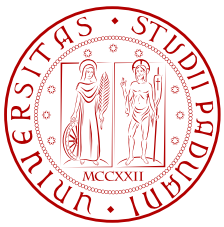
UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Strutture bidimensionali di memoria con pipeline ad alta banda



Relatore:
Prof. Gianfranco Bilardi

Candidato:
Giorgio Vezzaro

25 Ottobre 2011

Ringraziamenti

A conclusione di un percorso così lungo e significativo, trovo doveroso ricordare con gratitudine le persone che mi hanno accompagnato.
Ringrazio il Professor Gianfranco Bilardi, relatore di questa tesi, per i preziosi insegnamenti, non solo didattici, che hanno favorito la mia crescita intellettuale.
Ringrazio l'Ing. Emanuele Milani che, nelle fasi decisive per la conclusione del lavoro, mi ha fornito un aiuto concreto e preciso.
Non ci sono parole a sufficienza per ringraziare i miei genitori Grazia e Antonio, che mi hanno permesso di arrivare fin qui, realizzando il desiderio che avevo fin da piccolo. A loro sono grato di essere quello che sono e, dopo tanti anni di sacrifici per me, voglio che sentano loro questa laurea.
Il pensiero si estende a tutta la mia famiglia, che mi ha sempre sostenuto e appoggiato. Con ciascuno di loro voglio condividere questo momento importante.
Un pensiero speciale ad Helena, che standomi al fianco mi ha incoraggiato e dato fiducia, specie nelle fasi più complicate. A lei esprimo il mio più sentito ringraziamento per la vicinanza e l'aiuto. Desidero che questo mio traguardo sia seguito presto dal suo.
Ringrazio le tante persone che hanno fatto il tifo per me lungo la salita.
Un pensiero è infine per i tanti compagni di battaglia, con i quali ho affrontato e superato in questi anni tante difficoltà.

« Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza »

Se insisti e resisti raggiungi e conquisti

Sommario

La caratteristica della *Random Access Machine* (RAM) di eseguire le istruzioni in tempo costante rimane confinata in un piano puramente ideale: a causa di limiti fisici fondamentali, quali la taglia minima di un dispositivo e la velocità massima di un segnale, essa non può essere considerata realizzabile.
Questo lavoro si innesta nella ricerca di come la RAM ideale possa essere approssimata con macchine fisiche. Esso nasce come sviluppo al progetto di G. Bilardi, K. Ekanadham e P. Pattnaik, del quale approfondisce in maniera specifica la memoria dell'architettura di calcolo. L'idea è quella di rimuovere dal progetto in questione il vincolo di una sola porta per le comunicazioni tra memoria e processore, aumentando così la banda di I/O. A partire da questa nuova condizione, si esplora la possibilità di progettare una memoria nel piano che consideri questo potenziamento e tragga vantaggio dal parallelismo, senza pregiudicare la logica sequenziale delle istruzioni.
La trattazione illustra i vari passi del percorso di ricerca, per giungere infine ad un'organizzazione *pipelined* bidimensionale con M nodi, ciascuno contenente μ celle di memoria. Tale struttura è in grado di gestire fino a $\theta(\frac{\sqrt{M}}{\sqrt{\mu}})$ richieste per unità di tempo, con latenza $\theta(\sqrt{\mu}\sqrt{M})$. Considerando le due metriche di banda e latenza, i valori raggiunti sono ottimi entro un fattore costante.

Indice

Elenco delle figure

Capitolo 1

Introduzione

La *Random Access Machine* (RAM) è un modello computazionale astratto che, negli ultimi decenni, ha fornito le basi per la progettazione e l'analisi di svariati algoritmi sequenziali.
La RAM può essere vista anche come una versione alquanto idealizzata della *architettura di Von Neumann*, che per più di cinquant'anni ha costituito il riferimento per la progettazione di molti calcolatori sperimentali e commerciali.
Una caratteristica centrale della RAM è rappresentata dalla possibilità di accedere ad una qualsiasi locazione di memoria in un solo passo base della macchina, indipendentemente dal numero di locazioni di memoria disponibili. È inoltre consuetudine definire il tempo di esecuzione di una computazione RAM come il numero di passi base che si susseguono in questa computazione.
Questo è possibile assumendo che un passo base della RAM possa essere eseguito in una quantità di tempo indipendente dalla taglia della memoria, condizione che appare impossibile da verificare in qualsiasi macchina fisica. Più precisamente, l'apparente impossibilità deriva dalla congiunzione di due principi:

- Principio di massima densità di informazione, che afferma la necessità di un volume spaziale minimo per mantenere un bit di informazione
- Principio della velocità massima dell'informazione, che afferma l'esistenza di un limite superiore alla velocità con cui un bit di informazione può viaggiare

Il primo principio implica che memorizzare M bit, in uno spazio d-dimensionale, richiede una regione la cui dimensione lineare cresce proporzionalmente a $M^{1/d}$ che, attraverso il secondo principio, si traduce in una crescita similare del tempo di accesso.
In definitiva, un passo base di una RAM con M bit di memoria impiega, nel caso peggiore, un tempo fisico $\Omega(M^{1/d})$. È d'altronde piuttosto immediato pensare ad un'organizzazione di memoria il cui tempo di accesso sia limitato superiormente da

$O(M^{1/d})$, consentendo ad un passo della RAM di essere eseguito entro il medesimo limite temporale. Quindi, in un'opportuna macchina fisica d-dimensionale, un'arbitraria computazione RAM di N passi può essere eseguita in tempo $T = O(NM^{1/d})$. L'ampia questione di fondo su cui indaga questo lavoro è se tale limite temporale può essere migliorato, attraverso organizzazioni architetturali più sofisticate, per classi significative di computazioni RAM.
Mentre la nostra attenzione è focalizzata principalmente su come i vincoli fisici fondamentali limitino le prestazioni di macchine asintoticamente grandi, il principale interesse nella progettazione delle attuali macchine è rappresentato dal collo di bottiglia alle prestazioni che deriva dagli accessi di memoria lenti rispetto alla velocità del processore. La differenza ha subito un incremento costante a partire dal 1980, per diventare particolarmente acuta nell'ultimo decennio, durante il quale è conosciuta come *memory wall, gap* o *barrier*. Il problema tuttavia non era sfuggito all'attenzione di von Neumann, fin dagli albori del calcolo automatico.
Ci sono fondamentalmente due modi per attenuare l'impatto della latenza di memoria sul tempo complessivo di esecuzione.
La prima via consiste nell'organizzare la memoria come una *gerarchia* di livelli, dal più piccolo e veloce al più capiente e lento, e nello strutturare la computazione in modo tale che gli accessi ai livelli veloci siano più frequenti rispetto ai livelli più lenti. È comune affermare che le computazioni così strutturate esibiscono *località temporale*.
La seconda modalità prevede invece di organizzare la memoria in modo da abilitare la *concorrenza* negli accessi e nello strutturare la computazione in maniera da trarre vantaggio da questa concorrenza.
Le due strategie possono anche interagire, come avviene nella maggior parte dei sistemi computazionali, in cui la memoria è partizionata in insiemi di parole con indirizzi consecutivi, chiamati *blocchi* o *linee*, e i trasferimenti tra livelli della gerarchia avvengono in blocchi, in maniera concorrente per tutte le parole del blocco. Questa caratteristica si traduce in un vantaggio delle prestazioni quando la computazione tende ad accedere ad indirizzi consecutivi in passi vicini, quando cioè la computazione esibisce la proprietà nota come *località spaziale*.
Questo lavoro si inserisce dunque nel contesto di approssimazione del modello RAM con macchine fisicamente realizzabili. Il progetto di riferimento è costituito dalla macchina *general-purpose* proposta da Bilardi, Ekanadham e Pattnaik nell'articolo *On Approximating the Ideal Random Access Machine by Physical Machines* [2]. A partire da questa macchina nasce un'investigazione che si concentra in maniera specifica sulla sezione relativa alla memoria, allo scopo di incrementare le performance in termini di realizzazioni fisiche del modello RAM.
L'origine dell'indagine è la rimozione del vincolo di una singola porta per le comunicazioni tra memoria e processore.
Poiché le leggi fisiche non impediscono nulla in questo senso, si esplora la possibilità

