Array Processor

parte seconda

di Giuseppe Cardinale Ciccotti

Nel precedente articolo sugli Array Processor, abbiamo introdotto i concetti fondamentali che caratterizzano i sistemi multiprocessore detti Array Processor. In questa categoria vengono raccolti sistemi con struttura spesso assai differente tra loro, sia dal punto di vista topologico che dal punto di vista del meccanismo di funzionamento. Tuttavia il fatto che tutti i Processor Element (PE) di un Array Processor, eseguano la stessa operazione negli stessi istanti di tempo, è una caratteristica determinante per la loro classificazione in un'unica categoria. Questo tipo di gestione delle fasi di calcolo, si riflette naturalmente sulla programmazione di tali sistemi

La programmazione

Allo stato delle attuali conoscenze, il programmatore deve esplicitamente gestire tutte le trasmissioni di dati fra i PE dell'Array, prevedere modi di sincronizzazione ed eventuali stati fittizi per tutti i PE non coinvolti nel passo dell'algoritmo. Appare poi naturale, che ogni struttura restituisca prestazioni migliori di altre per quegli algoritmi per cui è stata esplicitamente progettata: un Array Processor i cui elementi siano collegati ad albero binario avrà delle prestazioni «ottime» per gli algoritmi di tipo «divide et impera», dove l'algoritmo prevede operazioni ricorsive su sottoinsiemi binari dell'insieme di ingresso; una struttura a reticolo sarà invece molto più indicata per algoritmi dove il calcolo di una variabile dipende per esempio dalle m variabili adiacenti nell'insieme, come avviene per esempio negli algoritmi di Image Processing. A meno che il sistema sia esclusivamente dedicato ad una specifica funzione e perciò il suo programma sia fissato in fase di progetto, anzi in tal caso l'algoritmo determina le scelte progettuali, il programmatore di un Array Processor ha il compito di dover implementare un algoritmo efficiente per la struttura che eseguirà il programma. Come abbiamo già avuto modo di osservare, l'attenzione maggiore deve essere posta nella minimizzazione delle comunicazioni interPE, ciò si ottiene progettando accuratamente l'allocazione iniziale dei dati le eventuali condivisioni di questi e le inevitabili dipendenze fra di essi. L'altro punto su cui focalizzare l'attenzione è quello della successione temporale dell'esecuzione delle varie istruzioni, spesso infatti un'operazione complessa è spezzata in semplici operazioni eseguite in tempi diversi su PE diversi: un'errata sincronizzazione di tali operazioni può portare a risultati errati oltreché ad inefficienze; tra l'altro tali situazioni sono assai difficili da scoprire in fase di debugging. È ovvio infine che tali architetture SIMD «Single Instruction Multiple Data», offrono le loro migliori performance negli algoritmi in cui si debbano fare multiple operazioni su un insieme omogeneo di dati.

```
For i= 1 to n do
 For j= 1 to n do
  cij=0                          (inizializzazione)
  For k=1 to n do
   cij=cij+aik · bkj    (moltiplicazione scalare e accumulo)
  End k
 End j
End i
```

Esempio A

```
For i= 1 to n do
 Par for k= 1 to n do
  c=0                            (inizializzazione)
 For j= 1 to n do
  Par for k=1 to n do
   cik=cik+aij · bjk    (moltiplicazione vettoriale e accumulo)
 End j
End i
```

