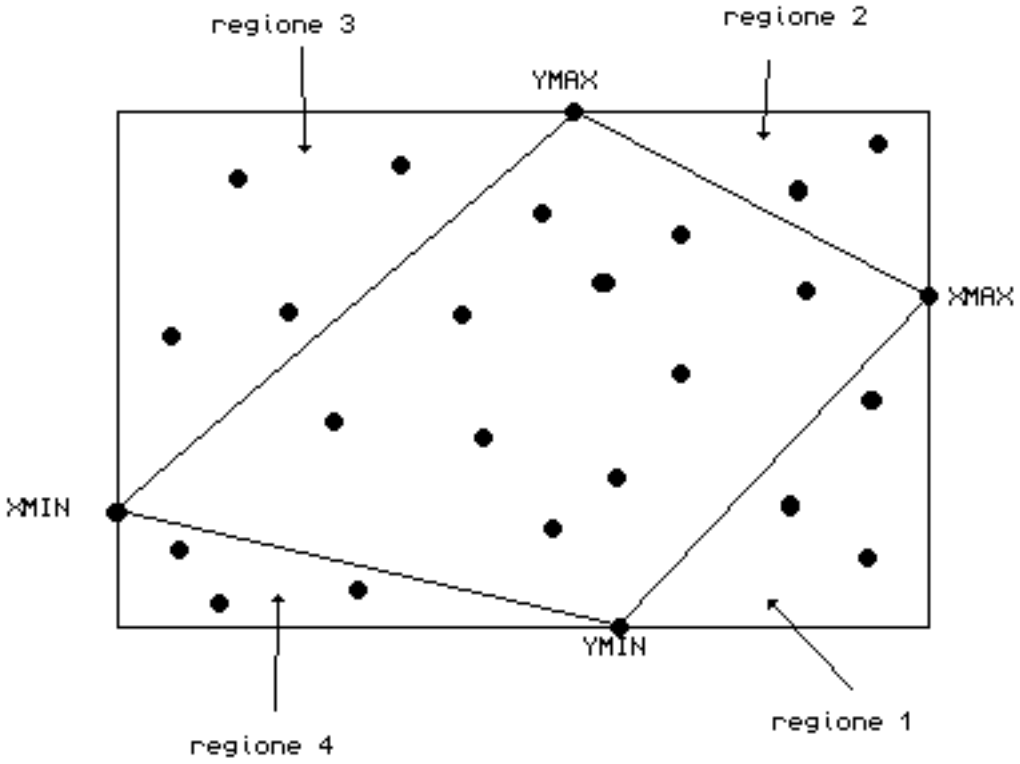


Figura 2.48

Prendendo in considerazione il poligono formato da questi quattro vertici, rimane ovvio che qualunque punto che cada all'interno del quadrilatero formato da tali punti non appartiene all'involucro convesso. Il problema di identificare l'involucro è quindi ridotto a quello di trovare un percorso poligonale convesso che unisca due punti estremi in ciascuna regione 1,2,3,4; l'involucro convesso viene ottenuto congiungendo queste quattro spezzate. L'identificazione dei lati dell'inviluppo si basa quindi sul fatto che (figura 2.49) due punti definiscono uno spigolo se tutti i punti restanti cadono dallo stesso lato rispetto alla retta tracciata tra tali due punti.

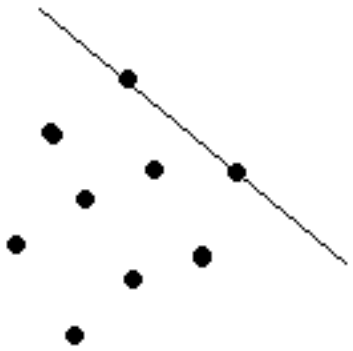


Figura 2.49

Siano ora p_i e p_j due vertici consecutivi dell'involucro convesso e si prenda p_i come origine delle coordinate; allora rispetto a tutti i punti di S, p_j forma con p_i il più piccolo angolo (sia positivo che negativo) rispetto all'asse delle x (figura 2.50).