di realizzare una memoria con un numero maggiore di porte.
Si avvia così la ricerca di una struttura di memoria nel piano che migliori le prestazioni del sistema, caratterizzata da una topologia a mesh. L'obiettivo è cioè quello di valutare la possibilità di realizzare una memoria che consideri il potenziamento della banda I/O, nel rispetto della *serial consistency*.
Si svolge un'esplorazione sistematica sul ruolo della concorrenza: si indaga sul modo in cui, aumentando la banda, sia possibile trarre vantaggio dal parallelismo, senza pregiudicare la sequenzialità delle istruzioni. In linea di principio si considera il progetto di una *Finite State Machine* nel piano.
L'intenzione è quella di giungere ad una memoria con un'organizzazione *pipelined*, non gerarchica, che risponda a M richieste di accesso in tempo $O(\sqrt{M})$.
Contestualmente, sempre nell'obiettivo di incrementare l'efficienza della memoria, si introduce il processore nella memoria, attraverso unità di elaborazione dedicate.
Un progetto si qualifica solitamente secondo le regole di *extensibility*. La maggior parte dei sistemi non sono estensibili ma, in questo ambito, tale aspetto, approfondito in seguito, è considerato un requisito importante per l'indagine.
Il lavoro esplora, con risultati diversi, due modelli di memoria pipelined non gerarchica. Entrambe le soluzioni presentate sono compatibili con l'estensibilità, assicurata dal fatto che le comunicazioni sono garantite da distanze tra elementi del circuito che sono o costanti o parametrizzate in maniera indipendente dalla taglia della mesh.

1.1 Estensibilità nell'ambito dei vincoli fisici

La progettazione e le prestazioni delle macchine vengono esplorate sotto vincoli di implementazione assolutamente fondamentali, affinché rimangano valide con l'evoluzione della tecnologia e con la costruzione di sistemi di taglia maggiore. Ci si concentra quindi su sistemi che siano *estensibili*, come specificato dalle seguenti proprietà:

1. Il sistema è formulato come un'interconnessione di moduli di funzionalità relativamente semplici, specificando la sua mappatura nel sistema fisico e includendo un posizionamento dei moduli e delle loro interconnessioni nello spazio fisico.

2. La progettazione del sistema è parametrizzata sulla taglia del sistema stesso, in modo tale che lo stesso modello possa essere istanziato nello spazio fisico per taglie differenti.

3. Il tempo di esecuzione di una computazione nel sistema è espresso in numero di passi, come funzione della taglia, dove ciascun passo è una transizione di stato dell'implementazione fisica.

4. Il tempo impiegato per una transizione di stato deve essere invariante al crescere della taglia del sistema.

Il sistema esplorato da questo lavoro deve essere infine implementato in uno spazio bidimensionale, in cui la fisica impone dei limiti inferiori alla taglia del dispositivo e dei limiti superiori alla velocità di trasmissione del segnale.
L'estensibilità costituisce dunque un aspetto tecnico molto importante. Esso può essere assicurato dall'organizzazione, formulata in termini di *celle* che possono essere realizzate con una quantità costante di logica ed essere incorporate in un array bidimensionale, con collegamenti diretti tra le celle che coprano al più un numero costante di lati dell'array.
Questo lavoro considera l'estensibilità un requisito primario di progetto, pertanto la macchina esplorata rientra nella classe appena descritta.

Si fornisce di seguito un piano della trattazione. Esso rispecchia fedelmente il percorso seguito nel lavoro di esplorazione: comprensione e definizione del problema, ricerca e valutazione delle soluzioni.
Nel Capitolo 2, dopo la contestualizzazione dei concetti di memoria *pipelined* non gerarchica, si descrivono le scelte strutturali sul modulo di memoria e la dinamica che coinvolge la banda di I/O.
La soluzione inizialmente indagata è contenuta nel Capitolo 3, che presenta un primo modello di specifiche progettuali. Si espongono la strategia di base e le modalità di implementazione, analizzando poi le prestazioni complessive.
Il Capitolo 4 illustra un modello più evoluto, sviluppato cercando di capitalizzare i margini di miglioramento lasciati dal modello precedente. In esso si descrivono le modalità con cui, attraverso una strategia profondamente rinnovata, si raggiunge l'obiettivo prefissato.
In conclusione si analizza l'avanzamento della ricerca in prospettiva e si profilano alcune direzioni per ulteriori indagini sulle questioni studiate in questo lavoro.

Capitolo 2

Organizzazione della memoria

Questa prima parte della trattazione rappresenta i primi passi dell'investigazione. Essa è dedicata alla contestualizzazione del problema e alle definizioni alla base della ricerca: si introducono i concetti di memoria pipelined nello spazio bidimensionale e viene presentata la struttura di un modulo di memoria.
La discussione si sofferma sulle motivazioni di ciascuna scelta progettuale e su tutti quegli aspetti legati all'introduzione della banda di comunicazione parallela, quali la dinamica di I/O delle richieste e la definizione di un protocollo per il rispetto della serial consistency.
Per *input* della memoria si intende l'insieme delle richieste sottoposte dai processori; l'*output* è invece l'insieme delle richieste soddisfatte, con risposte disponibili per i processori in caso di richieste di lettura.
In maniera coerente con quanto definito nell'articolo di riferimento [2], si mantiene l'ipotesi della lunghezza costante delle parole di memoria, trascurando in tutta l'analisi la relazione logaritmica con la capacità della memoria stessa.
Dopo la presentazione del layout, viene evidenziata, nel capitolo, la necessità di proseguire l'indagine con modelli e algoritmi che gestiscano in maniera accurata tutti gli aspetti del problema.
La nuova organizzazione della memoria, completata quindi dai modelli e unita ad un'opportuna organizzazione del processore, condurrà a macchine per l'esecuzione efficiente di programmi RAM.

2.1 Topologia

La configurazione della memoria nel piano è una scelta decisiva per soddisfare il requisito strutturale di estensibilità.
La topologia che offre le migliori garanzie sotto questo aspetto è la mesh, in cui si costruiscono celle identiche ed equispaziate e si realizzano tra di esse collegamenti

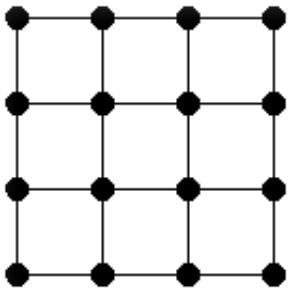


Figura 2.1: Topologia della memoria

near neighbor. L'array bidimensionale quadrato che ne risulta è illustrato in Fig. 2.1.
Una simile topologia rende indipendente la lunghezza dei collegamenti tra nodi dalla taglia complessiva della rete. I nodi sono realizzati con una quantità di logica costante e la distanza tra due vicini prescinde dal numero totale di nodi presenti in memoria. In Fig. 2.2 si osserva in maniera ravvicinata questo particolare: è possibile prolungare in maniera indefinita la mesh con altre celle senza intervenire sui collegamenti già esistenti.
In generale, gli algoritmi sulla mesh in esame non sono veloci come quelli su altre

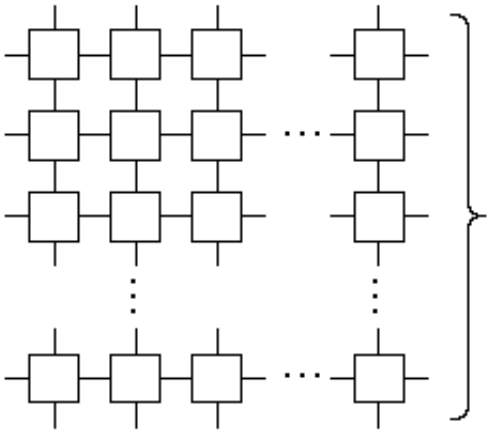


Figura 2.2: Caratteristica della rete mesh: collegamenti *near neighbor* a lunghezza costante

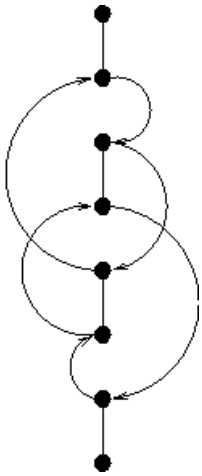


Figura 2.3: Topologia shuffle-exchange

configurazioni collocabili nel piano, come *shuffle-exchange* (Fig. 2.3) o *cube connected cycles* (*CCC*) (Fig. 2.4). In queste ultime, tuttavia, la condizione discussa ed evidenziata in fig 2.2 non si verifica.
Le reti shuffle-exchange e cube connected cycles sono infatti esempi di topologie in cui, se aumenta la taglia della rete, aumenta la lunghezza dei fili necessari a realizzare i collegamenti tra vicini.
È immediato notare che, aumentando il numero di nodi o anche solo la lunghezza del ciclo nel CCC, cresce la lunghezza dei collegamenti.
La proprietà di *bounded degree network* è dunque soddisfatta nella topologia mesh. La località delle comunicazioni, la semplicità degli schemi di interconnessione e la regolarità della struttura garantiscono estensibilità e scalabilità, giustificandone la scelta in questo contesto.