Esempio B

Moltiplicazioni di matrici

Vediamo quindi un esempio per chiarire i concetti che abbiamo espresso; consideriamo il problema della moltiplicazione di matrici bidimensionali. Le matrici non sono altro che tabelle ordinate e possono essere memorizzate in un vettore bidimensionale. Ogni elemento sarà perciò individuato da un indice di riga ed uno di colonna. Chiamiamo le due matrici A e B, vogliamo calcolare la matrice C=A*B. Il prodotto tra matrici viene eseguito come si dice «righe per colonne», vale a dire che si moltiplica ciascuna riga della matrice a sinistra per ciascuna colonna della matrice a destra. Le matrice A perciò dovrà avere numero di colonne pari al numero di righe della matrice B. Le dimensioni delle matrici dovranno essere perciò A(m×n) B(n×p), la matrice prodotto C avrà dimensione C(m×p). Nel nostro esempio considereremo, per semplicità, matrici «quadrate» cioè con numero di righe uguale al numero di colonne, la matrice C avrà perciò lo stesso numero di righe e di colonne delle matrici operando A e B. In figura 1 potete trovare esplicitati i sedici termini della matrice C(4×4)=A(4×4)*B(4×4). Dal punto di

vista operativo, dobbiamo soltanto eseguire 64 moltiplicazioni e 16 somme. Vediamo perciò come è organizzato l'algoritmo seriale e valutiamone la complessità che ci servirà come paragone per lo speed-up dell'algoritmo parallelo. Una codifica ad alto livello di un frammento di programma per la moltiplicazione di matrici n×n (gli elementi delle matrici sono individuati dalla rispettiva minuscola con doppio indice, riga, colonna) può essere rappresentata come nell'esempio A.

Questo algoritmo ha una complessità che può essere semplicemente valutata considerando che la moltiplicazione e accumulo è interna ai 3 cicli: è perciò eseguita n^3 volte, la complessità asintotica sarà perciò $O(n^3)$.

Ora vogliamo eseguire la stessa operazione su un Array Processor di n PE. Come abbiamo detto le performance dell'algoritmo dipendono «pesantemente» dalla maniera in cui gli elementi delle matrici sono allocati in memoria. Scegliamo di organizzare i dati come mostrato in figura 2; nella memoria di ciascun PE, sono memorizzate le stesse colonne delle tre matrici. Quest'allocazione permette un accesso parallelo alle righe delle matrici. Otteniamo di conseguenza l'algoritmo parallelo come è pubblicato nell'esempio B.

I due costrutti Par do corrispondono alle operazioni parallele indicate dal corpo del costrutto stesso; in questo caso indicano che le n istruzioni di inizializzazione e di moltiplicazione sono eseguite contemporaneamente sugli n PE e devono essere considerate una singola operazione dal punto di vista funzionale. Bisogna considerare che invece la moltiplicazione vettoriale implica anche che la UC prelevi a_{ij} dalla memoria del PE_j e lo trasmetta a tutti i PE dell'Array Processor, in tal modo ognuno degli n PE, può eseguire contemporaneamente una moltiplicazione scalare sull'elemento b_{ik} della matrice B. In totale verranno perciò eseguite n*n moltiplicazioni vettoriali con una complessità asintotica pari a $O(n^2)$. In figura 3 trovate lo scheduling completo di tale algoritmo, le operazioni su ciascuna riga sono eseguite contemporaneamente e prendono il tempo di una sola operazione, tempo di trasmissione a parte. Lo speed-up di quest'algoritmo è pari a

$$\text{speed-up} = \frac{n^3\ tm}{n^2\ (tm+tt)}$$

tm = tempo di moltiplicazione
tt = tempo di trasmissione

che risulta uguale a n se tt è trascurabile rispetto a tm. Dall'algoritmo proposto si può evincere anche quale è la struttura ottimale per rispettare tale vincolo e ottenere perciò la massima performance. L'Array Processor di figura 4 consente con una sola istruzione della UC di trasmettere a_{ij} a tutti i PE contemporaneamente e quindi minimizza tt. La struttura a reticolo di figura 5 in cui ogni PE è connesso soltanto ai suoi 4 adiacenti, è invece meno efficiente poiché il tt varia con la posizione nel reticolo del PE da cui il dato è trasmesso e in ogni caso sono necessarie non meno di (n−1) trasmissioni e al più (2*n−2) trasmissioni per ciascuna moltiplicazione. Da questa considerazione appare evi-