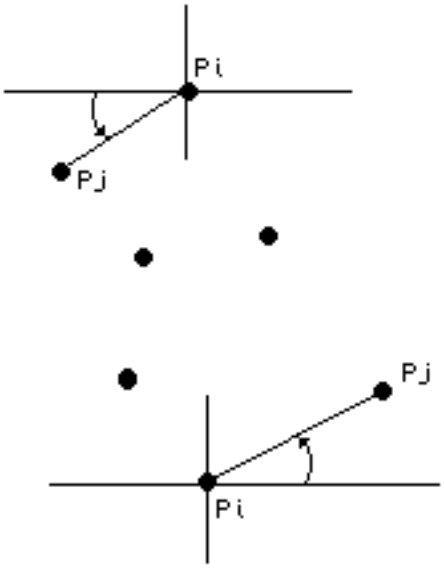


Figura 2.50

Presentiamo ora l'algoritmo supponendo di lavorare su una maglia di alberi costituita da n righe e n colonne di processori, assumendo che il processore nella riga i e colonna j denotato con P_{ij} contenga le coordinate (x_i, y_j). Abbiamo di conseguenza che:

1) tutti i processori in una colonna contengono le coordinate dello stesso punto di S;
2) le coordinate contenute in una riga formano l'insieme $S=\{(x_1,y_1),(x_2,y_2),...,(x_n,y_n)\}$ e l'algoritmo effettuerà i seguenti passi:

1)
I processori nella riga 1,2,3,4 calcolano XMAX, YMAX, XMIN, YMIN e immagazzinano le loro coordinate in P_{11}, P_{21}, P_{31}, P_{41} rispettivamente; utilizzando le connessioni ad albero, prima in colonna 1 e poi in riga 1, le coordinate dei quattro punti estremi vengono rese note a tutti i processori nella riga 1.

2)
I quattro processori corrispondenti ai punti estremali producono 1 come uscita indicando che sono vertici dell'involucro convesso. Tutti i processori nella riga 1 che corrispondono ai punti all'interno del quadrilatero formato dai quattro punti estremi producono uno zero in uscita ad indicare che non fanno parte dell'involucro convesso. Ciascuno dei processori rimanenti P_{1j} nella riga 1 identifica la regione 1,2,3 o 4 in cui il punto p_j cade comunicando questa informazione a tutti i processori P_{ij} nella colonna j. XMAX viene assegnato alla regione 1, YMAX alla regione 2, XMIN alla regione 3 e YMIN alla regione 4.

3)

Se il processore P_{1i} che corrisponde al punto P_i di S non ha fornito né 1 né 0, allora vengono eseguiti i seguenti passi dai processori nella riga i:
viene trovato il punto p_j nella stessa regione di p_i tale che (p_i,p_j) forma l'angolo più piccolo rispetto all'asse positivo x se p_i è nella regione 1 o 2, oppure l'asse negativo x se p_i è nella regione 3 o 4. Se tutti i punti rimanenti nella stessa regione di p_i e p_j cadono dallo stesso lato di una retta che passa per tali punti, allora p_i è un vertice dell'inviluppo convesso.

4)
Se p_i è stato identificato come un vertice dell'involucro convesso, allora P_{1i} produce 1 come uscita, altrimenti produce 0. Viene scelto un punto all'interno del quadrilatero formato dai quattro punti estremi che prendiamo come origine di coordinate polari e si calcolano gli angoli polari formati da tutti i punti identificabili come vertici dell'involucro convesso. Gli angoli calcolati vengono ordinati in ordine crescente fornendo i vertici del guscio ordinati in senso orario.

Ciascuno dei quattro passi richiede O(log(n)) operazioni, quindi la complessità dell'algoritmo è O(log(n)). Essendo n^2 il numero di processori, il costo dell'algoritmo risulterà $O(n^2\log(n))$, costo che non risulta ottimale.

2.16.1 Soluzione ottima per l'inviluppo convesso

L'algoritmo è strutturato per girare su un calcolatore EREW SIMD a memoria condivisa con $N=n^{1-z}$ processori. Supponiamo valide le stesse limitazioni imposte nel caso precedente e, analogamente, continuiamo ad identificare un punto di S, p_i, mediante le sue coordinate cartesiane (x_i,y_i). Indichiamo con XMIN e XMAX i due punti di coordinata x rispettivamente minima e massima, come da figura 2.51, l'inviluppo convesso è costituito da due parti: un percorso poligonale convesso superiore da XMIN a XMAX e uno inferiore da XMAX a XMIN che chiameremo rispettivamente Upper Poligonal (UP) e Lower Poligonal (LP) (figura 2.51).

