Università di Roma "La Sapienza"
C.d.L. in Informatica
Algoritmi e Strutture Dati
Anno accademico 2000-2001

MACCHINE PARALLELE

P-RAM

RETI D'INTERCONNESSIONE

TEOREMA DI BRENT

Roberto Belloni
Marco Gesumundo
Emanuele Quintarelli
Andrea Deodori

Calcolo Parallelo e Classificazione di Flynn

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.

Un **algoritmo parallelo** è un metodo di soluzione per un dato problema pensato per essere eseguito su di una macchina parallela. Per progettare efficientemente un algoritmo parallelo, è necessaria perciò, una chiara comprensione del modello computazionale parallelo, per il quale l'algoritmo è ideato.

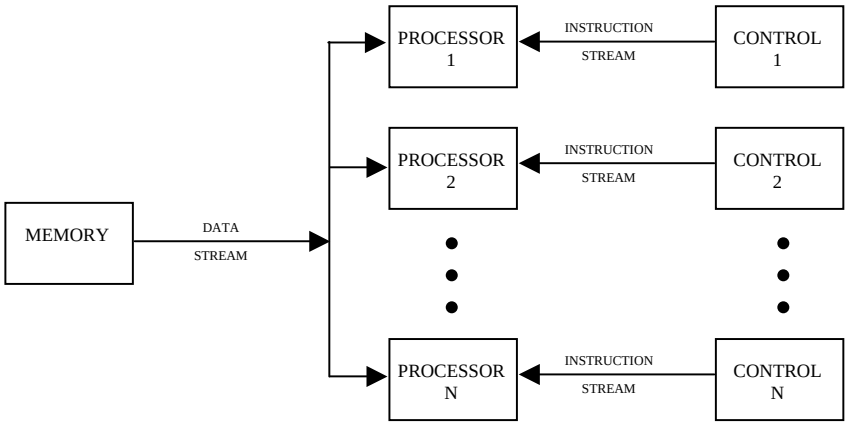
Per chiarire a quale modello di macchina nel seguito si sta facendo riferimento, richiamiamo velocemente una classificazione ormai classica dovuta a **Flynn.**

Questa classificazione distingue i modelli di macchine parallele in base ai flussi di dati ed istruzioni:

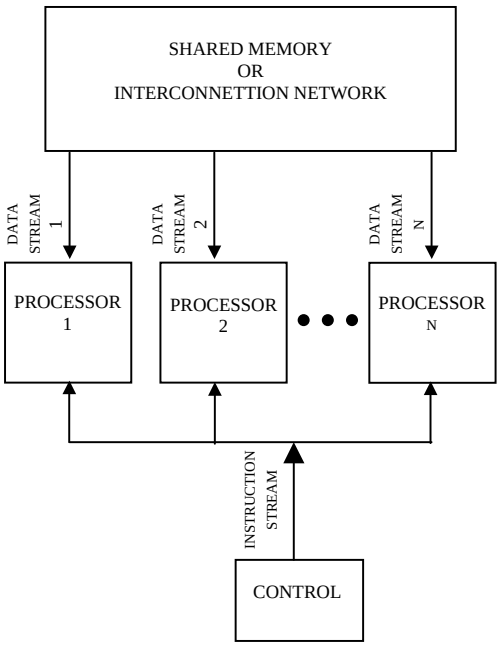
- **SISD** (Single Instruction Single Data) : un'unica unità controlla l'unico flusso d'istruzioni che lavora su di un solo flusso di dati.
  La Macchina di Von Neumann è ad esempio una SISD così come la RAM (Random Access Machine) che astrae tutti i modelli di macchine reali sequenziali.

- **MISD** (Multiple Instruction Single Data): più flussi d'istruzioni lavorano su un unico flusso dati. In questo caso allora, dalla memoria arriva l'unico flusso dati che viene dato in input a tutti i processori. Ognuno di questi, tramite la propria Control Unit, esegue il suo programma sui dati simultaneamente con gli altri.
  Ad esempio nel riconoscimento di diverse categorie di oggetti possiamo utilizzare piu' 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): unico flusso d'istruzioni che lavora su più flussi dati. In sostanza si tratta di una serie di processori dotati di una memoria locale comandati da una sola CU e capaci di comunicare tra loro grazie ad un opportuno blocco funzionale che può essere realizzato tramite una Memoria Condivisa od una Rete di Interconnessione. Mentre la CU distribuisce il flusso di istruzioni ai vari processori, questo blocco si occupa di distribuire i dati ai diversi processori.



- **MIMD** (Multiple Instruction Multiple Data): è il modello più generale e prevede più flussi di istruzioni e più flussi dati. Abbiamo cioè un numero qualsiasi di processori capaci di lavorare su un numero qualsiasi di flussi dati. Eventualmente ogni processore ha associato un suo flusso dati. A differenza delle SIMD non esiste più un'unica CU che distribuisce le istruzioni ai processori. I processori sono perciò in grado di eseguire algoritmi diversi. In questa categoria cadono anche i modelli per il calcolo distribuito nei quali la connessione tra i diversi processori avviene tramite una rete od una linea telefonica.

In realtà ai fini dei nostri discorsi è poco significativo il fatto che le istruzioni eseguite dai diversi processori siano uguali o diverse. Per questo durante il corso si è fatto riferimento principalmente al modello SIMD.

LA P-RAM

Come la RAM (Random Access Machine) costituisce il modello di calcolo astratto sequenziale, così le macchine parallele vengono modellate dalla **P-RAM** (Parallel Random Access Machine) che rappresenta per molti versi una generalizzazione della RAM.

Nella P-RAM si hanno a disposizione N processori, ognuno dotato di una CP e di una memoria locale. I processori condividono una memoria comune tramite la quale possono comunicare. Ogni processore può dialogare in ogni istante con qualsiasi altro in tempo costante. Questa assunzione nasconde in realtà i limiti e le difficoltà delle realizzazioni pratiche della P-RAM.

E' da notare inoltre come nel modello di calcolo astratto vengano completamente ignorati i tempi di accesso alla memoria secondaria.

Le operazioni che i processori sono in grado di eseguire sulle celle di memoria sono la **lettura** e la **scrittura**. Grazie alla pluralità di processori, è dunque possibile pensare che più unità vogliano accedere nello stesso istante alla stessa cella.

In base alle possibilità di accesso simultaneo da parte di più processori alla stessa cella possiamo identificare **quattro sottoclassi**:

- **EREW** (Exclusive-Read / Exclusive-Write) : Accesso alla memoria esclusivo. Non è possibile che due o più processori accedano in lettura o scrittura alla stessa cella di memoria contemporaneamente. Proprio per questo, il modello EREW è il piú realistico e semplice da costruire nella pratica.

- **CREW** (Concurrent-Read / Exclusive-Write) : Sono permesse letture concorrenti, ma non scritture concorrenti.

- **ERCW** (Exclusive-Read / Concurrent Write) : E' possibile che più processori scrivano contemporaneamente il contenuto di una cella, ma la lettura rimane esclusiva.

- **CRCW** (Concurrent-Read / Concurrent-Write) : Sono permesse sia letture che scritture multiple simultanee.

Se permettere letture multiple concorrenti non crea nessuna difficoltà, pensare alla possibilità di più processori che scrivono informazioni potenzialmente diverse nello stesso istante nella stessa cella di memoria pone problemi ben maggiori. Pertanto, in un modello che prevede una scrittura concorrente uno dei punti da affrontare è come determinare in modo univoco il contenuto della memoria al termine di una operazione di scrittura. Chiedere solamente una lettura concorrente (ma rinunciare ad una scrittura concorrente) significa in particolare permettere agli algoritmi che girano sulla macchina di conoscere il contenuto delle celle di memoria in ogni istante.

Perché ciò sia possibile, si utilizzano di norma delle regole che tutti i processori sono tenuti a rispettare. I **conflitti di scrittura** vengono cioè risolti tramite opportune convenzioni come le seguenti:

- **CW comune**: la scrittura è permessa solo se tutti i processori cercano di scrivere la stessa informazione nella locazione di memoria, altrimenti il valore della locazione resta invariato;

- **CW arbitrario**: la scrittura è permessa solo ad un processore, scelto arbitrariamente tra quelli in conflitto;
- **CW a priorità**: si assume che gli indici dei processori siano ordinati, e si permette la scrittura solo a quello di indice minimo (o massimo);
- **CW combinato**: il risultato (il valore della locazione di memoria dopo la scrittura) è una combinazione dei valori che i vari processori intendono scrivere, la combinazione è tipicamente una funzione associativa e commutativa, come ad esempio il massimo, il minimo, o la somma.

I modelli che permettono la concorrenza sono naturalmente più potenti di quelli ad accesso esclusivo, ma la realizzazione pratica di tali macchine è proibitiva, perché la complessità della gestione della concorrenza ne influenza negativamente le prestazioni.

Il modello EREW è dunque il più realistico da adottare per la descrizione di algoritmi, sebbene sia molto più debole degli altri, e implichi in genere una maggiore complessità nella fase di programmazione, poiché un algoritmo per questo modello deve essere progettato in modo da evitare ogni tentativo d'accesso concorrente.

TOPOLOGIE D'INTERCONNESSIONE

La P-RAM a memoria condivisa è un modello computazionale estremamente potente poiché, come detto sopra, consente ad un numero qualsiasi di processori di accedere in qualsiasi istante ad una cella della memoria condivisa contemporaneamente ed in tempo costante.

E' abbastanza intuitivo comprendere come questa possibilità sia estremamente difficile da garantire nella pratica, tanto che la P-RAM risulta effettivamente realizzabile solo per un piccolo numero di processori.

Ci sono però tecniche per aggirare le difficoltà costruttive. Diciamo aggirare perché tutti gli approcci proposti causano inevitabilmente una perdita di potenza computazionale per il modello che diventa più debole della P-RAM, ma anche più facilmente realizzabile.

E' tuttavia importante notare che ogni algoritmo pensato per la P-RAM continua ad essere eseguibile su questi modelli più poveri, anche se al costo di un numero di passi o di una occupazione di memoria maggiore.

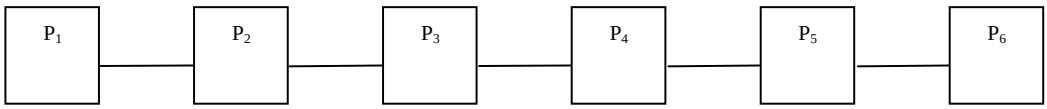
Abbiamo visto come, per far dialogare i processori, nelle SIMD esista un blocco funzionale incaricato della comunicazione che prendeva il nome di Memoria Condivisa / Rete di Interconnessione.