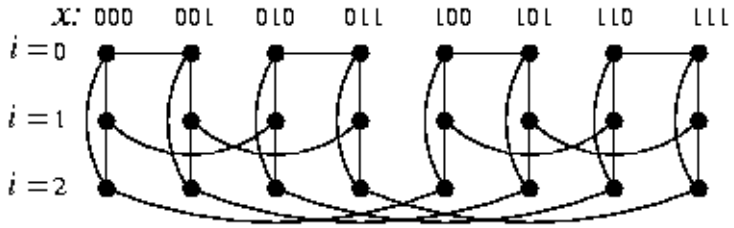


Figura 2.4: Topologia cube connected cycle

2.2 Memoria *pipelined* multiporta

Definizione 1. *Una* ***memoria pipelined multiporta nel piano*** *soddisfa le seguenti proprietà:*

- La memoria consiste di M nodi, con **indirizzi** *0, 1, ... , M-1.* All'interno dei nodi si collocano le celle di memoria primaria. Il contenuto di una cella x (ad un istante di tempo implicitamente conosciuto) è una parola, indicata con m(x).
- La memoria ha $\sqrt{\mathbf{M}}$ ***input port*** attraverso cui accetta le richieste di lettura (R) e di scrittura (W) e $\sqrt{\mathbf{M}}$ ***output queue*** attraverso cui consegna le risposte.
- Una ***richiesta di lettura (R)*** è un pacchetto nella forma $load(x; tag)$; in risposta, la memoria inserisce un pacchetto in una output queue, con l'informazione $(x, m[x]; tag)$. Il campo *tag*, che non viene modificato dalla memoria, fornisce al processore un meccanismo conveniente per associare alla richiesta informazione che risulterà utile nel processare la risposta.
- Una ***richiesta di scrittura (W)*** è un pacchetto nella forma $store(x, v)$ e causa l'impostazione del contenuto $m(x)$ della cella x al valore v.
- I request packets (RPs) diretti alla stessa locazione di memoria hanno effetto nello ***stesso ordine in cui sono sottoposti*** alle input port.
- Input port e output queue sono accoppiate tra loro in maniera biunivoca. Le coppie {input port, output queue} sono equispaziate tra loro sul lato nord della memoria.
- In ciascun passo temporale, una input port può accettare al più un RP e ciascuna output queue può consegnare al più un RP (in risposta ad una precedente richiesta di lettura).
- Le prestazioni della memoria sono caratterizzate dalla ***funzione di accesso*** o ***latenza*** $a(x)$, che denota un limite superiore al tempo che intercorre tra la sottoposizione ad una input port di un RP di lettura per la cella x e la consegna del risultato all'output queue corrispondente.

2.3 Struttura generale di un modulo di memoria

Si descrive un modulo $\mathcal{M}$ di memoria *pipelined bidimensionale* e *non gerarchica.* Esso è organizzato come un array bidimensionale quadrato di taglia M, per comodità una potenza di 4, la cui taglia può essere espressa come $[0..\sqrt{M}-1,0..\sqrt{M}-1]$.

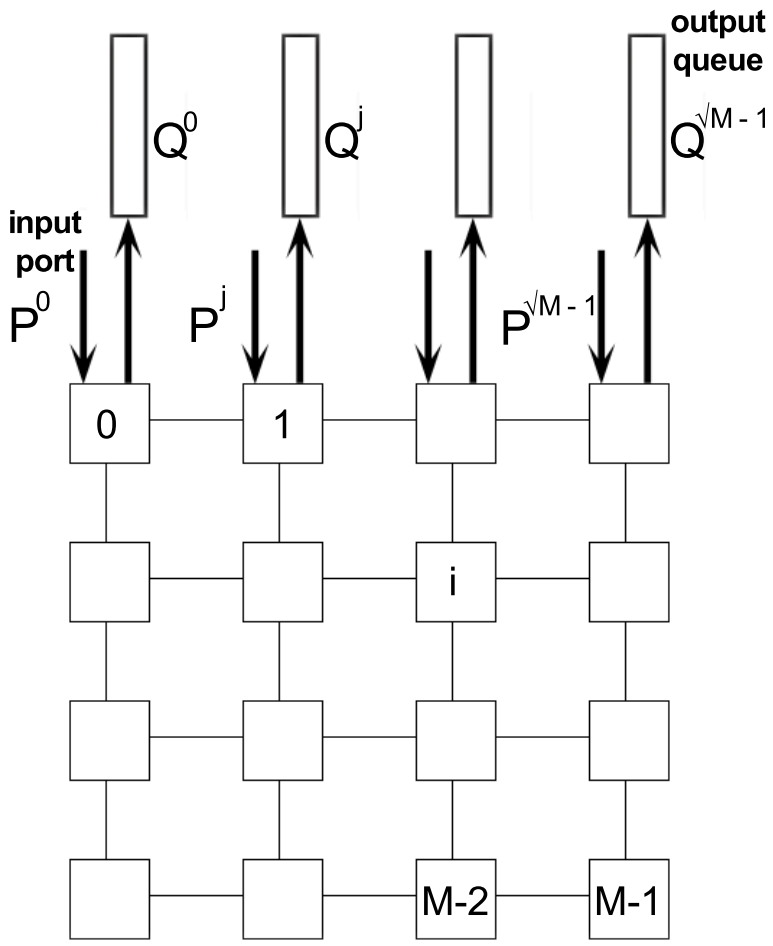


Figura 2.5: Modulo di memoria $\mathcal{M}$

I nodi sono indicizzati secondo lo schema row-major in $[0, M-1]$, in maniera crescente sulle righe della mesh.
Sul lato nord della mesh, lungo la prima riga, sono collocate le *input port*, attraverso le quali le richieste fanno il loro ingresso nella mesh.
Sullo stesso lato della mesh sono situate anche $\sqrt{M}$ *output queue*, le code da cui i processori prelevano le eventuali risposte.
Una questione importante è la sequenzialità delle risposte secondo l'ordine definito dalla semantica. Per specificare le modalità attraverso cui essa viene garantita è necessario conoscere la struttura dei processori, per cui la descrizione in questo punto rimane ad alto livello.

Le ipotesi di soluzione sono varie, con la possibilità di attribuire il compito interamente all'uno o all'altro componente o ad una struttura dati intermedia che coordini i due componenti.
Se il compito fosse affidato completamente alla memoria si prospetterebbero due possibilità di soluzione: una strutturale, sostituendo le code con strutture dati che implementano una diversa politica degli accessi, e una algoritmica, che introduce un *reverse* delle risposte sulle colonne in fase di svuotamento della mesh.
Se invece fosse compito dei processori, essi potrebbero sfruttare i tag nelle risposte per risalire all'ordine temporale.
In fig. 2.5 è rappresentato il layout di un modulo di memoria $\mathcal{M}$.
Input port e output queue costituiscono la banda di I/O della memoria. Come visibile in figura, entrambe sono indicizzate in row-major, seguendo la numerazione $[0, \sqrt{M} - 1]$ dei nodi della prima riga.
Un processore riceve l'eventuale risposta ad una richiesta nella stessa zona in cui essa è stata presentata alla memoria: se una richiesta entra in memoria tramite l'input port P^n, l'eventuale risposta ad essa relativa sarà disponibile nella output queue Q^n.
Le code disaccoppiano la memoria dal processore in maniera asincrona, garantendo ai processori l'accesso alle risposte in coda in modo indipendente dal processo in atto nella memoria.
Oltre alle celle, ciascun nodo contiene un'unità di elaborazione, secondo il modello MIMD di una mesh sincrona. L'elaboratore è strettamente dedicato alle funzionalità di gestione della memoria.
Ulteriori dettagli progettuali sono specificati nei modelli presentati in seguito, che arricchiscono la struttura di base con varie componenti, a seconda dell'idea di soluzione su cui si fondano.