Matrice A

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$$

Matrice B

$$\begin{matrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{matrix}$$

$$c_{11} = a_{11}*b_{11} + a_{12}*b_{21} + a_{13}*b_{31} + a_{14}*b_{41} \qquad c_{31} = a_{31}*b_{11} + a_{32}*b_{21} + a_{33}*b_{31} + a_{34}*b_{41}$$
$$c_{12} = a_{11}*b_{12} + a_{12}*b_{22} + a_{13}*b_{32} + a_{14}*b_{42} \qquad c_{32} = a_{31}*b_{12} + a_{32}*b_{22} + a_{33}*b_{32} + a_{34}*b_{42}$$
$$c_{13} = a_{11}*b_{13} + a_{12}*b_{23} + a_{13}*b_{33} + a_{14}*b_{43} \qquad c_{33} = a_{31}*b_{13} + a_{32}*b_{23} + a_{33}*b_{33} + a_{34}*b_{43}$$
$$c_{14} = a_{11}*b_{14} + a_{12}*b_{24} + a_{13}*b_{34} + a_{14}*b_{44} \qquad c_{34} = a_{31}*b_{14} + a_{32}*b_{24} + a_{33}*b_{34} + a_{34}*b_{44}$$
$$c_{21} = a_{21}*b_{11} + a_{22}*b_{21} + a_{23}*b_{31} + a_{24}*b_{41} \qquad c_{41} = a_{41}*b_{11} + a_{42}*b_{21} + a_{43}*b_{31} + a_{44}*b_{41}$$
$$c_{22} = a_{21}*b_{12} + a_{22}*b_{22} + a_{23}*b_{32} + a_{24}*b_{42} \qquad c_{42} = a_{41}*b_{12} + a_{42}*b_{22} + a_{43}*b_{32} + a_{44}*b_{42}$$
$$c_{23} = a_{21}*b_{13} + a_{22}*b_{23} + a_{23}*b_{33} + a_{24}*b_{43} \qquad c_{43} = a_{41}*b_{13} + a_{42}*b_{23} + a_{43}*b_{33} + a_{44}*b_{43}$$
$$c_{24} = a_{21}*b_{14} + a_{22}*b_{24} + a_{23}*b_{34} + a_{24}*b_{44} \qquad c_{44} = a_{41}*b_{14} + a_{42}*b_{24} + a_{43}*b_{34} + a_{44}*b_{44}$$

Figura 1 - Moltiplicazione di matrici. Gli elementi di A, B e C sono individuati da una coppia di indici, rispettivamente di riga e colonna. Gli elementi della matrice C risultato sono calcolati moltiplicando ogni riga di A per ciascuna delle colonne di B e sommando i prodotti ottenuti.

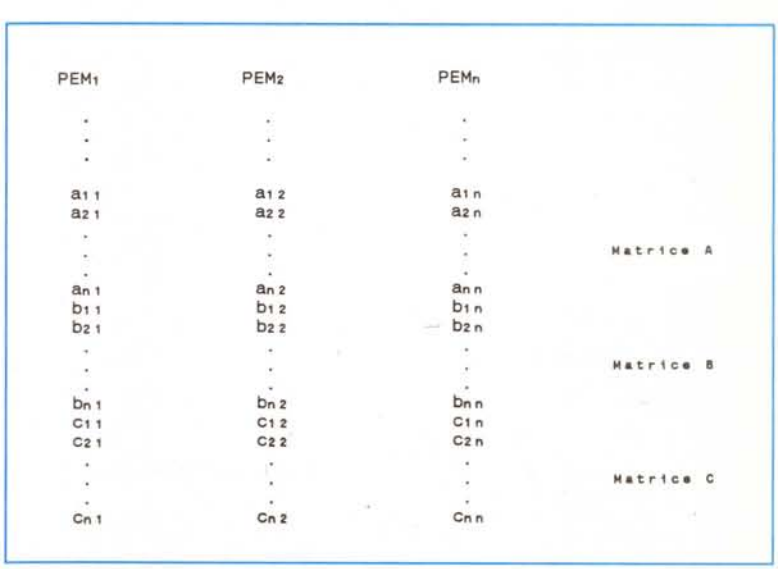

Figura 2 - Allocazione dei dati nei moduli di memoria dell'Array Processor per l'algoritmo di moltiplicazione di matrici. Le locazioni relative agli elementi della matrice C sono inizializzate a 0.