Solo permettendo ai processori di comunicare è infatti possibile distribuire in modo effettivo ed efficace il lavoro. Questa possibilità è offerta in modo totale da una **memoria condivisa.**

Tramite una **topologia di interconnessione** si introducono delle restrizioni sui processori che possono dialogare contemporaneamente, guadagnando però in costruttibilità. Questa strategia può essere impiegata dato che in molte applicazioni solo un piccolo sottinsieme delle connessioni tra tutte le coppie di processori è necessario per garantire buone prestazioni.

Le possibilità sono in questo caso numerosissime:

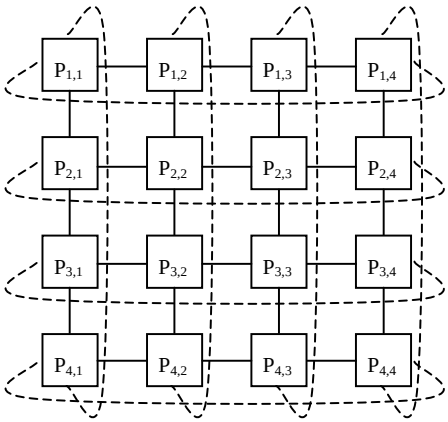
- **Vettore lineare** (ing. Array) è il modo più semplice per connettere N processori. In questo modo ogni processore comunica direttamente solo con il precedente ed il successivo (fatta eccezione di quelli esterni). Per far comunicare processori non adiacenti è perciò necessaria la collaborazione di tutti i processori intermedi.



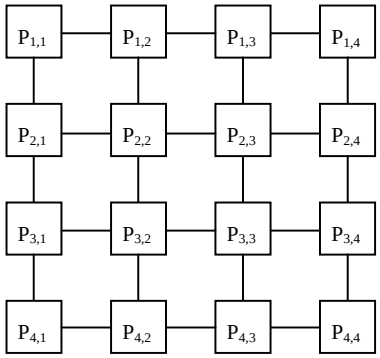
- **Matrice** (ing. Mesh) : si tratta di una rete bidimensionale in cui tutti i nodi hanno connettività 4, fatta eccezione per quelli al bordo. E' però possibile collegare fra di loro i nodi sul bordo in modo da avere connettività costante ed uguale a 4 per tutti.

La matrice che si ottiene in questo modo è detta toroidale.

Intendiamo con **dimensione della matrice** il numero di processori presenti su una riga o su una colonna.

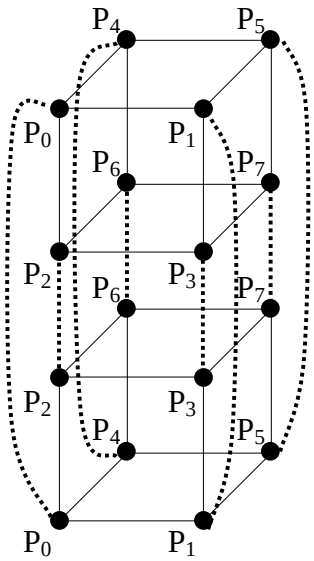


Matrice Toroidale



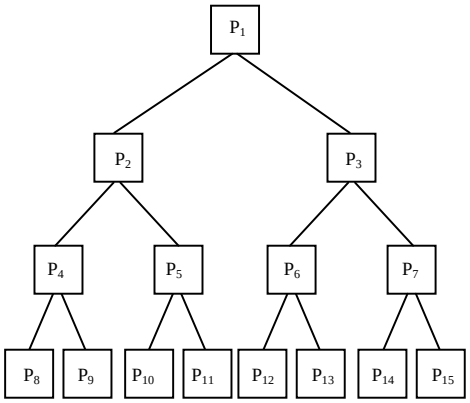
Matrice

- **N-Cubo (ipercubo)** : ogni processore comunica con n vicini se n è la dimensione del cubo. Un n-cubo si ottiene prendendo due (n-1)-cubi insieme e collegando fra loro i nodi corrispondenti. Uno 0-cubo è costituito da un solo processore.

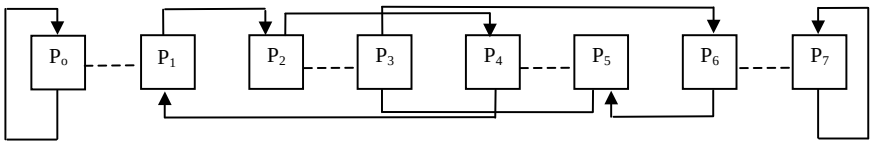


La figura a lato rappresenta un 4-Cubo, dove i segmenti a tratto continuo uniscono un 3-Cubo mentre quelli tratteggiati uniscono due 3-Cubo

- **Albero** : i processori vengono organizzati un una struttura gerarchica ad albero binario completo. Ogni processore è connesso ai suoi due figli al livello precedente ed al padre posto al livello successivo. La radice è un nodo senza padre, mentre le foglie sono nodi privi di figli. Si pensa ad una struttura bilanciata perché questo garantisce una profondità logaritmica nel numero di foglie (o di nodi, poiché l'ordine di grandezza non cambia). Rinunciando al bilanciamento si potrebbe degenerare verso un'vettore lineare di profondità lineare.



- **Shuffle** : le connessioni sono ottenute codificando in binario gli indici dei processori e connettendo ogni nodo con i nodi il cui indice si ottiene spostando (operazione di "shift") verso sinistra di una posizione la codifica del nodo in esame. Oltre a queste connessioni che sono unidirezionali, a volte si aggiungono collegamenti bidirezionali tra tutti i nodi di indice pari ed i loro successori.



Come è possibile notare, in questi esempi non è più vero che tutti i nodi siano in grado di comunicare in tempo unitario, al contrario di quanto avveniva nella P-RAM a memoria condivisa.

La profondità della struttura è particolarmente importante, poiché da essa dipende il tempo richiesto dalla rete per risolvere i problemi. In un modello a rete di interconnessione infatti, come detto in precedenza, ogni nodo può comunicare direttamente (cioè a distanza

uno) solo con un sottoinsieme dei nodi della rete. Perché sia possibile uno scambio di informazione tra nodi non adiacenti è pertanto necessaria la collaborazione dei nodi intermedi. Questi dovranno fare da tramite colmando la distanza tra i nodi che intendono comunicare. A questo punto però, perché i dati arrivino a destinazione, è necessario attendere un certo numero di passi. Ciò si tramuta in un peggioramento per i tempi di esecuzione di tutti gli algoritmi che lavorano sul modello.

Simulazione di una P-RAM CRCW con una P-RAM EREW

Con un modello EREW è possibile simulare algoritmi CRCW, pagando un aumento delle richieste di spazio e di tempo.

SIMULAZIONE DI LETTURA CONCORRENTE SU UN MODELLO A LETTURA ESCLUSIVA

Un'istruzione di lettura concorrente di una P-RAM CR in cui N processori leggono da una medesima locazione di memoria può essere implementata su una P-RAM ER per essere eseguita in tempo O(log N).

Il seguente algoritmo di broadcast (EREW) realizza la simulazione.

Algoritmo di simulazione della lettura concorrente

Sia x il valore contenuto nella locazione I della memoria condivisa, e siano $P_1, P_2, \ldots, P_N$ i processori che richiedono di leggere la locazione I; ogni processore ha a disposizione una locazione L nella propria memoria locale; A è un vettore ausiliario di lunghezza N nella memoria condivisa;

```
Passo 1. P1: L ← x
             A[1] ← L

Passo 2. for  i = 0  to  ( log N – 1)  do
           for  j = 2^i + 1  to  2^(i+1)  pardo
             Pj : L ← A[ j – 2^i ]
                  A[ j ] ← L
.
```

Al passo 1, P_1 legge x e lo scrive in A[1], P_1 è l'unico processore ad accedere realmente alla locazione I.

Il passo 1 ha tempo d'esecuzione costante.

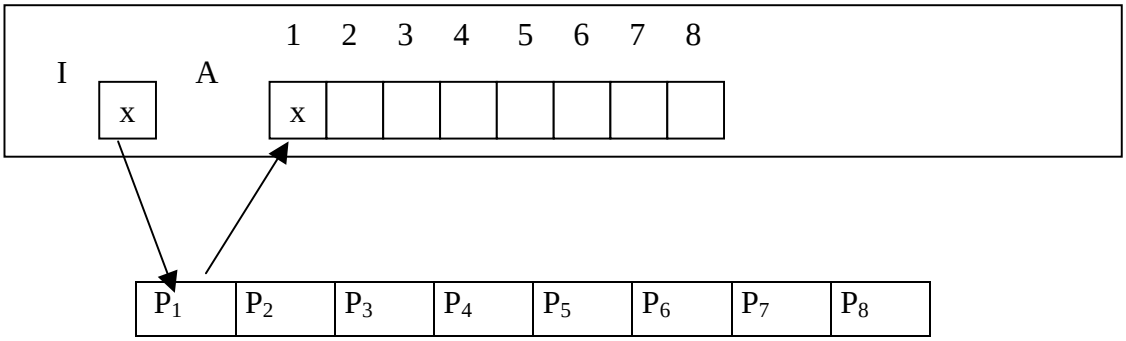


Figura 1: la situazione dopo il passo 1

Alla prima iterazione del passo 2, quando i vale 0, l'unico processore che lavora è P_2, che legge A[1] e scrive x in A[2];

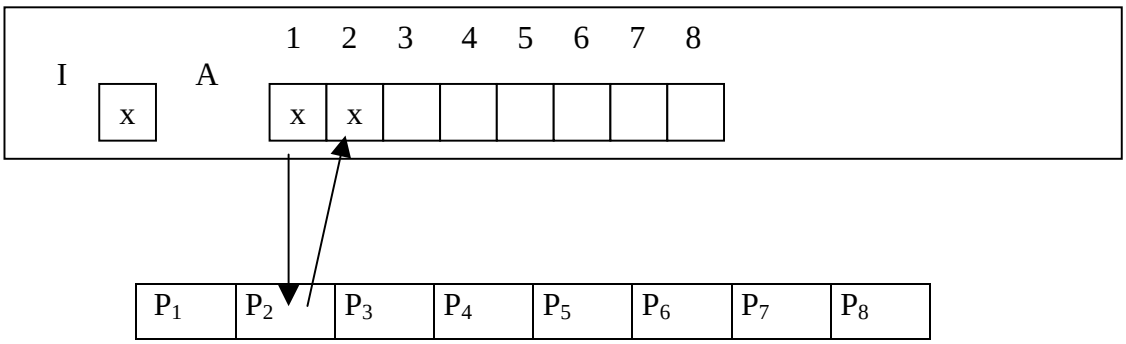


Figura 2: la prima iterazione del passo 2

Alla seconda iterazione, lavorano in parallelo i processori P_3 e P_4, che leggono rispettivamente A[1] e A[2].