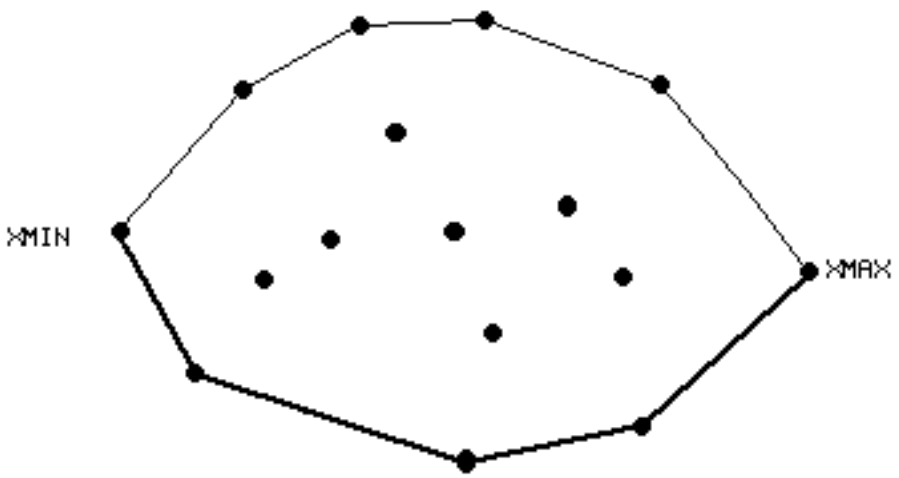


figura 2.51

La giustapposizione di UP e LP ci fornirà il nostro inviluppo convesso. L'algoritmo prende in input i punti di S e restituisce in output la lista dei vertici dell'inviluppo e può essere schematizzato nel modo seguente:

1) xmin:= indice di XMIN in S ; xmax := indice di XMAX in S;
2) UP:= lista dei vertici del percorso poligonale convesso superiore;
3) LP:= lista dei vertici del percorso poligonale convesso inferiore;
4) LP:= LP-{xmin, xmax}; CH:= UP seguita da LP.

È possibile disporre immediatamente dei passi 1 e 4: il passo 1 mediante la selezione parallela che richiede un tempo pari a $O(n^2)$, il passo 4 mediante una operazione di rimozione di punti e di fusione di due liste realizzabile in tempo costante. Non ci resta che andare ad analizzare in dettaglio i passi 2 e 3 la cui complessità, come vedremo, ci permette di determinare l'inviluppo convesso in un tempo $O(n^z\log(h))$, ove h è pari al numero di lati dell'inviluppo convesso e un costo $O(n\log(n))$.

2.6.1.1 Ricerca della poligonale superiore

La ricerca della poligonale superiore si basa sulla seguente proprietà: se viene disegnata una linea verticale da qualche parte tra XMIN e XMAX in modo tale che non passi attraverso il vertice dell'inviluppo convesso, questa linea incrocia esattamente un lato della spezzata superiore. Poniamo quindi una linea verticale che suddivide S nei due insiemi S_{sx} e S_{dx} approssimativamente della stessa dimensione. Chiamiamo ponte da Ssx a Sdx l'unico lato intersecato da tale retta (figura 2.52).

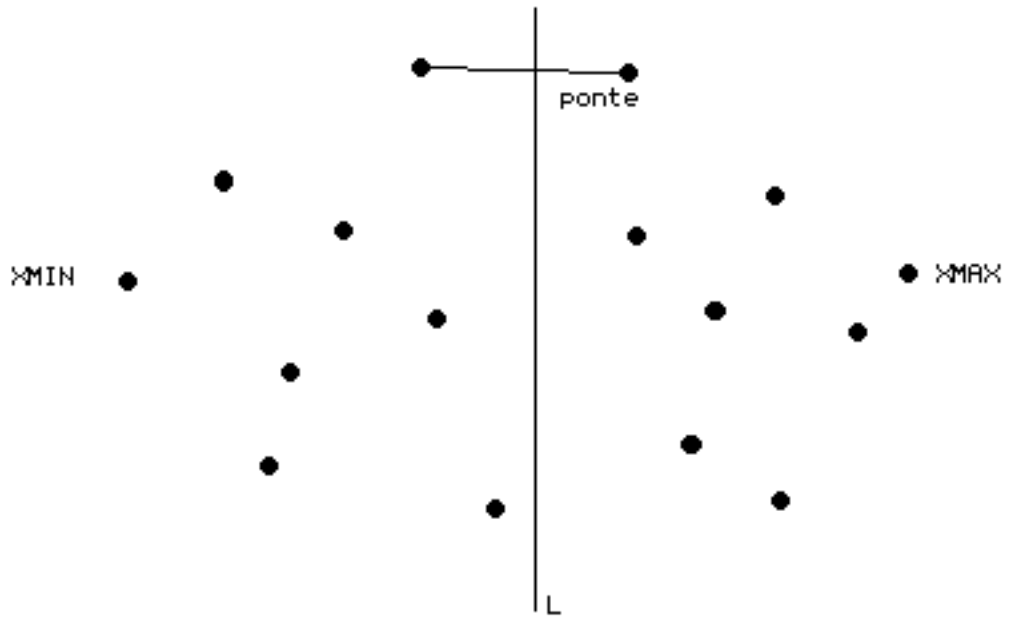


figura 2.52

Applichiamo quindi l'algoritmo in modo ricorsivo su S_{sx} e S_{dx} determinando tutti i ponti con una tecnica che, similmente al seriale, è basata su di uno schema di tipo divide et impera. Essendo un'implementazione parallela di questa tecnica, i due passi ricorsivi andrebbero eseguiti simultaneamente, ciò in realtà è irrealizzabile dato il numero inadeguato di processori necessari. Dato il bilanciamento dei due