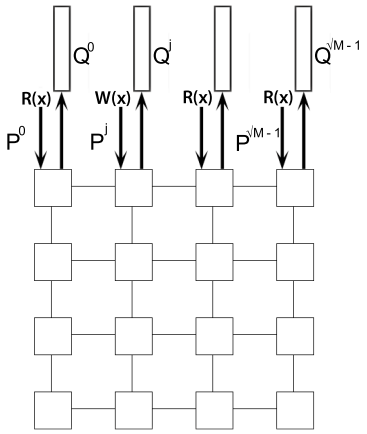
2.4 Semantica

L'introduzione di una banda proporzionale al lato della mesh accresce notevolmente le potenzialità della memoria, aumentando la possibilità di sottoporre in maniera concorrente le richieste. Per assicurare la coerenza della definizione precedente, in termini di effetti delle richieste destinate alla stessa locazione, si pone quindi la necessità di stabilire un protocollo di interazione che definisca un ordine. Le risposte fornite saranno sequenziali secondo questo ordinamento.
Le M richieste che entrano in memoria per essere soddisfatte in maniera concorrente devono essere suddivise in fronti successivi di taglia $\sqrt{M}$, vista la capacità della banda di comunicazione. Attraverso un fronte entrano quindi in maniera concorrente $\sqrt{M}$ richieste concorrenti. Per riempire completamente la mesh si creano

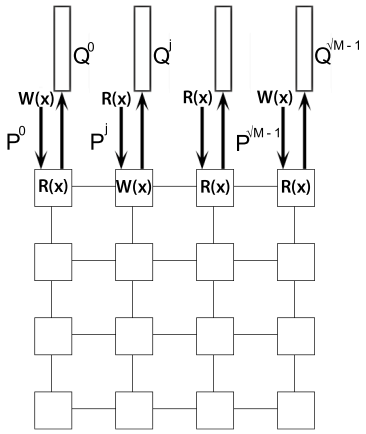
complessivamente $\sqrt{M}$ fronti successivi. L'ordine di fronte è dato dal tempo, in base all'istante in cui le richieste si presentano alle input port.
La priorità si completa con la definizione di un criterio all'interno di un fronte. Si considerano le porte numerate in row-major, da sinistra verso destra sulle righe. Seguendo la medesima numerazione, le richieste risultano ordinate in row-major sul fronte.
Questi criteri di precedenza sono fondamentali per stabilire, a parità di cella di destinazione, le priorità negli effetti delle richieste, rispettando l'ordine con cui sono state presentate dai processori.
Tale logica viene seguita di conseguenza anche per la restituzione delle eventuali risposte nelle code.

2.5 Dinamica di I/O delle richieste

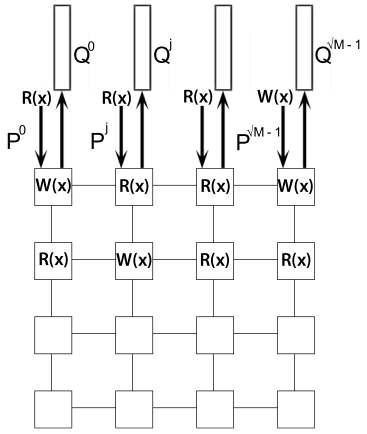
In fase di esecuzione dei programmi, secondo le esigenze che si manifestano nelle varie istruzioni, i processori inviano i request packet corrispondenti alla memoria.
I request packet sono sottoposti alla memoria a fronti di $\sqrt{M}$.
I primi passaggi della fase di input sono rappresentati in figura 2.6. Il primo fronte di richieste è quello in Fig. 2.6a). La memoria, nel successivo passo computazionale, blocca istantaneamente la ricezione degli input e fa avanzare sulla prima riga della mesh le richieste presenti alle porte. Se la mesh non è vuota, i fronti già presenti scendono contestualmente sulla riga inferiore.
Successivamente le input port si sbloccano e tornano disponibili alla ricezione di nuove richieste. Al completamento di un nuovo fronte il processo si ripete (Fig. 2.6b e 2.6c).
Dopo l'ingresso di $\sqrt{M}$ fronti, la mesh contiene un request packet per ogni nodo. Le input port si bloccano definitivamente e non accettano più alcuna richiesta. La fase di input è esaurita e lascia spazio alla fase di elaborazione, in cui le richieste vengono processate e soddisfatte.
L'elaborazione si conclude con la ricezione della risposta da parte dei request packet che contengono una richiesta di lettura. Si procede quindi allo svuotamento della memoria, con l'eliminazione delle richieste dai nodi della mesh. I request packet che contengono richieste di scrittura vengono ora semplicemente cancellati dai nodi in cui risiedono. Gli eventuali packet che contengono letture, ora soddisfatte, sono destinati alle output queue, dalle quali i processori potranno poi prelevarli.
L'inserimento nelle code si realizza tramite un semplice algoritmo di instradamento. Ciascun nodo contenente un request packet con lettura completata trasmette al vicino superiore sulla riga, fino alla prima, che inserisce direttamente nelle output queue.



(a) Ingresso fronte 1



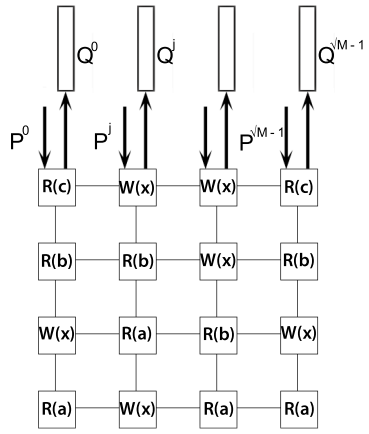
(b) Ingresso fronte 2



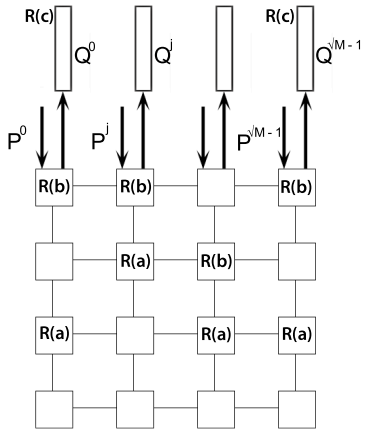
(c) Ingresso fronte 3

Figura 2.6: Dinamica di ingresso delle richieste

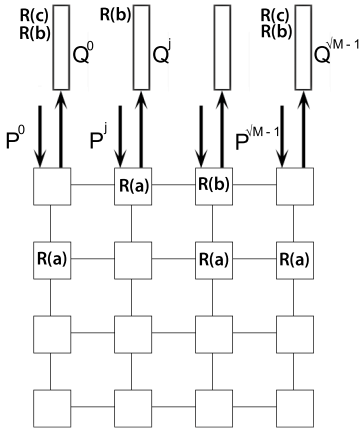
Poiché le code disaccoppiano in maniera asincrona la memoria dai processori, essi possono prelevare in qualsiasi momento l'output disponibile nelle code, indipendentemente dalle fasi in atto nella memoria.
La fase di svuotamento della memoria e restituzione dell'output è un processo atomico e, nel caso peggiore, si risolve in $\sqrt{M}$ passi.
La figura 2.7 illustra la sequenza di passaggi che compongono il processo. Il caso rappresentato richiede proprio $\sqrt{M}$ passi perché su tutti i fronti sono presenti packet che richiedono letture. Per evidenziare la sequenza di ingresso nelle code, le destinazioni delle richieste di lettura sono rappresentate in ordine alfabetico sulle colonne, a partire dalla richiesta del primo fronte. Dopo essere entrati nelle output queue,



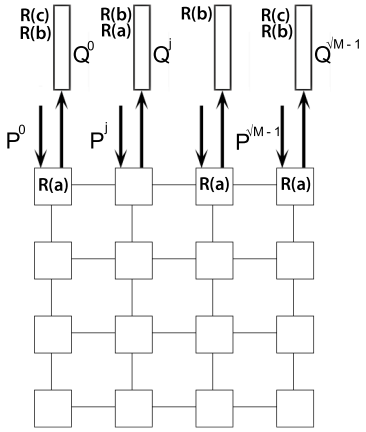
(a) Mesh dopo l'elaborazione



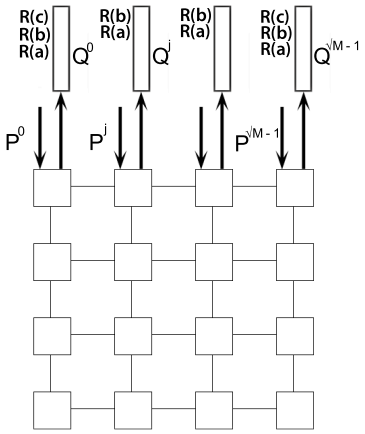
(b) Uscita fronte 1



(c) Uscita fronte 2



(d) Uscita fronte 3



(e) Output

Figura 2.7: Dinamica di svuotamento della memoria

i pacchetti diventano disponibili per i processori, secondo il criterio rappresentato dall'ordine alfabetico.
Con l'ingresso in coda degli ultimi packet soddisfatti, le input port si sbloccano e la memoria torna disponibile all'ingresso di nuove richieste.