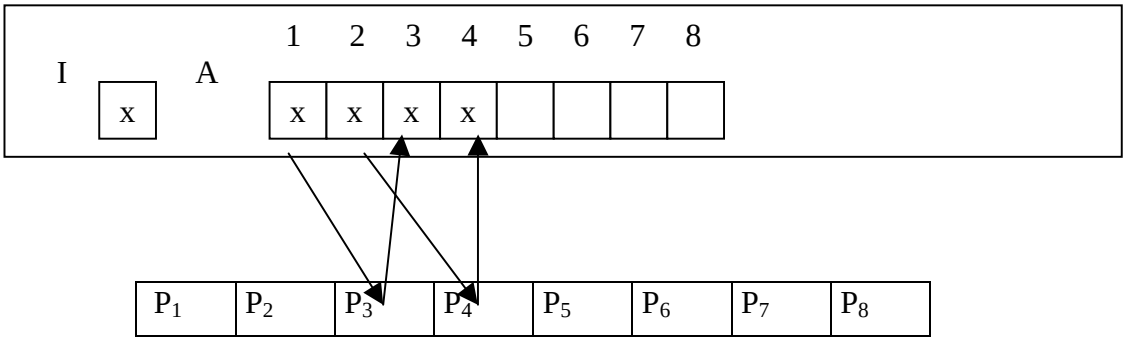


Figura 3: la seconda iterazione del passo 2

Alla successiva iterazione lavorano i processori da P_5 a P_8 e così via; dopo log N iterazioni del passo 2 ogni processore ha caricato x nella propria memoria locale. Il corpo del ciclo for … do del passo 2, il ciclo parallelo, è eseguito in tempo costante, quindi la complessità dell'algoritmo è O(log N).

SIMULAZIONE DI SCRITTURA CONCORRENTE SU UN MODELLO A SCRITTURA ESCLUSIVA

Un'istruzione di scrittura concorrente di una P-RAM CW comune, in cui N processori scrivono in una medesima locazione di memoria può essere implementata su una P-RAM EW per essere eseguita in tempo O(log N).

Il modello CW comune permette la scrittura solo se tutti i processori cercano di scrivere lo stesso valore.

Algoritmo di simulazione della scrittura concorrente (CW comune)

Sia R una locazione della memoria condivisa, e siano $P_1, P_2, \ldots, P_N$ i processori che richiedono di scrivere in R rispettivamente i valori $a_1, a_2, \ldots, a_N$

A è un vettore ausiliario di lunghezza N,

B è un vettore di booleani lungo N,

entrambi sono nella memoria condivisa.

```
Passo 0. for i = 1  to  N  pardo
  P_i : B[ i ] = T
        A[ i ] = a_i

Passo 1. for i = 1  to  log N  do
           for  j = 1  to   N/ 2^i  pardo
            P_j :
              if (B[j]≠T  or  B[ j + N/ 2^i ]≠T  or  A[j]≠A[ j + N/ 2^i ])
              then  B[j] ← F

Passo 2.  if   B[1] = T   then  P_1:  R ← a_1
```

Il passo 0 è un'inizializzazione in parallelo dei vettori A e B.

Alla prima iterazione del passo 1, la prima metà dei processori è attivata in parallelo, il processore P_j confronta A[j] e A[j + N/ 2] , se sono diversi mette a false B[j], il che vuol dire che la scrittura non potrà andare a buon fine.

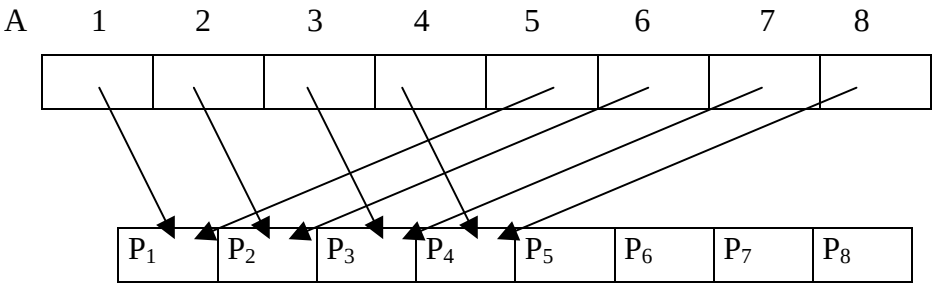


Figura 4: il passo generico dell'algoritmo

Ad ogni passo P_j controlla anche B[j] e B[j + N/2^i]; alla prima iterazione hanno entrambi valore T, perché sono stati appena inizializzati, ma successivamente nel vettore B ci sarà un valore F se sono stati trovati due elementi di A differenti, e questa informazione deve essere trasmessa fino in fondo, fino a B[1], che dopo l'ultima iterazione del passo 1 avrà valore T solo se tutti i valori da scrivere (gli A[i]) sono uguali.

La caduta di una qualunque delle tre condizioni dell' if del passo 1 implica che la scrittura non può andare a buon fine, perché o in passato sono stati trovati due valori diversi (se B[j] o B[j + N/ 2^i] è Falso) , o A[j] ≠ A[j + N/ 2^i].

Ad ogni iterazione si dimezza il numero dei processori che lavorano, quindi dopo log N iterazioni si esce dal ciclo for e viene eseguito il passo 2, in cui il processore P1 scrive in R se non si sono verificati problemi, cioè se i valori da scrivere sono tutti uguali; l'algoritmo ha dunque complessità O(log N).

Questo algoritmo si adatta facilmente anche alla simulazione di una P-RAM CW combinata.

Algoritmo di simulazione della scrittura concorrente (CW combinata)

Se si vuole simulare una P-RAM che scrive in R il massimo degli a_i, basta modificarlo leggermente, non c'è neanche più bisogno del vettore di booleani, perché la scrittura è sempre eseguita:

```
Passo 0.   for i = 1  to  N  pardo
             P_i  : A[ i ] = a_i

Passo 1.   for i = 1  to  log N  do
             for  j = 1  to   N/ 2^i  pardo
               P_j : A[j] ←  max( A[j] , A[j + N/ 2^i]

Passo 2.  P_1: R←  A[1]
```

Alla fine in R risulterà il massimo fra tutti gli a_i.

L'estensione anche al caso del minimo o della somma è immediata.

CASO GENERALE

Si può estendere la simulazione al caso più generale in cui N processori eseguano un accesso concorrente alla memoria condivisa, ma non necessariamente tutti accedano alla stessa locazione di memoria, per ottenere il seguente risultato:

Un' istruzione di lettura o scrittura concorrente di una P-RAM CRCW con N processori può essere implementata su una P-RAM EREW con N processori per essere eseguita in tempo O(log N).

La dimostrazione di questo risultato consiste in una prova di simulazione che fa uso di un algoritmo EREW per ordinare k numeri in tempo O(log k) con k processori.

Per un approfondimento sugli algoritmi di ordinamento si veda [6]

Segue un algoritmo per simulare una scrittura concorrente di una P-RAM CRCW a priorità con N processori su un modello EREW.

Sia l_i la locazione della memoria condivisa in cui il processore P_i intende scrivere un dato x_i , con i = 1,2,…..N

Sia M un vettore di lunghezza N in memoria condivisa.

```
Passo 1.  for i = 1 to N pardo
             Pi  :  M [ i ] ← ( li  , xi  )
Passo 2.  ordina  M  in ordine non decrescente
          rispetto al primo elemento della
          coppia ( li )
Passo 3.  for i = 1 to N pardo
             P1 : ( j , d ) ← M [1]
                  j ← d
             Pi :  ( j1  , d1 ) ← M[i]
                   ( j2  , d2 ) ← M[i-1]
                   if j1 ≠ j2    then  j1 ← d1
```

Il passo 1 è un'inizializzazione del vettore M, in cui si memorizzano le coppie "(locazione, dato)" dalle scritture di ogni processore;

in figura (**Figura 5**) è rappresentata la situazione dopo il passo 1 in un esempio con valori arbitrari di locazioni e dati: la coppia (25, 6) in M[1] dopo il primo passo sta ad indicare la richiesta di P_1 di scrivere il dato 6 nella locazione 25 della memoria condivisa

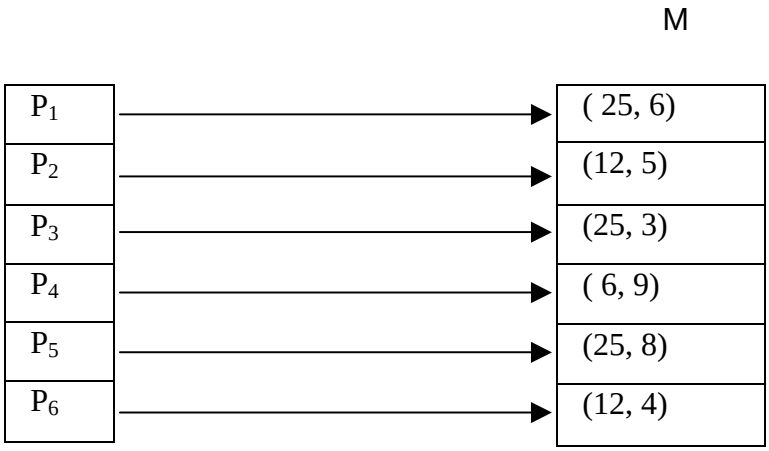


Figura 5: la situazione dopo il passo 1

Il passo 2 ordina M rispetto alla prima componente in tempo O(log N):

(6, 9)
(12, 5)
(12, 4)
(25,6)
(25, 3)
(25,8)

Figura 6: ordinamento del vettore nel passo 2

Il vettore può essere visto come suddiviso in blocchi di elementi che hanno la prima componente della coppia uguale; ognuno di questi blocchi rappresenta un insieme di

richieste di scrittura su una stessa locazione di memoria. Nell'esempio si individuano tre blocchi, sono da effettuare pertanto tre scritture in memoria, rispettivamente nelle celle 6, 12 e 25.