dente che il progetto di un algoritmo è strettamente legato all'Array Processor da utilizzare e il programmatore deve cercare di volta in volta l'algoritmo che meglio si adatta al proprio sistema. Tuttavia è sempre possibile adottando un sistema a topologia riconfigurabile, nel quale la rete interPE possa connettere i PE secondo le proprie necessità, scavalcare il problema della ricerca dell'algoritmo, «riadattando» il sistema secondo le proprie esigenze. Addirittura si può pensare di adottare strutture sufficientemente grandi in modo che in essa possano essere contenute strutture di ordine inferiore: per esempio in un reticolo di 3×3 PE, in figura 6, è contenuto un albero binario di 3 livelli composto di 7 PE; con questo approccio però si «sprecano» alcuni PE e l'efficienza, la percentuale di utilizzo di ciascun PE durante l'esecuzione dell'algoritmo, è in generale bassa. Tuttavia, anche si iniziano ad intravedere nuovi approcci, questo è stato il metodo di progetto più diffuso fino ad oggi per la realizzazione di sistemi multiprocessori SIMD commerciali.

L'allocazione dei dati

I lettori più esperti di programmazione sicuramente riconosceranno l'importanza che riveste, in fase di progetto di un programma, la scelta della struttura con cui allocare i dati significativi dell'algoritmo. Molto spesso da una struttura di dati adatta al problema dipende l'efficienza del programma stesso; non a caso quindi lo studio delle strutture di dati costituisce uno dei pilastri dell'informatica. Nell'ambito del parallel processing la strategia di allocazione dei dati si rivela di importanza fondamentale ai fini dell'efficienza e della correttezza del programma. Una errata allocazione può infatti causare errori non previsti, per esempio un certo PE_1 richiede in un passo del programma un dato indispensabile per produrre un certo risultato, ma tale dato non è disponibile perché il PE_2 che lo deve fornire ha bisogno del risultato del PE_1. Si cade perciò in una situazione detta di blocco critico. In generale tale situazione porta alla paralisi del sistema o parte di esso, perché PE_1 e PE_2 «si aspettano» a vicenda; tuttavia, nel caso degli Array Processor, in cui tutti i PE sono sincroni tra loro, il blocco critico non comporta uno stallo, ma soltanto un risultato errato. Inoltre il debugging di un programma che origina tali errori può essere molto complesso in quanto l'indeterminatezza della situazione fa sì che il risultato della computazione possa variare di volta in volta. Ulteriore complicazione deriva dal fatto che l'algoritmo, su cui è stato costruito il programma, può essere corretto se non esplicita l'allocazione dei dati stessi. Non esiste fino ad oggi una metodologia per valutare la correttezza dei programmi paralleli. Vediamo ora un esempio di un programma che esegue la moltiplicazione di due matrici A e B, introdotta nel paragrafo precedente, verificando come una diversa allocazione dei dati permette, di ottenere prestazioni migliori su una struttura che con