sottoinsiemi, ognuno dei due contiene approssimativamente n/2 punti di S e quindi richiederebbe $(n/2)^{1-z}$ processori, ossia una quantità maggiore degli $(n^{1-z})/2$ processori che sarebbero stati assegnati ad ogni sottoproblema se i due passi fossero stati eseguiti simultaneamente. Dovendo tener conto di tali limitazioni l'algoritmo procede nel modo seguente: sia $k=2^{1/(z-1)}$, si trovano per prima cosa 2k-1 linee verticali $L_1,L_2,...,L_{2k-1}$ che dividono S in 2k sottoinsiemi S_i i=1,2,...,2k di dimensione n/2k ciascuno, questi sottoinsiemi sono tali che $S_{sx}=S_1 \cup S_2 \cup ... \cup S_k$ e $S_{dx}=S_{k+1} \cup S_{k+2} \cup ... \cup S_{2k}$ (figura 2.53).

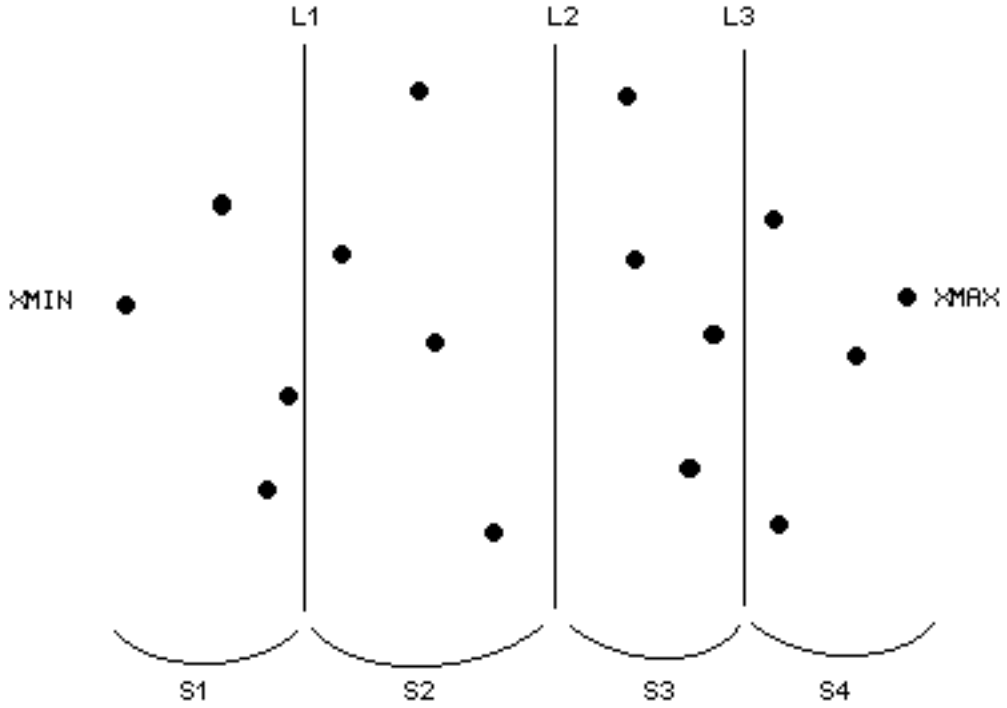


figura 2.53

Nel passo successivo l'algoritmo calcola il lato (a_i,b_i) del percorso superiore che incrocia la linea verticale L_i, i=1,2,...2k. La procedura viene applicata in modo ricorsivo in parallelo prima a S_i i=1,2,...,k poi a S_j j=k+1, k+1,...2k utilizzando $(n^{1-z})/k$ processori per ogni sottoinsieme (figura 2.54).