Un modello CRCW a priorità permette la scrittura solo al processore di indice minimo fra quelli in conflitto, per la simulazione c'è dunque necessità di selezionare una sola fra le richieste di scrittura in ogni blocco.

Siano P_i e P_j , con i < j, due processori in conflitto per una scrittura nella locazione l; dopo il passo 1 la coppia (l , x_i) si troverà in M chiaramente prima della coppia (l , x_j), e dunque anche dopo l'ordinamento (passo 2), che lascia invariato l'ordine relativo fra due elementi uguali, si troverà (l , x_i) prima di (l , x_j) in M; all'interno di un blocco si troverà dunque per prima la coppia corrispondente alla richiesta di scrittura del processore di indice minimo fra quelli in conflitto.

Il passo 3 dell'algoritmo realizza la selezione del primo elemento di ogni gruppo, che esegue la scrittura: ogni processore P_i controlla in tempo costante se l'elemento i-esimo di M è all'inizio di un blocco confrontandolo con quello precedente.

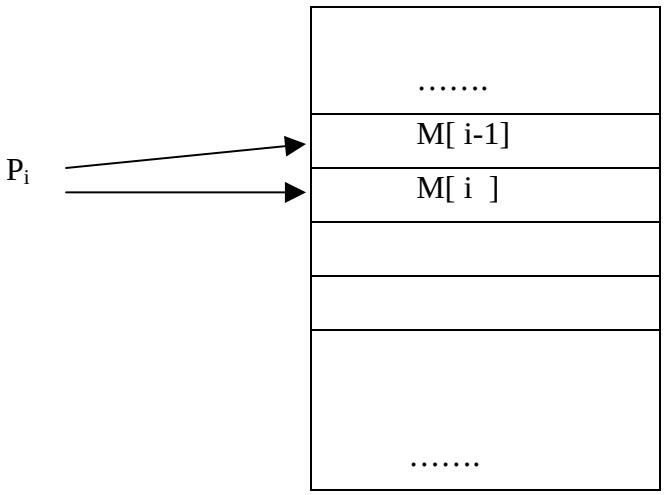


Figura 7: selezione del primo elemento di ogni gruppo

Se si tratta di due accessi a locazioni diverse, allora l'i-esimo elemento è il primo di un nuovo blocco e dunque rappresenta la richiesta di scrittura di un processore prioritario e viene eseguita.

Si noti che non è il processore che ha richiesto la scrittura che la esegue praticamente.

Nell'esempio:

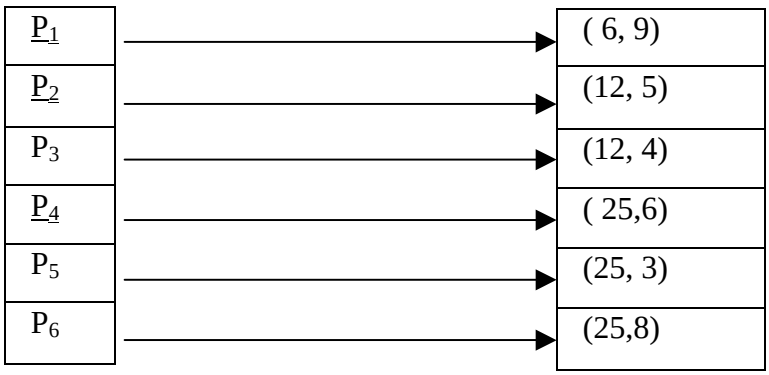


Figura 8: attivazione dei processori 1,2 e 4

I processori 1,2 e 4 si attivano ed eseguono la scrittura in memoria:

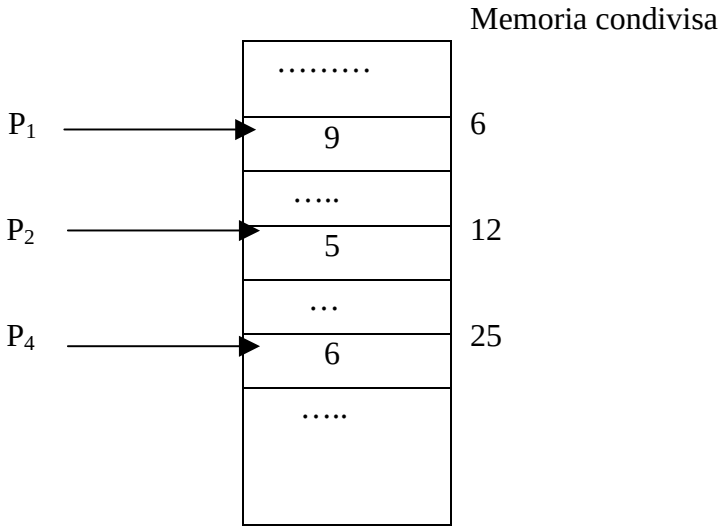


Figura 9: scrittura in memoria

I passi 1 e 3 richiedono tempo costante; il passo 2, l'ordinamento, può essere eseguito in tempo O(log N) su un modello EREW. L'algoritmo ha dunque tempo d'esecuzione logaritmico.

La stessa idea può essere sfruttata per simulare la lettura concorrente: il processore P_i scrive in M[i] la locazione di memoria dalla quale vuole leggere, e il proprio indice per poter ricevere in seguito l'informazione. Il vettore M viene ordinato e suddiviso in blocchi, per poi attivare una lettura esclusiva in memoria per ogni blocco; l'algoritmo termina con l'esecuzione di un broadcast opportunamente modificato, effettuato in parallelo da ogni blocco.

- sia l_i la locazione della memoria condivisa da cui il processore P_i intende leggere, con i = 1,2,…..N.
- sia M un vettore di lunghezza N in memoria condivisa,
- siano Blocco[1…N] un vettore di interi,
- Attivo[1…N] un vettore di booleani,
- A[1…N] un vettore d'appoggio, tutti in memoria condivisa

```
Passo 1.  for i = 1 to N pardo
             Pi  :  M [ i ] ← ( li  , i )

Passo 2.  ordina  M  in ordine non decrescente
          rispetto al primo elemento della coppia ( li )

Passo 3.  for i = 1 to N pardo
            P1 :  Blocco[1] ← 1
                  Attivo[1] ← T
                  ( j , k ) ← M[1]
                  A[1] ← contenuto della locazione di memoria j;
```

```
            Pi :  ( j1  , h  ) ← M[i]
                  ( j2  , k  ) ← M[i-1]
                  if j1 ≠ j2   then
                       Blocco[i]  ← 1
                       Attivo[i] ← T
                       A[i] ← contenuto della locazione di memoria j1;
                  else
                       Blocco[i] ← 0
                       Attivo[i] ← F

Passo 4.  Somme prefisse sul vettore Blocco

Passo 5.  for k = 0 to (logN – 1) do
             for i = 1 to N pardo
               Pi : if  Attivo[i] = T then
                        if (i+ 2^k ≤ N ) and Blocco[i] = Blocco[i+2^k] then
                             A[i+2^k] ← A[i]
                             Attivo[i+2^k] ← T
                        else Attivo[i] ← F

Passo 6. for i = 1 to N pardo
           Pi : ( j, k) ← M[i]
                 Pk : legge A[i]
.
```

Il Passo 1 è l'inizializzazione del vettore M. In figura (**Figura 10**) è rappresentata la situazione dopo il passo 1 in un esempio.

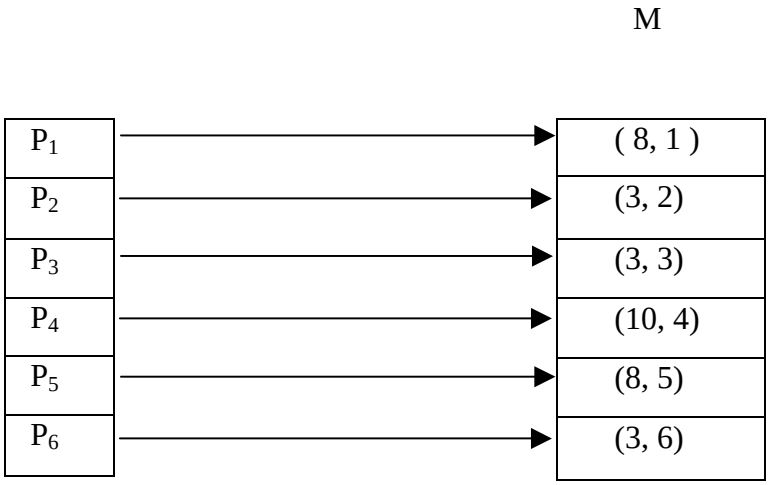


Figura 10: situazione dopo il passo 1

Dopo l'ordinamento effettuato al passo 2:

(3, 2)
(3, 3)
(3, 6)
(8, 1)
(8, 5)
(10, 4)

Figura 11: ordinamento

Il passo 3 realizza l'individuazione del primo elemento di ogni blocco allo stesso modo del precedente algoritmo.

Se l'i-esimo elemento è il primo del suo gruppo, il processore P_i scrive 1 in Blocco[i], altrimenti scrive 0; successivamente viene applicato l'algoritmo "Somme prefisse" (a tal proposito si veda [4]) al vettore Blocco. Alla fine del passo 4, per ogni i , in Blocco[i] si troverà l'indice del gruppo a cui appartiene l' i-esimo elemento.

Al passo 3 viene anche inizializzato il vettore booleano ausiliario Attivo: Vero per gli elementi all'inizio di ogni blocco, Falso per gli altri.

Inoltre per ogni blocco il processore associato al primo elemento effettua la lettura in memoria (esclusiva), e copia il valore trovato nel vettore ausiliario A.

Tornando all'esempio, e supponendo di avere nelle locazioni di memoria condivisa 3, 8 e 10 rispettivamente i valori x, y e z, la situazione dopo il passo 4 è la seguente:

M	Blocco	Attivo	A
(3, 2)	1	T	x
(3, 3)	1	F	
(3, 6)	1	F	
(8, 1)	2	T	y
(8, 5)	2	F	
(10, 4)	3	T	z

Figura 12: situazione dopo il passo 4

Il passo 5 consiste in un ciclo che esegue materialmente il broadcast, in parallelo per ogni blocco, facendo uso delle informazioni contenute nei vettori ausiliari.