2.6 Movimento delle informazioni

Ogni request packet, in base all'istante in cui si presenta ad una input port, fa il proprio ingresso nella rete da un nodo ben preciso. Possono seguire eventuali discese dei fronti, fino al riempimento della mesh. Al termine della fase di input, ciascun request packet si stabilisce in un nodo in maniera definitiva. Esso persiste nella medesima locazione durante tutta la fase risolutiva.
I request packet non si spostano direttamente nella rete. Infatti, non tutti i campi contenuti in esso sono utili per il soddisfacimento della richiesta, perciò si evita di trasportare continuamente informazione non necessaria. È il caso del campo *tag* per le richieste di lettura, utile solo ai processori al termine delle operazioni. In secondo luogo, nessun sottoinsieme dei campi in questione è sufficiente per l'elaborazione: sono necessarie le informazioni relative al nodo di ingresso. Si preferisce però evitare di intervenire direttamente sulla richiesta originale aggiungendo campi ulteriori. Infine, la scelta tiene conto anche del fatto che i processori prelevano la risposta dall'output queue accoppiata all'input port attraverso cui è stata sottoposta la richieste.
In generale, le informazioni sono diffuse sulla rete attraverso l'inoltro di messaggi tra nodi vicini, sfruttando la topologia. Allo stesso modo, le richieste sono trasmesse sulla rete tramite dei messaggi che trasportano le informazioni necessarie a soddisfarle.
Il primo passaggio della fase di elaborazione prevede proprio questo.
Di seguito, per non appesantire la notazione, vengono omesse dal formato dei messaggi le informazioni relative al nodo mittente e al nodo destinatario. Esse vengono menzionate nella trattazione e considerate implicite nel messaggio.
Ogni nodo, a partire dal request packet che contiene, crea un apposito messaggio che veicola la richiesta. Il suo formato è:

$$\langle \textit{indirizzo richiesta},\ \textit{tipo richiesta},\ \textit{indice nodo} \rangle$$

L'ultimo campo è indispensabile per rispettare la semantica: da esso è possibile risalire al fronte d'ingresso e alla posizione del request pack all'interno del fronte.
L'eventuale messaggio di risposta, destinato al nodo in cui risiede un request packet di lettura, è invece in formato:

$$\langle \textit{valore acquisito} \rangle$$

Il suo contenuto va a completare il campo $m[x]$ del request packet.
In tutta la trattazione, una richiesta è considerata soddisfatta già con l'elaborazione del messaggio che la trasporta, sia in caso di scrittura che di lettura. L'invio degli eventuali messaggi di ritorno è considerato un passaggio scontato e demandato al nodo in cui avviene l'elaborazione.
Inoltre, a seconda della strategia implementata nei diversi modelli progettuali, possono rendersi necessari ulteriori messaggi intermedi per la trasmissione di particolari informazioni. Essi sono discussi nella presentazione del modello specifico.

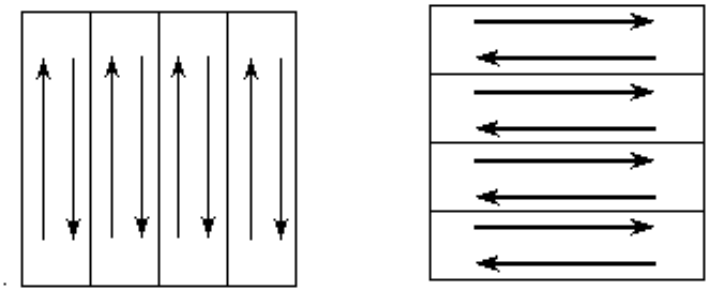
2.7 Una prima soluzione intuitiva

Con una simile struttura di memoria e un meccanismo di ingresso come quello descritto potrebbe essere intuitivo pensare di risolvere tutte le richieste che vengono sottoposte alla memoria in modo piuttosto immediato. Lo strumento che sembra essere più indicato è un algoritmo di *routing on-line* sulla mesh.
Un tale algoritmo, che si può chiamare *colonna-riga*, si basa su un'idea molto semplice: due spostamenti successivi lungo le dimensioni della mesh, prima la colonna, poi la riga, come si può vedere in fig. 2.8. Una logica di questo tipo, semplice anche da implementare, prevede due soli passaggi precisi:

1. In parallelo su tutte le colonne, le richieste di una colonna si spostano lungo la stessa fino ad individuare la riga che contiene il nodo destinazione (Fig. 2.10a)

2. In parallelo lungo le righe individuate, le richieste si spostano fino a trovare il nodo di destinazione (Fig. 2.10b)

Un'analisi più attenta permette di concludere però che una simile soluzione porta facilmente a congestione. Nei casi peggiori tutte le richieste della mesh insistono



(a) Step 1 (b) Step 2

Figura 2.8: Dinamica dell'algoritmo colonna-riga

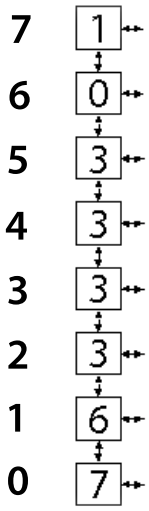
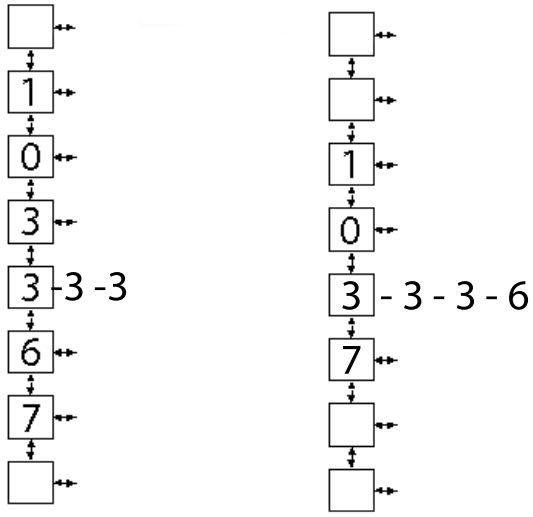


Figura 2.9: Richieste iniziali sulla colonna

su pochi nodi, fino al caso limite in cui tutte convergono simultaneamente ad un solo nodo. Questa situazione non è conciliabile con le caratteristiche del nodo, né in caso di taglia costante né in caso di taglia più che costante, che porterebbe ad un miglioramento solo nel caso limite di una capacità $\theta(M)$ per ciascun nodo.
Già durante la prima fase, se le richieste sono destinate in prevalenza ad una riga specifica, i nodi sulla colonna che si trovano sul percorso verso questa riga sono soggetti a congestione.
Per illustrare una situazione di questo tipo si consideri una mesh quadrata di taglia $M = 64$, le cui righe sono numerate secondo la definizione, a partire dalla più vicina alle porte. Su di essa è sufficiente esaminare una colonna, che contiene inizialmente le richieste rappresentate in Fig. 2.9. A sinistra si vedono gli indici delle righe mentre i nodi contengono schematicamente solo gli indici delle righe di destinazione delle richieste, per evidenziare il fulcro della questione.
La congestione si manifesta già dopo le prime iterazioni dell'algoritmo (Fig. 2.10): molti messaggi convergono ad un nodo in particolare, sia in quanto destinatario, sia in quanto punto intermedio verso le righe superiori, aumentando ad ogni passaggio i messaggi in coda nel nodo.
Il fenomeno poi può ripetersi o manifestarsi nella seconda fase dell'algoritmo.
La congestione aumenta in maniera considerevole la latenza, fatto che impone di escludere questa prima proposta, troppo semplicistica, per progettare soluzioni alternative, che sfruttino meglio la struttura a disposizione.