Operazioni parallele per $k=1,2,\ldots n$

Ciclo	Ciclo			
1	j	$C_{11}=C_{11}+a_{1j}*b_{j1}$	$C_{12}=C_{12}+a_{1j}*b_{j2}$	$C_{1n}=C_{1n}+a_{1j}*b_{jn}$
1	1	$C_{11}=C_{11}+a_{11}*b_{11}$	$C_{12}=C_{12}+a_{11}*b_{12}$	$C_{1n}=C_{1n}+a_{11}*b_{1n}$
	2	$C_{11}=C_{11}+a_{12}*b_{21}$	$C_{12}=C_{12}+a_{12}*b_{22}$	$C_{1n}=C_{1n}+a_{12}*b_{2n}$
	.	.	.	.
	.	.	.	.
	.	.	.	.
	n	$C_{11}=C_{11}+a_{1n}*b_{n1}$	$C_{12}=C_{12}+a_{1n}*b_{n2}$	$C_{1n}=C+a_{1n}*b_{nn}$
2	1	$C_{21}=C_{21}+a_{21}*b_{11}$	$C_{22}=C_{22}+a_{21}*b_{12}$	$C_{2n}=C_{2n}+a_{21}*b_{1n}$
	2	$C_{21}=C_{21}+a_{22}*b_{21}$	$C_{22}=C_{22}+a_{22}*b_{22}$	$C_{2n}=C_{2n}+a_{22}*b_{2n}$
	.	.	.	.
	.	.	.	.
	.	.	.	.
	n	$C_{21}=C_{21}+a_{2n}*b_{n1}$	$C_{22}=C_{22}+a_{2n}*b_{n2}$	$C_{2n}=C_{2n}+a_{2n}*b_{nn}$
	.	.	.	.
	.	.	.	.
	.	.	.	.
n	1	$C_{n1}=C_{n1}+a_{n1}*b_{11}$	$C_{n2}=C_{n2}+a_{n1}*b_{12}$	$C_{nn}=C_{nn}+a_{n1}*b_{1n}$
	2	$C_{n1}=C_{n1}+a_{n2}*b_{21}$	$C_{n2}=C_{n2}+a_{n2}*b_{22}$	$C_{nn}=C_{nn}+a_{n2}*b_{2n}$
	.	.	.	.
	.	.	.	.
	.	.	.	.
	n	$C_{n1}=C_{n1}+a_{nn}*b_{n1}$	$C_{n2}=C_{n2}+a_{nn}*b_{n2}$	$C_{nn}=C_{nn}+a_{nn}*b_{nn}$
	Processore	PE_1	PE2	PEn

Figura 3 - Scheduling dell'algoritmo di moltiplicazione di matrici (n×n). Le operazioni su ciascuna riga sono eseguite negli stessi istanti.

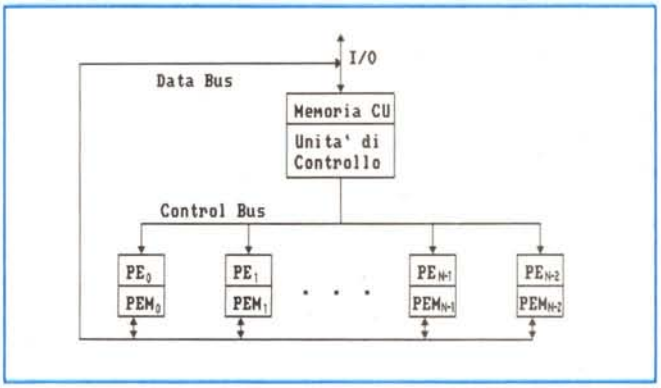

Figura 4 - Array Processor che permette la comunicazione di a_{ij} in un solo ciclo di trasmissione a tutti i PE della struttura.

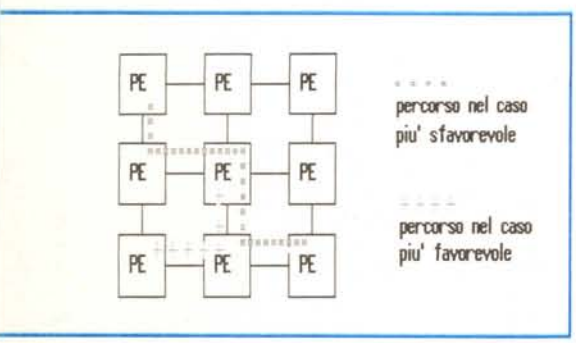

Figura 5 - Nella struttura reticolare la trasmissione da un processore a tutti gli altri comporta da un minimo di n−1 ad un massimo di 2n−2.