Ad ogni iterazione del ciclo i processori P_i tali che Attivo[i] è vero effettuano una copia del valore A[i] in una nuova locazione usando la tecnica di broadcast vista nel primo algoritmo di simulazione di lettura concorrente, effettuando però i dovuti controlli, che impediscono di invadere lo spazio di un altro blocco (Blocco[i] = Blocco[$i+2^k$]) e di eccedere le dimensioni del vettore ($i+2^k \leq N$).

Nell'esempio alla prima iterazione si attivano i processori P_1 , P_4 e P_6, dei quali solo i primi due effettuano la copia, mentre il terzo si disattiva.

La situazione dopo la prima iterazione è la seguente:

M	Blocco	Attivo	A
(3, 2)	1	T	x
(3, 3)	1	T	x
(3, 6)	1	F	
(8, 1)	2	T	y
(8, 5)	2	T	y
(10, 4)	3	F	z

Figura 13: situazione dopo la prima iterazione del passo 5

Alla seconda iterazione (per k = 1) si attivano i processori P_1 , P_2 , P_4 e P_5 , dei quali però solo P_1 effettua realmente l'operazione, perché per gli altri risultano falsi i controlli; P_1 copia A[1] in A[3] e il broadcast è completato. Il numero di passi necessari a completare la trasmissione è logN, perché ogni blocco la effettua in parallelo e la dimensione di un blocco è limitata da N.

Il passo 6 effettua la lettura in parallelo (esclusiva) del vettore A.

La complessità dell'algoritmo è dell'ordine di logN, che è la complessità dei passi 2 , 4 e 5; i passi 1, 3 e 6 hanno tempo d'esecuzione costante.

La simulazione di altri modelli a scrittura concorrente (comune e combinato) sfrutta le stesse idee ed è derivabile direttamente dagli algoritmi proposti.

Da quanto dimostrato segue che:

dato un algoritmo per una P-RAM CRCW con N processori che ha tempo d'esecuzione T,

tale algoritmo può essere implementato su una P-RAM EREW con N processori per essere eseguito in tempo O(T log N).

Simulazione Broadcast su altri Modelli

Vedremo in questo paragrafo l'algoritmo di broadcast applicato a tre esempi di macchine parallele a **Reti Interconnesse**: albero, matrice, ipercubo. Per simulare la lettura concorrente si usa la tecnica accennata nel paragrafo precedente.

ALBERO

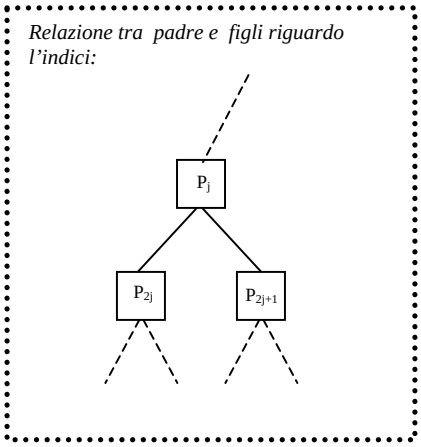
Sfruttiamo nell'algoritmo la seguente idea: ogni nodo, se padre (nell'algoritmo ipotizziamo che l'albero sia binario, senza perdere di generalità), si occupa di trasmettere ai propri figli il valore dell'informazione precedentemente ricevuta.

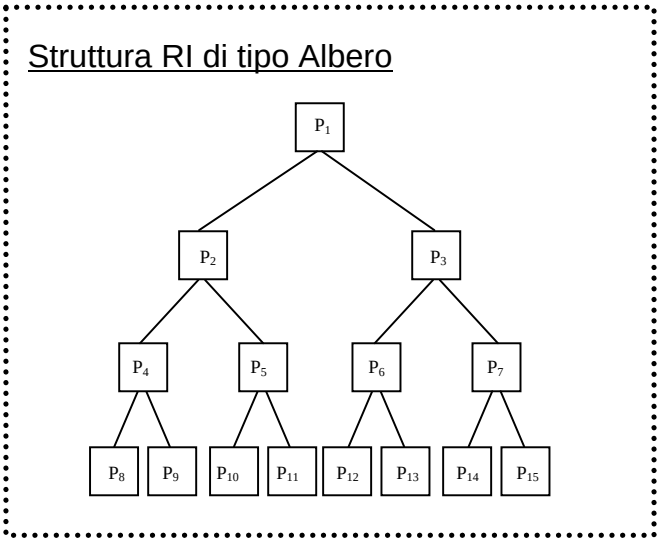
Per questo ad ogni passo si attivano in parallelo un numero di processori pari al numero di nodi del livello dell'albero su cui l'algoritmo sta lavorando. I processori trasferiscono in blocco l'informazione al livello sottostante. Iteriamo il procedimento appena citato sino al raggiungimento della base dell'albero, il tutto in un Tempo= O(log N) che corrisponde all'altezza dell'albero.

Algoritmo in pseudo linguaggio:

```
Procedure Broadcast (X, N)
Step1:  # processor P1#
    C1 ← read (X)
Step2: For i = 0 to ⌊log N⌋ –1 do
        For j = 2^i to 2^(i+1)-1 pardo
          # processor Pk=2j#
          Ck ← Cj
          Ck+1 ← Cj
End Procedure
```

N = "numero di processori" = 7
X = "locazione di mem. contenente l'elemento da leggere"





MATRICE

In questo modello l'informazione si distribuisce, non più "a macchia d'olio" come nell'albero, ma prima in una direzione e poi nell'altra.

Più formalmente la procedura può pensarsi suddivisa in due passi:

- Il primo si occupa di trasmettere l'informazione dall'angolo sup. sinistro all'angolo sup. destro della matrice attraverso una riga. Questo passo è intrinsecamente seriale, perché un solo processore può accedere in memoria centrale, l'implementazione è tuttavia svolta con una tecnica parallela (non alterando i tempi di esecuzione) per comodità.

- Nel secondo passo si può trasferire parallelamente l'informazione da una riga alla successiva fino ad arrivare all'ultima.

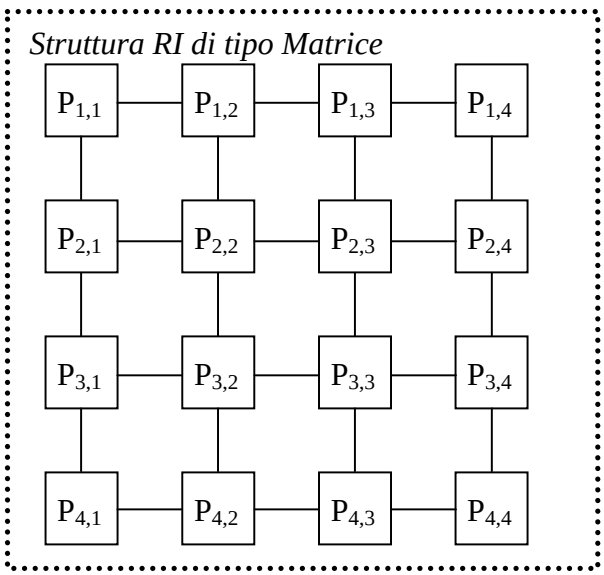
Il tempo totale è dato dalla somma dei due passi t=O(#colonne)+O(#righe).

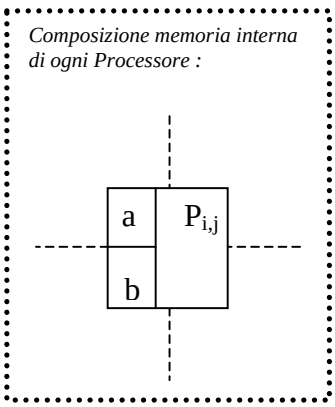
Algoritmo in pseudo linguaggio:

```
Procedure Broadcast ( r, c, X)
Step1: For i = 1 to c do
        For j = 1 to c pardo
          # processor P1,j#
          if (j=1) then a1,j ← read(X)
          else a1,j ← b1,j-1
        b1,j ← a1,j
Step2: For i = 2 to r do
      For j = 1 to c pardo
        # processor Pk=(i*c)+j#
          ai,j ← bi-1,j
          bi,j ← ai,j
End Procedure
```

```
Step1 (seriale):
    a1,1←read(X)
    b1,1←a1,1
    For j = 2 to c pardo
      #  processor  P1,j#
      a1,j      ←      b1,j-1
      b1,j ← a1,j
```

N = "numero di processori" = c*r=9
X = "locazione di mem. contenente l'elemento da leggere"
c = "# di colonne"
r = "# di righe" r = c per IP.





L'algoritmo appena presentato consente di distribuire a tutti i processori una stessa informazione. In realtà possiamo pensare alla distribuzione di una informazione diversa per ognuno di essi.

Vediamo allora un'algoritmo che fornisce questa possibilità in tempo = O(r*c)

Algoritmo in pseudo linguaggio :

```
Procedure Broadcast ( N, X)
  For k = 1 to r do
    For i = 1 to c do
      For j = 1 to c pardo
        # processor P1,j#
        if (j=1) then a1,j ← read(X[(k-1)*r+i])
        else a1,j ← b1,j-1
        b1,j ← a1,j
    if k < r
      then {
            For i = 2 to r pardo
              For j = 1 to c pardo
               # processor Pi,j#
               ai,j ← bi-1,j
               bi,j ← ai,j
          }
End Procedure
```

N = "numero di processori"
X = "vettore di N elementi distinti da leggere"
c = "# di colonne"
r = "# di righe" r = c per IP.

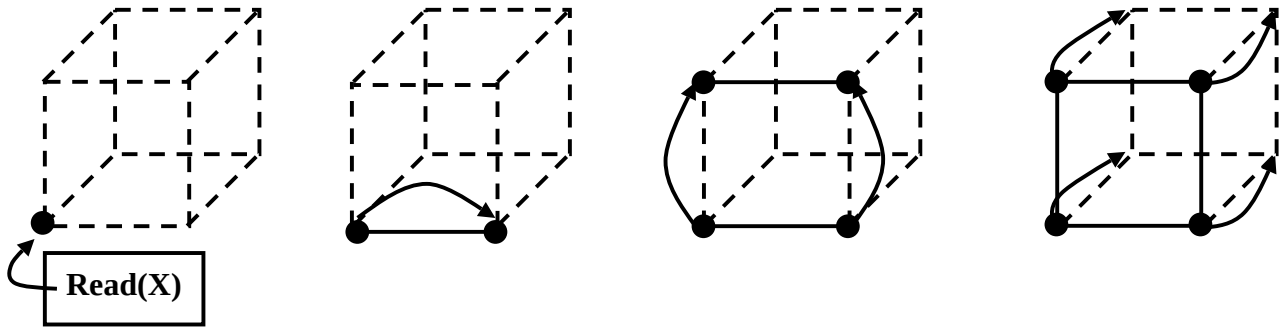
O(c)

O(r)

O(1)

I PERCUBO

Qui l'espandersi dell'informazione si muove come nel caso di P-RAM solo occorre fare attenzione alla numerazione degli spigoli del cubo, cioè i processori. In pratica ogni volta che ci muoviamo su una dimensione diversa mai visitata, i processori precedentemente attivati operano in parallelo. In questo modello, si vede bene dalla figura sotto, quali processori si attivano di volta in volta e come si muove l'informazione.