(a) Iterazione 1 (b) Iterazione 2

Figura 2.10: Iterazioni successive

2.8 Ottimalità e modelli

Nei modelli illustrati in seguito, che forniscono ulteriori specifiche di progetto, sono utilizzati diversi algoritmi, presentati in termini di complessità asintotica. Alcuni di essi costituiscono delle alternative per la soluzione di un sottoproblema. In queste situazioni, la scelta dell'algoritmo ottimo non è affatto banale. Le modalità per identificare l'algoritmo migliore non sono scontate e necessitano di un chiarimento. Gli aspetti da considerare sono sostanzialmente due: il fatto che l'analisi asintotica nasconda le costanti e la scelta del modello di esecuzione in cui confrontare gli algoritmi, per verificare l'ipotesi fondamentale sulla loro comparabilità.
Nell'analisi degli algoritmi è frequente considerare il modello *classico* di mesh, in cui, in un passo, il nodo può:

- eseguire un numero costante di operazioni elementari di elaborazione
- trasmettere/ricevere un pacchetto lungo ciascuno dei propri collegamenti ai vicini
- trasferire informazione al proprio interno.

Nelle ipotesi del modello di questo progetto, tuttavia, quella che viene considerata unità di tempo nel modello classico, risente, dal punto di vista fisico, delle quantità

di logica e di memoria presenti nel nodo. L'area A occupata dal layout del nodo è infatti strettamente legata a queste due quantità.
Il tempo effettivamente necessario per eseguire un algoritmo è dato dal prodotto tra il numero di passi e la durata di un passo di computazione, cioè il periodo di clock. Avvicinandosi alla realizzazione fisica, l'analisi non può fermarsi al numero di passi ma deve interessare anche la durata del periodo di clock. In questo contesto, a parità di condizioni, la durata dei passi di computazione e di comunicazione dipendono almeno dalla dimensione lineare del nodo, cioè sono almeno proporzionali a $\sqrt{A}$.
Nella trattazione gli algoritmi verranno quindi presentati con la loro complessità asintotica ma solo uno studio dettagliato, che tenga in considerazione gli aspetti discussi, potrebbe rivelare quale sia effettivamente il migliore in caso di alternative. Complessivamente, è necessario trovare il miglior compromesso tra area, clock e numero di passi necessari alla risoluzione.

Capitolo 3

Il modello iniziale

Un semplice algoritmo di instradamento non è sufficiente a risolvere tutte le problematiche che si celano dietro la gestione di una struttura di memoria e delle richieste che vi giungono da parte dei processori della macchina.
Un simile macro-processo deve essere necessariamente scomposto e approfondito più in dettaglio: è indispensabile comprendere il ruolo di ciascun componente, le funzionalità di cui deve essere dotato e analizzare con strategia i vari sotto-problemi, per individuare poi gli algoritmi che li risolvono nel modo più adeguato.
In questo capitolo si presenta un primo modello di gestione della memoria, caratterizzato da una corrispondenza biunivoca tra una cella di memoria e un nodo della mesh. Secondo questa ipotesi di soluzione le richieste sono soddisfatte mediante una strategia intuitiva di recapito-risoluzione; la fase cruciale per rispondere ad una richiesta è a carico del rispettivo destinatario.
Il modello tiene conto dei limiti dimensionali del nodo nella progettazione e nella scelta dei vari algoritmi.

3.1 Il nodo

Il nodo è concepito fondamentalmente come la locazione di una cella di memoria primaria. Esso può essere identificato come il contenitore e il gestore di questa porzione di memoria. A ciascun nodo è associato un indirizzo, attraverso il quale è possibile riferirsi alla relativa cella.
La memoria riceve richieste di scrittura (W) o di lettura (R) indirizzate dai processori della macchina alle varie locazioni. Il nodo si occupa di ricevere e soddisfare le richieste destinate alla propria cella e mantenere coerenti i dati memorizzati al suo interno.
La struttura del nodo è fortemente incentrata sulla cella di memoria e strettamente funzionale alla sua gestione. Il nodo dunque deve contenere anche le strutture per

Figura 3.1: Struttura del nodo

svolgere funzioni minime di elaborazione.
La struttura schematica del nodo può essere visualizzata in figura 3.1. Il nodo, oltre alla cella di memoria primaria, è fornito di un'unità di elaborazione e di una coda di taglia costante (*working queue*). Questa struttura dati costituisce una memoria di servizio, per mantenere temporaneamente le richieste che giungono al nodo, in attesa di essere soddisfatte. La taglia costante impone una certa oculatezza nel recapitare le richieste al nodo destinatario. Se un nodo è destinatario di molte richieste, non può mantenerle tutte in attesa al proprio interno, ma deve prima soddisfarne alcune e liberare così spazio nella coda. Un discorso analogo è valido anche per le richieste di lettura soddisfatte: una volta acquisito il valore, per il medesimo problema dimensionale, non possono essere mantenute in memoria in vista di un invio successivo al processore mittente.
L'unità di elaborazione è dotata di funzionalità elementari: scrittura/lettura dalla cella di memoria e inoltro/ricezione di messaggi, per movimentare le richieste provenienti dai processori. Essa è tuttavia priva di funzioni aritmetico-logiche e si dedica esclusivamente alla gestione e al controllo della cella di memoria.
Se si vuole qualificare il nodo dal punto di vista delle funzionalità, in questo modello esso si caratterizza maggiormente come componente di memoria primaria che come unità di elaborazione.

3.2 Strategia risolutiva

Lo schema che questo modello utilizza per soddisfare tutte le richieste sottoposte alla memoria è lineare e intuitivo.
Date le condizioni iniziali, con la mesh contenente in ciascun nodo una richiesta di accesso ad una locazione di memoria, l'idea è quella di recapitare ciascuna richiesta al proprio indirizzo di destinazione. A seconda del fronte e all'input port d'ingresso, prima di giungere alla cella indirizzata, ciascuna richiesta può attraversare un numero di nodi proporzionale al diametro della mesh,. È un caso particolare e favorevole quello in cui il nodo che contiene il request packet e l'indirizzo di destinazione della stessa coincidono. Il principio di funzionamento di questo modello prevede che ogni nodo, oltre che come locazione temporanea della richiesta, funga poi da instradatore e inoltri il messaggio di richiesta verso il nodo che contiene effettivamente la cella di destinazione.
Una volta giunta al nodo contenente la cella in questione, la richiesta viene processata e soddisfatta. Per il nodo la gestione delle richieste di lettura e di scrittura prevede una ovvia ma sostanziale differenza: le richieste di lettura (R), per essere soddisfatte, richiedono la trasmissione del valore acquisito al nodo contenente il request packet corrispondente . Solo dopo questo passaggio esse possono dirsi esaurite ed essere eliminate dal nodo. Le richieste di scrittura (W) si esauriscono invece