Bibliografia

Hwang K., Briggs F. ***Computer Architecture and Parallel Processing***, MC Graw-Hill, 1988.

Kung S.Y., Lo S.C., Jean S.N., Hwang J.N. ***Wavefront Array Processors. Concept to Implementation***, IEEE Computer, luglio 1987, pp. 18-33.

Izzi M. ***Software RIA***, Tesi di Laurea, Facoltà di Ingegneria Università «La Sapienza» Roma, marzo 1989.

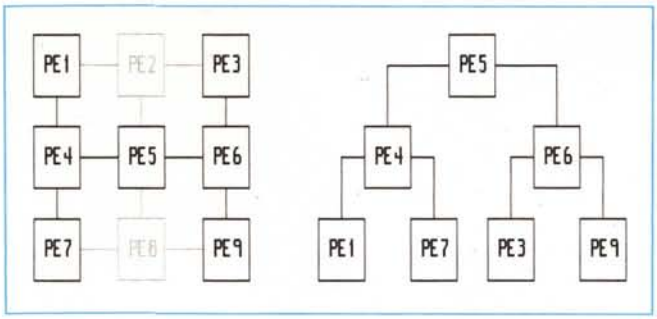

Figura 6 - Una struttura ad albero binario è contenuta in un reticolo. Vengono però sprecate risorse e l'efficienza è bassa.

l'allocazione di dati disposta in precedenza, figura 2, aveva una scarsa efficienza.

Wavefront array

Disponiamo un reticolo di n×n PE dove n è la dimensione delle matrici quadrate A e B; poniamo n=4 ottenendo l'Array Processor in figura 7. In ogni passo del programma, ciascun PE riceve a_{ij} e b_{ij} dai PE a sinistra e sopra. Il calcolo procede quindi da sinistra a destra e dall'alto in basso. I PE sul bordo sinistro e su quello più in alto dell'Array vengono interessati dall'input esterno. I risultati, le componenti della matrice C, sono contenuti in ciascun PE in modo che il PE_{ij} fornisca C_{ij}. L'Array si comporta come una pipeline bidimensionale, come si vede dalla figura considerando l'ordine di arrivo degli operandi e tenendo conto che ogni PE esegue una moltiplicazione ed accumulo solo quando son presenti a_{ij} e b_{ij}, altrimenti conserva l'operando. Ad ogni passo del programma ogni PE trasmette al PE adiacente in basso il b_{ij} ricevuto dal PE in alto nel passo precedente e al PE a destra l'operando ricevuto da sinistra. I risultati perciò sono disponibili secondo il fronte d'onda tratteggiato in figura, da questo tipo di scheduling proviene il nome Wavefront Array cioè Array a fronte d'onda. Le prestazioni di questo algoritmo si ricavano con la seguente considerazione: dopo aver immesso l'ultima serie di dati sui PE di input (vale a dire l'ultima colonna di A e l'ultima riga di B), PE_{11} fornisce C_{11}. Nel successivo passo PE_{12}, PE_{21}, PE_{22} forniscono C_{12}, C_{21} e C_{22} rispettivamente, nel secondo su PE_{13}, PE_{31}, PE_{23}, PE_{32}, PE_{33} sono disponibili i corrispondenti risultati e nel terzo passo si potranno prelevare C_{14}, C_{41}, C_{24}, C_{42}, C_{34}, C_{43}, C_{44} dai PE omologhi. In totale quindi otterremo la matrice C dopo 4+3 passi, con uno speed-up di:

$$\text{speed-up } 4\times4 = \frac{43\ tm}{7\ (tt+tm)} = \frac{64}{7} = 9.143$$

In generale avremo:

$$\text{speed-up} = \frac{n^3\ tm}{(n+n-1)\ (tt+tm)} = \frac{n^3\ tm}{(2n-1)\ (tt+tm)}$$

se tt è piccolo rispetto a tm e $n \gg 1$ allora lo speed-up è pari a $n^2/2$.