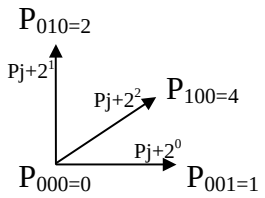
Per quanto riguarda gli indici degli spigoli vedere l'esempio successivo all'algoritmo per comprendere meglio i passaggi dell'algoritmo. Il tempo totale è dato dall'ultimo "step" cioè t=O(log N) visto che il primo è banale.

Algoritmo in pseudo linguaggio:

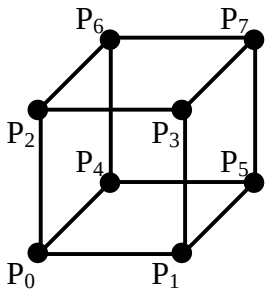
```
Procedure Broadcast ( N)
Step1:  # processor P0#
        C0 ← read(X)
Step2: For i = 0 to log N -1 do
            For j = 0 to 2^i-1 pardo
              # processor Pj#
              k   ← 2^i
              Cj+k ← Cj
End Procedure
```

N = "numero di processori" = 8
X = "locazione di mem. contenente l'elemento da leggere"

Relazione tra gli indici,codificati in binario, nelle tre dimensioni :



Struttura RI di tipo Ipercubo (n=3)



Teorema di Brent

Il teorema di Brent mostra che una PRAM può simulare con una data complessità un generico circuito combinatorio. Utilizzando questo teorema, è possibile sfruttare i risultati ottenuti nei circuiti combinatori (es: le reti di ordinamento), per adattarli al contesto del calcolo parallelo tramite PRAM. Si ottiene un limite superiore della complessità del generico algoritmo, che andrebbe a simulare su PRAM il circuito combinatorio.

CIRCUITI COMBINATORI

Prima di introdurre il teorema di Brent è necessario ricordare cosa si intenda per circuito combinatorio e quali siano le caratteristiche basilari di ogni circuito combinatorio.

Circuito combinatorio

Un circuito combinatorio è una rete aciclica di elementi combinatori

Particolari circuiti combinatori sono gli addizionatori: “half-adder”, “full-adder”. Altri particolari circuiti combinatori sono le reti di confrontatori e di ordinamento.

Per ulteriori informazioni sulle reti di ordinamento si veda [1]

Nella definizione si evidenziano due punti: l’elemento combinatorio, l’ interconnessione tra questi elementi tramite una rete aciclica.

Un elemento combinatorio è un elemento-funzione che dato un numero fisso di input calcola un numero fisso di output. In termini matematici è come se fosse una funzione $f:I \rightarrow O$, che ad ogni elemento ‘i’ dell’ insieme di input I, associa un e un solo elemento ‘o’ dell’insieme O degli output.

Particolari elementi combinatori sono gli elementi combinatori binari o porte logiche: “and”, “or”, “not”, … . In questo caso input e output hanno valori in {0,1}. Una porta

logica può essere facilmente descritta mediante una tabella di verità. Altri particolari elementi combinatori sono i confrontatori.

Per ulteriori informazioni sui confrontatori si veda [1]

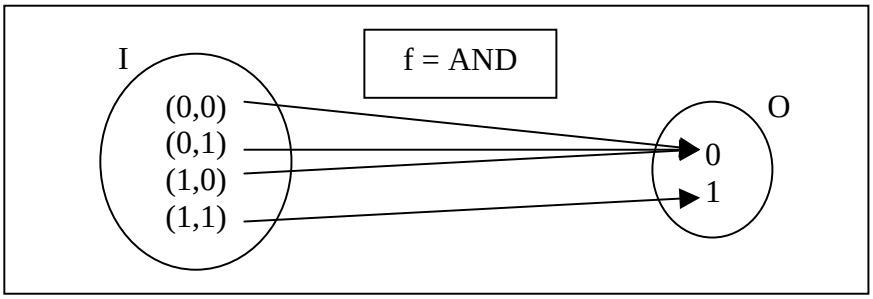
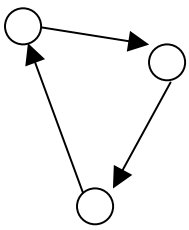


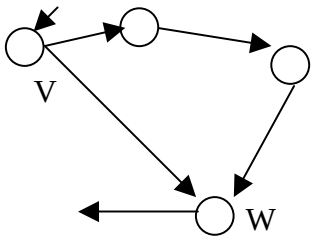
Figura 1: funzione f della porta logica AND

Il concetto di aciclicità per una rete combinatoria è analogo al concetto di aciclicità per grafi diretti. Dunque con rete aciclica si intende che il grafo del circuito combinatorio non ha un ciclo diretto, ovvero un ciclo in cui tutti gli archi hanno lo stesso orientamento. Gli archi del grafo sono orientati secondo il percorso obbligatorio dettato dalla funzione degli elementi combinatori, che parte dagli input applica una computazione e arriva agli output. Dunque se il grafo è aciclico, allora non è possibile entrare in un elemento combinatorio C , seguirne il percorso degli output, attraversare altri elementi combinatori e ritornare in C.

Il grafo di sinistra è un grafo diretto che ha un ciclo e pertanto non può essere un circuito



combinatorio. Il grafo diretto di destra non ha alcun ciclo diretto. Anche se non si sa da dove venga l'arco incidente su V e dove vada a finire l'arco uscente da W. Per risolvere questo problema in alcuni casi



vengono introdotti degli speciali elementi I ed O detti rispettivamente elementi di input e di output del circuito. Gli elementi I non hanno archi entranti, mentre gli elementi O non hanno archi uscenti. Per mantenere una esatta corrispondenza con il concetto di grafo diretto, nel disegno dovremmo avere un elemento I da cui parte un arco verso V e un elemento O che raccolga l'output di W. (Questa

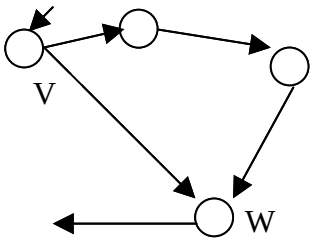
notazione non verrà utilizzata; inoltre il verso degli archi verrà usato solo dove necessario per evitare ambiguità.)

Nel disegno si nota inoltre che il vertice V ha la particolarità di “servire” con il proprio output due distinti vertici. Mentre sul vertice W “vanno a finire” gli output di due diversi elementi. Questo comportamento è possibile in un grafo diretto, come anche in un circuito combinatorio e viene spiegato con l’introduzione del concetto di fan-in e fan-out e la relazione che c’è tra questi è gli input e gli output di un elemento combinatorio.

Fan-in e fan-out di un elemento combinatorio

Il fan-in è il numero di input di un elemento combinatorio. In un circuito questo corrisponde al numero di collegamenti (o fili) in ingresso all’elemento.

Il fan-out è il numero di “posti” dove gli output dell’elemento combinatorio vanno a finire. Non è detto che gli output di un elemento combinatorio siano uguali al suo fan-out, infatti un elemento combinatorio con un solo filo di output può “servire” un numero qualunque di altri elementi.



Per semplicità consideriamo ogni arco come un unico filo che può assumere valori su {0,1} e supponiamo che ogni elemento combinatorio abbia un solo output. L’elemento V ha un solo input e dunque fan-in uguale a 1. A questo punto, poiché V ha come output valori su {0,1}, allora esso ha un solo output, che però va a finire su due diversi elementi, dunque V ha fan-out uguale a 2. L’ elemento W ha fan-in uguale a 2 e fan-out uguale a 1. Gli altri elementi hanno fan-in e fan-out uguale a 1.

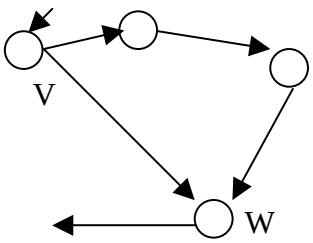
Dimensione e profondità di un circuito combinatorio

La dimensione di un circuito combinatorio è il numero di elementi combinatori che il circuito possiede.

Poco più complessa è la definizione di profondità.

Per definire la profondità di un circuito bisogna prima definire la profondità di un singolo elemento combinatorio. Questa è il numero di elementi combinatori contenuti nel più lungo cammino, che da un input del circuito mi porta all'output dell'elemento combinatorio.

Detto questo, la profondità di un circuito combinatorio è la massima profondità degli elementi combinatori che il circuito possiede.



La dimensione di questo circuito è 4.

La profondità di V è 1, infatti l'unico elemento combinatorio che incontro per arrivare a V è V stesso.

La profondità di W è 4, che è anche la massima profondità degli elementi del circuito.

Dunque la profondità del circuito è 4.

TEOREMA DI BRENT

Assunzioni

Nel teorema di Brent varranno le seguenti assunzioni:

- il fan-in di ogni elemento del circuito è superiormente limitato da una costante, dunque fan-in $\in$ O(1);
- ogni elemento combinatorio ha esattamente un output; infatti elementi con k (>1) output possono essere visti come k elementi distinti, ognuno dei quali calcola uno degli output dell'elemento originale;
- il fan-out di ogni elemento del circuito può essere anche illimitato. Infatti come precisato in precedenza il fatto che un elemento abbia un solo output non impedisce che esso vada a servire più di un elemento.

Teorema (di Brent)

Un circuito combinatorio di profondità d e dimensione n i cui elementi hanno un fan-in limitato, può essere simulato su una PRAM CREW con p processori da un algoritmo in tempo O(n/p + d).

Dim.:

Costruzione del circuito

Gli input del circuito vengono memorizzati in una apposito vettore in memoria. Questo vettore ha dimensione finita perché il fan-in è limitato.