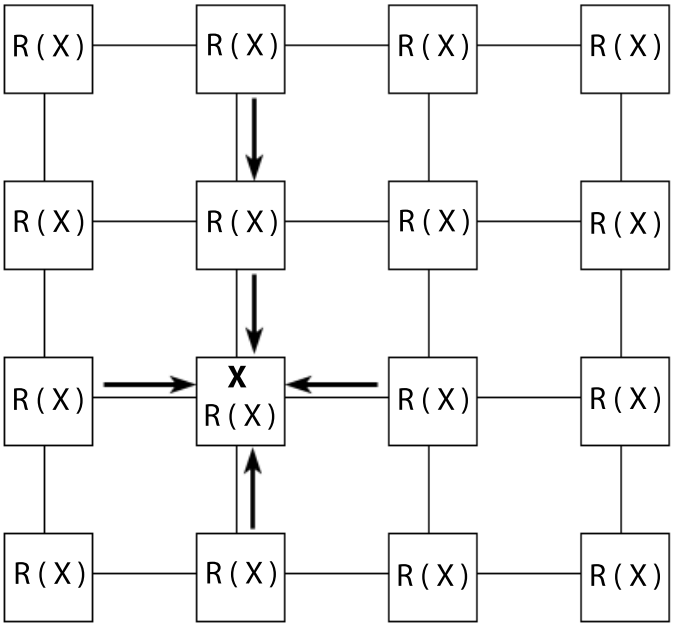


Figura 3.2: Caso $k = M$ per un nodo X

semplicemente con l'aggiornamento del valore nella cella primaria e possono essere poi eliminate.
Secondo quanto discusso in precedenza, ogni nodo è mittente di una sola richiesta ma, a sua volta, destinatario di un numero variabile k di richieste. Questa quantità non è però nota a priori, condizione che qualifica il problema in modalità $on-line$. Il *range* di valori in cui si definisce k è $[0, M]$, cioè un nodo, in riferimento alla cella di memoria che contiene, può non ricevere alcuna richiesta così come essere il destinatario di tutte quelle entrate nella mesh. k è perciò un numero potenzialmente grande, di ordine $\theta(M)$. Qualora si verificasse questa condizione, si pone un problema nella gestione delle richieste relative ad un nodo, poiché esso dispone di una quantità costante di spazio nella propria working queue.
La figura 3.2 è un esempio del caso più sfavorevole in assoluto, con la mesh di taglia $N = 16$ che contiene solo richieste di lettura tutte destinate al medesimo nodo, indicato con X. È evidente come il nodo risulti congestionato dal loro arrivo, situazione che si aggrava considerando taglie sempre maggiori della mesh.
Per superare questo problema è necessario adottare una politica di spianamento o diffusione, secondo la quale il nodo, per far spazio ad un'altra richiesta, si libera immediatamente della richiesta soddisfatta, eventualmente distribuendola ai nodi vicini. Quest'ultima eventualità si prospetta in caso di richieste di lettura. In questa situazione, le problematiche di instradamento, gestione e redistribuzione delle richieste si sovrappongono e si interlacciano.
Quando tutte le richieste sono soddisfatte e si raggiunge l'uniformità nella distribuzione tra nodi delle eventuali richieste di lettura soddisfatte, si procede a riportare queste ultime ai nodi della mesh da cui sono entrate.
Il passo finale e conclusivo è lo svuotamento della mesh e l'inserimento delle richieste soddisfatte nelle code, dalle quali i processori le prelevano per l'elaborazione.

3.3 Implementazione della strategia

Conoscendo le potenzialità di elaborazione a disposizione, è ora possibile rendere operativa la strategia descritta. Il processo di implementazione è immediato, con la risoluzione di ciascun sottoproblema individuato nella strategia mediante un algoritmo adeguato.
Inizialmente, dopo l'ingresso dell'ultimo fronte di richieste, ciascuno degli M nodi della mesh contiene un *request packet*. A partire dalle richieste informative contenute in esso, ciascun nodo produce un messaggio che deve essere inviato al nodo contenente la cella di destinazione della richiesta. Quindi, secondo quanto descritto, il primo sottoproblema da risolvere è: instradare il messaggio di ciascun nodo in modo che un qualsiasi processore sia destinatario di al più M pacchetti. La questione si conforma chiaramente come un problema di *routing* $1-k$, con $k = M$. La

soluzione risiede pertanto nell'utilizzo dell'algoritmo deterministico per il routing 1 - k sulla mesh, riportato nella sezione 3.5.1.
A causa della taglia limitata dei nodi, nel caso in cui un numero eccessivo di messaggi tenti l'ingresso nel medesimo nodo, si presentano degli ovvi problemi di congestione. In queste condizioni si presenta dunque un secondo sottoproblema, che richiede l'impiego di tecniche di spianamento delle richieste ad un nodo. Esso può cercare di scaricare temporaneamente le richieste in eccesso appoggiandosi ai nodi limitrofi.
Questa problematica di diffusione, qualora si manifesti, rappresenta la questione più spinosa dell'intero modello. Per la sua soluzione è necessario sviluppare un algoritmo specifico, studiato appositamente per il contesto. Alcune idee più precise sono discusse e analizzate nella sezione 3.5.2.
L'ultimo sottoproblema previsto dal procedimento risolutivo è legato al ritorno ai nodi d'ingresso dei valori prelevati dalle richieste di lettura. Ciascun nodo contiene al più un messaggio da inviare. Inoltre, al più un nodo invia un messaggio a un qualsiasi altro nodo. Si tratta nuovamente di un problema di instradamento, che tuttavia si configura in maniera diversa dal precedente. In queste condizioni il routing è vincolato $1 - 1$ e risulta quindi semplificato, poiché preclude i problemi legati alla congestione, che si manifestano quando molti messaggi tentano di entrare nello stesso nodo di destinazione. Si cerca perciò un metodo per la risoluzione del caso nella classe di algoritmi per le permutazioni. L'algoritmo individuato è discusso in sezione 3.5.3.
A questo punto è possibile procedere allo svuotamento completo della mesh attraverso l'apposito algoritmo che agisce in parallelo sulle colonne. Esso procede al trasferimento delle richieste soddisfatte nelle code preposte, rendendole così disponibili ai processori per l'elaborazione.

3.4 Capacità e indirizzamento della memoria

Un modulo di memoria realizzato secondo le specifiche definite da questo modello ha una capacità complessiva pari a M. Secondo quanto descritto nell'architettura del nodo, c'è infatti una corrispondenza biunivoca tra un nodo ed una cella di memoria. La mappatura delle celle nei nodi si risolve tramite una semplice uguaglianza. I nodi della mesh sono indicizzati secondo lo schema row-major e le celle seguono lo stesso schema di indirizzamento.

3.5 Algoritmi e analisi di complessità

3.5.1 Algoritmo per il routing 1-k

L'algoritmo individuato per risolvere il problema del *routing* $1-k$ sulla mesh è SHORTQROUTE di Sibeyn e Kaufmann [8]. Esso è stato scelto perché si conforma molto bene alla condizione di spazio costante nei nodi. Anziché trascurare la taglia della coda, SHORTQROUTE mira infatti al raggiungimento della miglior complessità per il routing nella mesh, in condizioni di *working queue* ≤ 4 *messaggi* nei nodi.
L'algoritmo parte da una suddivisione della mesh in k quadrati, di taglia $\sqrt{M}/\sqrt{k} \times \sqrt{M}/\sqrt{k}$, che giacciono su di un ciclo hamiltoniano. L'idea fondamentale è poi quella di distribuire le richieste sui quadrati, in modo tale che ci sia al più un messaggio con una certa destinazione in ciascun quadrato. Questo passaggio permette di ridurre il problema iniziale ad un problema di *routing* $1-1$ *parziale* (ossia di permutazioni parziali) all'interno dei quadrati.
Un miglioramento delle prestazioni deriva infine dallo sfruttamento di una tecnica di colorazione alternata (bianco e nero) dei messaggi e dal criterio di instradamento sempre ortogonale.
Come già esposto in precedenza, in questa dinamica di funzionamento della memoria, il valore di k non è noto a priori. Un'ulteriore caratteristica per cui l'algoritmo si adatta molto bene a questo contesto è data proprio dal fatto che, durante l'esecuzione, esso riesce a determinare e far conoscere k, in *broadcast* a tutti i nodi della mesh, in tempo $O(\sqrt{M})$.
Complessivamente l'algoritmo esegue il routing $1-k$ in $6/\sqrt{2} \cdot \sqrt{k} \cdot \sqrt{M} + 6 \cdot \sqrt{M} + O(k^{\,5/8} \cdot M^{\,3/8})$ passi e con taglia della working queue pari a 3.
Si conclude quindi che per tutti i $k = O(M)$ il tempo per il routing $1-k$ è limitato da $O(\sqrt{k} \cdot \sqrt{M})$. Nel caso peggiore, con richieste di sola lettura ad unico nodo nella mesh, il routing diventa $1-M$, con un tempo di esecuzione limitato $O(M)$.

3.5.2 Algoritmi di spianamento

Individuare una soluzione efficiente per lo spianamento non è affatto immediato. La difficoltà è legata al fatto che la questione si interlaccia alle problematiche di instradamento, con potenziale rallentamento di tutta l'esecuzione della strategia.
Come ampiamente discusso, un nodo, nel caso peggiore, non è in grado di accogliere tutte le richieste di cui è destinatario. Si tratta dunque di trovare, contestualmente all'instradamento, una sistemazione temporanea a queste richieste, in attesa che si liberi spazio nel nodo di destinazione. La soluzione più comoda sarebbe quella di piazzare queste richieste, momentaneamente in eccesso, nei nodi vicini al destinatario. Non è detto tuttavia che questo sia possibile, come nel caso di richieste