Confrontato allo speed-up del primo esempio relativo ad un Array ad n processori, si è ottenuto un incremento di un fattore n al prezzo però di dover disporre di n^2 PE. I lettori che hanno seguito i precedenti articoli, ricorderanno che la struttura pipeline è caratterizzata da un tempo di latenza che risulta ininfluente nel computo totale dello speed-up se una volta instaurata la pipeline, veniva immesso un numero di dati molto superiore al numero di stadi della pipeline stessa. Nel caso del Wavefront Array si può fare la stessa considerazione, perciò il tempo di latenza pari a (n−1)*(tm+tt) può essere trascurato se sfruttiamo l'Array proprio come una pipeline immettendo di seguito le righe e le colonne di m matrici $B_1...B_m$ e $A_1...A_m$ di cui vogliamo le matrici prodotto $C_1...C_m$. In tal caso avremo un solo tempo di latenza e la performance complessiva sarà m*n+n−1 passi e lo speed-up risulterà:

$$\text{speed-up} = \frac{m*n^3\ tm}{(m*n+n-1)\ (tm+tt)} = \frac{m*n^3\ tm}{(n(m+1)-1)\ (tm+tt)}$$

pari a n^2 se $tt \ll tm$ e m è grande rispetto a 1.

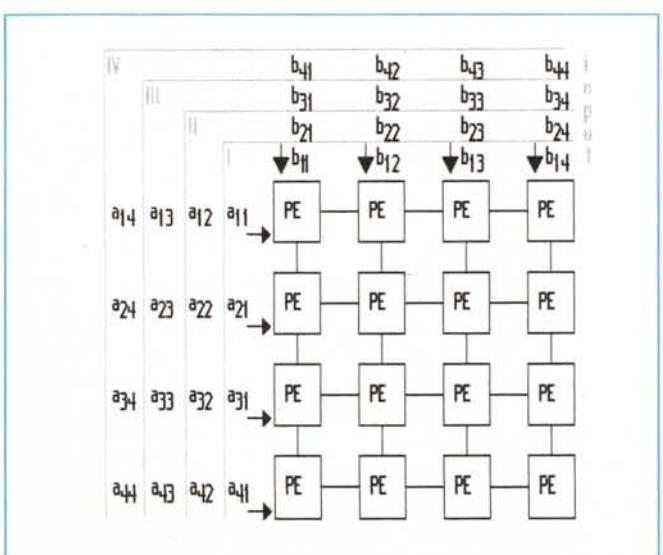

Figura 7 - Successione dei dati di input per il programma di moltiplicazione di matrici in un Wavefront Array. Gli input sono sui PE di bordo, gli output su tutti i PE.

Conclusione

L'esempio della moltiplicazione di matrici ha messo in luce le diverse maniere in cui può essere implementato uno stesso problema. Abbiamo verificato l'intuitiva relazione che lega la diminuzione del tempo di calcolo con l'aumento delle risorse a disposizione. Tuttavia ci siamo resi conto dell'importanza di una corretta allocazione dei dati rispetto alla struttura hardware. Si fa presente che le metodologie di programmazione degli Array Processor e delle macchine parallele in genere sono oggi insufficienti a garantire uno sviluppo del software diretto e pianificato. Il programmatore si deve occupare di gestire tutti i processi e le loro interazioni a basso livello; tale compito si rivela di difficoltà crescente con la complessità dell'algoritmo. Di conseguenza i problemi che meglio sono stati risolti, sono quelli che comportano un «grande» numero di semplici operazioni ripetitive. Questa è la ragione dello sviluppo delle macchine di tipo SIMD, anche di dimensioni elevate. Tuttavia man mano che vengono prodotte nuove conoscenze ed esperienze, si preferiscono architetture più potenti e flessibili come quelle che illustreremo nei prossimi appuntamenti.