Ogni elemento combinatorio del circuito è una particolare struttura nella memoria globale della PRAM, che tiene nota delle locazioni di memoria in cui è possibile reperire i valori di input all'elemento e ha a disposizione una locazione di memoria per scrivere l'output dell'elemento. In questo modo ogni elemento può essere simulato da un unico processore. Il processore legge i valori di input dalle opportune locazioni, calcola la funzione dell'elemento e scrive il risultato nella locazione di output.

Osservazioni

È possibile che due processori, che simulino due diverse porte in parallelo, vogliano accedere simultaneamente allo stesso input presente in memoria. In questo caso è necessario la lettura concorrente (CR).

C'è una diversa locazione di output per ogni diverso elemento del circuito dunque la scrittura dell'output viene effettuata in modo esclusivo (EW).

La funzione di un elemento combinatorio è semplice e poiché il suo fan-in è limitato, l'output può essere calcolato da un singolo processore in tempo costante ($\forall$ elemento $\in$ circuito $T_{elemento} \in O(1)$).

Posizionando visivamente gli elementi del circuito dall' alto verso il basso (**Figura 2**), in modo che elementi con la stessa profondità siano allo stesso livello, si nota che la simulazione di un elemento di livello 'i' deve essere preceduta da quella degli elementi ai livelli precedenti. Solo in questo modo tutti i suoi input saranno disponibili. Per elementi

allo stesso livello non sussiste invece alcuna dipendenza. Dunque il parallelismo può essere sfruttato solo per elementi allo stesso livello.

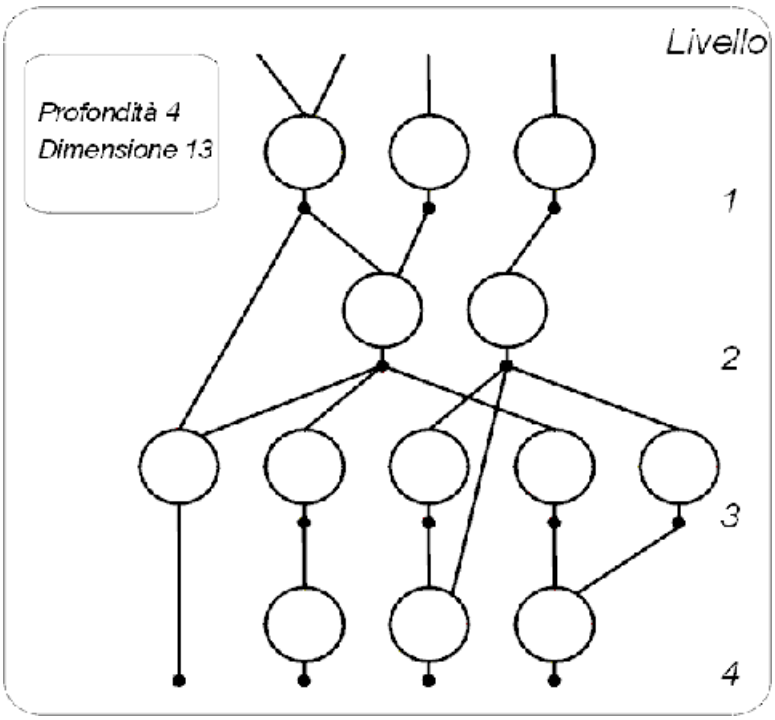


Figura 2: il circuito combinatorio

Tempo di simulazione

La simulazione partirà dagli elementi di livello 1 per passare a quelli di livello successivo solo quando tutti quelli del livello 1 saranno stati simulati e così via fino all'ultimo livello. Per ogni livello si cercherà di sfruttare al massimo il parallelismo assegnando (per quanto possibile) un processore ad ogni elemento.

Detto n[i] il numero di elementi di livello i, allora $\sum_{i=1}^{d} n_i = n$. Il numero dei passi paralleli ad ogni livello è $\left\lceil \frac{n_i}{p} \right\rceil$. In uno dei passi sono simulati $n_i - \left\lfloor \frac{n_i}{p} \right\rfloor p$ elementi, mentre negli (eventuali) altri esattamente p (**Figura 3)**. Il numero totale dei passi paralleli dà l'ordine di grandezza del tempo di simulazione del circuito:

$$T = \sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil \leq \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) = \mathrm{n/p} + \mathrm{d}$$

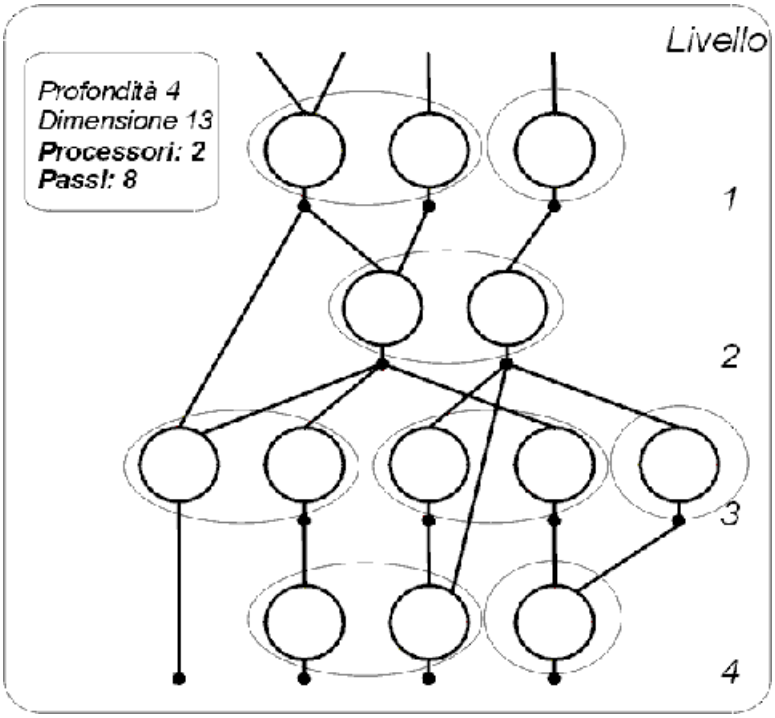


Figura 3: simulazione del circuito

Conclusioni

Osservazioni aggiuntive al teorema di Brent.

Se anche il fan-out fosse limitato basterebbe una PRAM EREW.

Infatti nella simulazione del circuito sarebbe sufficiente che il processore che simula un elemento combinatorio, calcolata la funzione, scriva (in tempo costante) il suo output negli elementi che lo richiedono. Questa politica della scrittura in tempo costante può essere applicata proprio perché il fan-out è limitato, infatti se il fan-out fosso illimitato, l'operazione di copia dell'output non potrebbe essere effettuata in O(1). Dunque procedendo nella simulazione, gli altri elementi si limiterebbero a leggere i propri input da **diverse** locazioni precedentemente fissate nella memoria.

SIMULAZIONE DI CIRCUITI A FAN-IN ILLIMITATO CON P-RAM CRCW

Se si fa a meno dell'ipotesi sulla limitatezza del fan-in, cioè si ammette che un elemento combinatorio possa avere un numero arbitrario di input, l'approccio utilizzato fino ad ora non può funzionare, perché un processore non può garantire la simulazione in tempo costante dell'elemento combinatorio a cui è associato.

Diventa inoltre necessario un modello a scrittura concorrente: infatti ad esempio il calcolo dell'AND logico di n variabili richiede un tempo di O(log(n)) su P-RAM CREW o EREW;

su una P-RAM CRCW comune con n processori è possibile invece calcolare l'AND di n variabili in tempo O(1), basta che ogni processore sia associato ad una delle variabili e scriva nella locazione risultato il valore di tale variabile: il risultato è 1 se e solo se tutte le variabili valgono 1.

Per approfondimenti sugli algoritmi di base nel calcolo parallelo su diversi modelli di macchine si veda [3]

Sia f una funzione booleana , $f : \{0,1\}^n \rightarrow \{0,1\}$; dato che f può essere espressa in forma normale congiuntiva, avendo a disposizione porte logiche a fan-in illimitato si può realizzare un circuito combinatorio che calcola l'output di f (che è uno solo) e che abbia profondità $d \leq 3$.

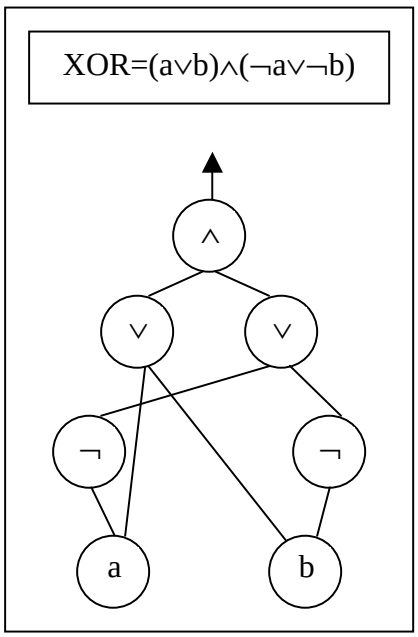


Figura 4: circuito combinatorio in forma normale per lo XOR

Per questo motivo i circuiti a fan-in illimitato vengono anche detti "circuiti a profondità limitata".

Sia C un circuito che realizza la forma normale di f, di profondità $d \leq 3$; C può essere simulato da una P-RAM CRCW in tempo O(1) con un numero di processori $\leq n2^n$.

Dim:

Poiché f ha n argomenti, la sua forma normale sarà l' AND di termini del tipo$(x_1 \vee \neg x_2 \vee \ldots \vee x_n)$;

Tali termini sono realizzati mediante porte OR a n ingressi, e sono al più 2^n.

Il primo livello del circuito è composto da al più da n porte NOT, ed è dunque simulabile in O(1) con n processori.

Una porta OR a n ingressi può essere simulata da una P-RAM CRCW comune con n processori allo stesso modo dell'AND , basta impostare il valore iniziale del risultato a 1, il suo valore sarà cambiato in 0 se e solo se tutte le variabili valgono 0.

Avendo a disposizione $n2^n$ processori è dunque possibile simulare in tempo O(1) ogni livello di C, che ha non più di tre livelli, quindi la simulazione ha tempo costante.

Estendiamo il ragionamento a una funzione booleana più generale $f : \{0,1\}^n \rightarrow \{0,1\}^m$:

sia C il circuito a n ingressi e m uscite che realizza f, sia s il numero di archi di C e d la sua profondità, C può essere simulato da una P-RAM CRCW comune con O(s) processori in tempo O(d).

Dim:

Sia M un vettore nella memoria condivisa della P-RAM.