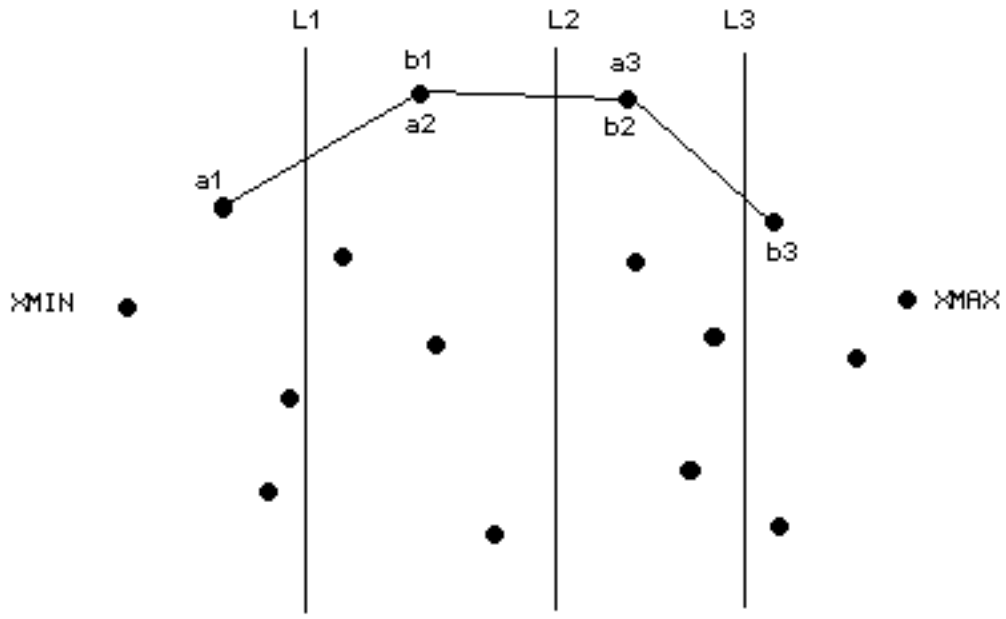


figura 2.54

Rimane da vedere come viene effettuato il calcolo dei ponti. L'algoritmo seguente dato un insieme S di n punti e un numero reale A come ingresso, restituisce due punti a_i e b_i dove (a_i,b_i) è l'unico lato del percorso superiore che interseca la linea verticale L_i la cui equazione è x=A:

1)
Ad ognuno degli n^{1-z} processori assegnamo un sottoinsieme di dimensione n^z di punti di S. In parallelo ogni processore crea $n^z/2$ coppie (p_u,p_v) tali che $x_u<x_v$. Le coppie ordinate definiscono n/2 linee rette di cui calcolo le pendenze S_1, S_2,...$S_{n/2}$ al fine di determinare l'angolazione ipoteticamente giusta utile a determinare i lati del guscio convesso. Il passo richiede quindi tempo $O(n^z)$.

2)
Trova la mediana K tra pendenze S_i mediante selezione parallela in un tempo pari al tempo del passo precedente.

3)
Massimizzando la quantità y_j-Kx_j, troviamo una linea retta Q di pendenza K che contenga almeno un punto di S e che non contenga alcun punto di S sopra di se, operando in parallelo, ciascun processore sul suo sottoinsieme di coppie di punti. Anche tale passo richiede un tempo $O(n^z)$.

4)
In tempo costante un processore verifica se Q contiene due punti di S, uno su ciascun lato di L_i e in caso affermativo restituisce tale coppia di punti come risultato, altrimenti aggiorna S riducendolo opportunamente.

Tale algoritmo viene applicato ricorsivamente su S con la seguente procedura:

Procedura PONTE (S,A)

1)

Ogni processore in parallelo determina (p_u,p_v) tali che $x_u<x_v$ sul suo sottoinsieme di punti, ne calcola la pendenza e la memorizza in una opportuna casella del vettore P delle pendenze.

2)

Determiniamo la mediana K nel vettore delle pendenze.

3)

Determiniamo una retta Q di pendenza K che passi p_i e che non abbia nessun p_j sopra di se.

```
4)
If Q contiene due punti di S, uno su ciascun lato di Li : x=A
    then li restituisce come spigolo del guscio convesso
        else if Q non contiene nessun punto di Sdestro
            then per ciascuna linea retta che attraversa (pu,pv)
            con una pendenza più grande o uguale a K S:=S-{pu}
            else if Q non contiene nessun punto di Ssinistro
                then
                per ciascuna linea retta che attraversa (pu,pv) con
                una pendenza minore o uguale a K, S:=S-{pv}
            endif
        endif
endif

5)
PONTE(S,A).
```

La procedura ponte richiede quindi un tempo di calcolo dell'ordine di n^z . La ricerca del guscio inferiore si risolve analogamente a questo caso.