Si numerino i nodi di C a partire dai nodi di input, e si associ al nodo i la locazione M[i], per ogni i.

Inizialmente gli n ingressi vengono memorizzati in M[i], per $1 \leq i \leq n$.

Associando un processore $P_{i,j}$ all'arco che va dal nodo i al nodo j in C, ogni livello del circuito può essere simulato in tempo O(1) sfruttando la scrittura concorrente nel modo descritto in precedenza. I processori $P_{i,j}$ al variare di j determinano il valore della locazione M[j].

Viene simulato C livello per livello attivando ad ogni passo i processori $P_{i,j}$, per ogni nodo j del livello corrente.

Dopo O(d) passi sequenziali si ottengono gli output del circuito.

Si noti che sono stati necessari O(s) processori, uno per ogni arco di C.

REALIZZABILITÀ DEI MODELLI

Tutti i risultati dimostrati fanno riferimento a modelli teorici di circuiti combinatori non sempre realizzabili in pratica per vari motivi.

In realtà un elemento combinatorio a fan-out arbitrariamente grande è impossibile da costruire:

si supponga di avere una linea l d'uscita di una porta logica E, che viene suddivisa in N linee per distribuire il segnale ad altrettanti elementi combinatori:

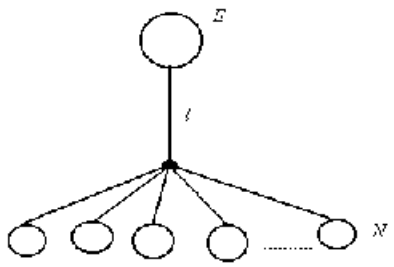


Figura 5: un elemento combinatorio con grande fan-out

Come conseguenza dei principi dell'elettromagnetismo (leggi di Ohm, seconda legge di Kirchoff), l'intensità di corrente di I viene suddivisa fra le N linee; al crescere di N l'intensità di corrente deve aumentare in proporzione per permettere agli elementi combinatori che seguono di ricevere il segnale e quindi l'output di E. Per valori di N grandi tale esigenza diventa proibitiva per la realizzazione del circuito.

Inoltre la costruzione di elementi combinatori a fan-in arbitrariamente grande comporta una crescita della complessità di realizzazione tale da renderli economicamente sconvenienti; si preferisce dunque aumentare la complessità dei circuiti utilizzando elementi molto semplici.

In pratica è difficile che vengano realizzate porte AND con più di otto entrate, anche se niente lo impedisce in teoria.

RISULTATI DEL TEOREMA DI BRENT

Reti di ordinamento

Per approfondimenti sulle reti di ordinamento si veda [1]

Una rete di ordinamento che fonde sequenze binarie qualunque può ordinare i suoi n input in tempo $T \in O(\log^2(n))$, dove il tempo sta ad indicare anche la profondità del circuito. Ogni livello della rete ha $O(n)$ elementi combinatori, dunque la dimensione del circuito è dell'ordine di $O(n \cdot \log^2(n))$. Poiché una rete di ordinamento è basata su confrontatori con fan-in e fan-out limitati segue dal teorema di Brent che questa può essere simulata da una PRAM EREW con **n** processori in tempo: $T \in O(n \cdot \log^2(n)/n + \log^2(n)) = O(\log^2(n))$.

Teorema

Se un algoritmo A viene eseguito su una PRAM con p processori in tempo T, allora, per ogni PRAM con p'< p processori, esiste un algoritmo A' che risolve lo stesso problema in tempo $T' \in O(T \cdot p/p')$.

Dim.:

La dimostrazione può essere ricavata direttamente dal risultato enunciato dal teorema di Brent.

L'esistenza di un algoritmo A eseguito su una PRAM con p processori, indica che esiste, secondo il teorema di Brent, un circuito combinatorio C di dimensione n e profondità d, che viene simulato dalla PRAM in tempo $T \in O(n/p+d)$. Data una seconda PRAM con p' processori, è allora possibile simulare il circuito C in tempo $T' \in O(n/p'+d)$. T' è il tempo dell'algoritmo A' impiegato per la simulazione di C.

E' possibile determinare il tempo T' in funzione del tempo T nel seguente modo:

$$T' \in O\left(\frac{n}{p'}+d\right) = O\left(\frac{p}{p}\frac{n}{p'}+d\right) = O\left(\frac{p}{p'}\frac{n}{p}+d\right) = O\left(\frac{p}{p'}\left(\frac{n}{p}+d\frac{p'}{p}\right)\right) =^{(*)} O\left(\frac{p}{p'}\left(\frac{n}{p}+d\right)\right) = O\left(\frac{p}{p'}T\right).$$

Per giustificare il passo dell' identità contrassegnato con $^{(*)}$, basta osservare che p'<p dunque p'/p<1. In (*) la quantità p'/p viene maggiorata con 1, lasciando immutato l'ordine di grandezza.

L'algoritmo A ha costo C = pT come anche A' che ha costo C' = p'T' = p'(p/p')T = pT. Quindi A' è efficiente tanto quanto lo è A (efficienza = $\frac{T_{p\,\text{A.S.M.}}}{C_{\,A.P.}}$).

Per approfondimenti sulle tecniche per misurare la complessità e l' efficienza di un algoritmo parallelo si veda [2].

Un algoritmo

Il precedente risultato verrà provato su un algoritmo. L'algoritmo trova il valore massimo di un vettore di N elementi usando una PRAM EREW con N processori.

Per approfondimenti sugli algoritmi di base nel calcolo parallelo su diversi modelli di macchine si veda [3]

Si supponga che N sia un quadrato potenza di 2 (es: 16), che i processori siano numerati da 1 a N così come gli elementi che sono contenuti nel vettore V. Il vettore si assume già riempito dei valori su cui si vuole operare. R è la locazione che conterrà il risultato finale ovvero max{a; a∈V}.

Algoritmo A

```
1.for i = 1 to logN do
2.  for j = 1 to N/2^i pardo
3.    P_j: V[j] := max( V[j], V[j+N/2^i] )
4.P_1: R := V[1]
```

L'algoritmo ha tempo T∈O(log N) e costo C∈O(N log N).

Algoritmo A'

Se ho una PRAM EREW con p'= $\sqrt{N}$ processori, allora deve esistere un algoritmo che trova il massimo su V in tempo T'∈O(T N/p') = O((log N) N/$\sqrt{N}$) = O($\sqrt{N}$ log N) e costo C' = O($\sqrt{N}$ $\sqrt{N}$ log N) = O(N log N) = C.

L'idea alla base dell'algoritmo e analoga a quella precedente. La sola differenza è che ora si hanno a disposizione meno processori e ogni passo parallelo di A verrà suddiviso in più parti.

```
1.for i = 1 to logN do
2.  for k = 1 to ⌈N^(1/2)/2^i⌉
3.    first = (k-1)*N^(1/2)+1
4.    last  = min( N/2^i, k*N^(1/2) )
5.    for j = first to last pardo
6.      P_(j-first+1): V[j] := max( V[j], V[j+N/2^i] )
7.P_1: R := V[1]
```

Il passo 1 è analogo a quello dell'algoritmo A.

Nella linea 2 non vengono fatti partire tutti i processori, ma si determinano i blocchi di $\sqrt{N}$ processori da far partire in parallelo, che sono $\lceil N^{1/2}/2^i \rceil$.

Il passo 3 indica quale è il primo indice di A su cui il blocco di processori dovrà lavorare.

Il passo 4 indica quale è l'ultimo indice di A su cui il blocco di processori dovrà lavorare. Non è vero che last = $k*N^{1/2}$, infatti nell'istruzione 2 l'ultimo valore di j è una parte intera superiore. Potrebbe accadere che $k* N^{1/2} > N/2^i$. In questo caso partirebbero più processori di quelli realmente necessari, mentre, come si nota nell'istruzione 2 dell'algoritmo A, si vuole lavorare sola sugli elementi da 1 a $N/2^i$. Questo giustifica la scelta del valore minore tra i due.

Il passo 5 fa partire tutti i processori disponibili in parallelo. Ovviamente nella linea 6 gli indici dei processori sono cambiati. Il processore che andrà a lavorare sulla generica posizione j del vettore avrà indice j-first+1.

L'algoritmo A' ha tempo $T' \in O\left(\sum_{i=1}^{\log N}\sum_{j=1}^{\sqrt{N}/2^i} 1\right) = O\left(\sum_{i=1}^{\log N}\frac{\sqrt{N}}{2^i}\right) = O\left(\sqrt{N}\sum_{i=1}^{\log N}\frac{1}{2^i}\right) = O\left(\sqrt{N}\sum_{i=1}^{\log N}\frac{1}{2}\right) =$

$O(\sqrt{N} \log N)$.

Riferimenti ed approfondimenti

In questa sezione vi è una breve lista dei riferimenti ad altre tesine, scritte dagli studenti del medesimo corso di Algoritmi, che sono indispensabili per la piena comprensione degli argomenti trattati, nonché una lista dei testi che sono stati consultati per la realizzazione della presente tesina e che sono utili per ulteriori approfondimenti.

[1] Tesina: “Algoritmi di Ordinamento” di D’Avello, Di Lullo, Le pera, Longo, Pascale, Righetti .

Problematiche di ordinamento: algoritmi di ordinamento e reti di ordinamento parallelo.

[2] Tesina: “Misure di Complessita e P-Completezza” di Cerini, De Santis.

Tecniche per misurare la complessità degli algoritmi paralleli: tempo, costo, efficienza, … .

[3] Tesina: “Algoritmi Paralleli di Base” di Concetta, Pilotto .

Algoritmi di base (somma, massimo, …) del calcolo parallelo, eseguiti su vari modelli di macchine: dalle PRAM ai multiprocessori immersi nella rete (vettoriali, mesh, …).

[4] Tesina: “Algoritmi Paralleli di Base” di Concetta, Pilotto.

Tecnica delle somme prefisse.

[5] “Introduzione agli algoritmi” volume 3 di T.H. Cormen, C.E. Leiserson, R.L. Rivest

Capitolo 29.1 – Circuiti combinatori

Capitolo 30.2 – Confronto tra algoritmi EREW e algoritmi CRCW

Capitolo 30.3 – Il teorema di Brent e l’efficienza rispetto al lavoro

Da pag 3 a 31 abbiamo seguito i seguenti testi di riferimento

“An introduction to parallel algorithms” di J. Jaja [6]

“The design and analysis of parallel algorithms” di S.G. Akl