concentrate su una limitata porzione della memoria. In tal caso è necessario instradare il messaggio in una qualche direzione, alla ricerca del primo nodo con spazio disponibile. La direzione della ricerca può essere intrapresa secondo criteri probabilistici legati alla posizione del nodo nella mesh oppure stabilita dopo la diffusione di messaggi broadcast.
In alternativa si possono instradare i messaggi in eccesso attraverso cerchi concentrici che amplino ad ogni iterazione il proprio raggio d'azione, a partire dai vicini del destinatario e proseguendo eventualmente nei nodi ad essi più esterni.
Un'altra possibilità è il livellamento dei messaggi sulle due dimensioni della mesh, con un travaso continuo, tra le righe e tra le colonne, dei messaggi in eccesso.
Al di là del metodo specifico utilizzato, lo spostamento di un messaggio in eccesso richiede, nel caso peggiore, un transito fino ad un nodo diametralmente opposto. Il numero di passi richiesti per l'approdo ad un nodo provvisorio e poi il ritorno al nodo di destinazione è $O(\sqrt{M})$. Se si considera che l'operazione può ripetersi potenzialmente per una quantità di messaggi $O(M)$, l'intero processo richiede complessivamente $O(M)$ passi.
Ad essi si aggiungono inoltre quelli relativi alle comunicazioni per richiamare i messaggi al nodo destinatario, per diffondere sulla mesh le informazioni relative a nodi con spazio libero o per le interrogazione di altri nodi da parte dei nodi congestionati.
Una simile complessità computazionale condiziona pesantemente l'intero procedimento risolutivo, rendendolo di fatto inefficiente. Viste le prospettive, è dunque poco conveniente approfondire l'indagine in questa direzione. Per questo motivo in questa sezione, diversamente dalle altre, ci si limita all'esposizione e all'analisi di alcune idee base per un eventuale algoritmo, evitando di soffermarsi sulla presentazione di uno pseudocodice completo.
A rendere ulteriormente poco percorribile questa soluzione contribuisce il fatto che l'implementazione di un algoritmo così complesso e articolato richiede una quantità consistente di logica nel nodo, aumentandone così la dimensione e il tempo di clock.

3.5.3 Permutazioni sulla mesh

Il primo metodo per risolvere in modo ottimo il problema del routing 1 - 1 è presentato da Leighton, Makedon e Tollis in [4]. Questo algoritmo di instradamento è specifico per array bidimensionali con code di taglia costante, proprio come si verifica nella nostra mesh. La filosofia dell'algoritmo prevede la distinzione tra *messaggi critici* e *messaggi non critici*, con i primi che, diversamente dagli altri, partono da un angolo della mesh e terminano nell'angolo opposto. Le prestazioni di un algoritmo di routing sono spesso limitate proprio da questi messaggi critici, motivazione alla base della loro denominazione. Tramite un semplice argomento di diametro è immediato dimostrare che il tempo minimo per instradarli sulla mesh è $2(\sqrt{M}-1)$.

A partire da questa classificazione, discende un metodo di routing diverso per le due categorie. Per instradare sulla mesh i messaggi non critici si usa fondamentalmente l'algoritmo di Kunde [3] mentre per i messaggi critici si sfrutta un metodo di instradamento ricorsivo sugli angoli della griglia.
L'utilizzo di code di taglia costante è garantito attraverso l'accorgimento che non siano contenuti messaggi nella regione in cui ha luogo la ricorsione sul routing.
Complessivamente l'algoritmo risolve il problema in $2\sqrt{M} - 2$ passi e, in base alle considerazioni sui messaggi critici, risulta quindi ottimo nel caso peggiore.
Un altro algoritmo deterministico ottimo è proposto da Sibeyn, Chlebus e Kauffman [9]. Esso si fonda sulla stessa classificazione dei messaggi di [4] ma, attraverso tecniche di dispersione e diffusione (*scattering e spreading*), lavora con code di taglia sensibilmente inferiore.
In questa situazione, solo un eventuale progetto completo a livello di porte logiche nel nodo può stabilire quale sia l'algoritmo più vantaggioso da utilizzare. Il miglioramento della taglia della coda potrebbe comportare infatti la richiesta di circuiti logici di dimensioni maggiori, quindi con un periodo di clock maggiore.

3.6 Problemi del modello

Le prestazioni della memoria che si ottengono implementando questo primo modello sono in generale piuttosto scarse. L'eventualità che le performance degradino notevolmente è piuttosto concreta.
La latenza può crescere notevolmente nel momento in cui si manifesta la necessità di gestire le numerose richieste indirizzate ad uno stesso nodo della rete.
Si ripropone, nel caso peggiore, una problematica simile a quella dell'algoritmo di instradamento colonna-riga, con i nodi sovraccaricati di messaggi rispetto alla propria capienza.
L'inefficienza di fondo è imputabile alla strategia di base. La situazione è lampante nel caso limite di un nodo destinatario di M richieste. Nella mesh si trova un nodo congestionato e altri $M - 1$ nodi scarichi, con unità di elaborazione pressoché inutilizzate. Lo sforzo computazionale è totalmente a carico del destinatario dei messaggi. Solo in seconda battuta, in caso di emergenza, si è costretti a ricorrere a dei meccanismi per alleggerire momentaneamente il carico su tale nodo. Questa è però solo una dilazione del carico e comporta una crescita della latenza che può risultare incontrollata nelle configurazioni più critiche.
Il problema si manifesta con sfumature diverse anche nel caso di richieste tutte concentrate su una piccola porzione della mesh.
In tali casi, l'aumento della taglia della memoria può peggiorare ulteriormente la problematica.
A fronte di un numero di messaggi direttamente proporzionale ai request packet, i

costi delle comunicazioni possono raggiungere livelli molto elevati nell'eventualità di dover movimentare continuamente questi messaggi.
Questo primo modello sfrutta solo marginalmente la concorrenza e, nella fase cruciale della risoluzione, il grado di parallelismo è molto basso. Esso risulta dunque scarsamente scalabile.
Vista la struttura a disposizione, è opportuno rielaborare la strategia in un'ottica più scalabile.
I margini di miglioramento sono quindi notevoli sotto diversi aspetti.

Capitolo 4

Un modello più evoluto

Le problematiche evidenziate nel modello precedente costituiscono un valido spunto di riflessione e un ottimo punto di partenza per proseguire l'esplorazione sulla memoria nella macchina fisica, nel tentativo di migliorarne le prestazioni.
Nasce così un nuovo modello, più complesso ed evoluto. Esso supera il precedente, che diventa strutturalmente un suo caso limite. L'approccio nella progettazione è completamente diverso e l'intero procedimento di risoluzione delle richieste si fonda su una strategia efficace ed efficiente. Questa strategia è presentata prima di qualsiasi definizione di funzionalità e componenti, perché essa è la chiave del sistema.
La struttura generale della memoria non subisce modifiche rilevanti ma cambia in maniera sostanziale l'organizzazione interna e il ruolo dei suoi componenti principali.
La strategia iniziale è implementata nei vari passaggi attraverso algoritmi specifici, che sfruttano il più possibile, attraverso i collegamenti della mesh, la concorrenza negli accessi alle celle di memoria e nello scambio di messaggi tra nodi.
Il modello che ne scaturisce, presentato in questo capitolo, si avvale di strumenti più potenti rispetto al modello precedente, per distribuire più uniformemente il carico computazionale e soddisfare così in maniera efficiente le richieste di accesso alla memoria primaria.

4.1 Strategia risolutiva

La strategia è alla base di tutto il meccanismo di gestione delle richieste di accesso alle celle di memoria. Il suo obiettivo è quello di soddisfarle in maniera efficiente, minimizzando nello stesso tempo il numero di accessi alle celle stesse.
L'idea è lavorare all'interno di ciascuna sequenza di richieste ad una specifica locazione di memoria primaria, sfruttando la conformazione della sequenza stessa. Le operazioni preliminari da svolgere sono quindi separare per indirizzo di destinazione

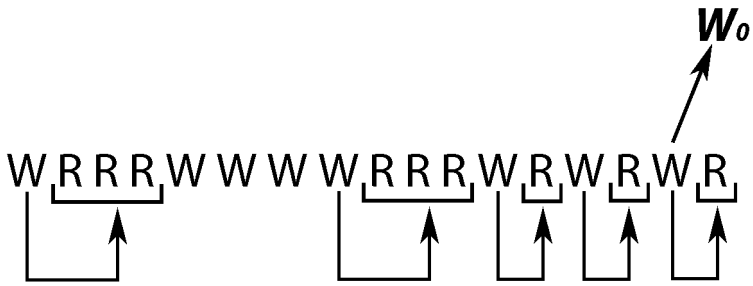


Figura 4.1: Caso più favorevole di sequenza: un solo accesso richiesto

le richieste e disporle in ordine di sottoposizione alla memoria centrale. Il punto chiave è poi soddisfare una richiesta attraverso l'informazione in possesso della richiesta precedente, con un semplice trasferimento che non coinvolge la cella indirizzata.
Secondo questa logica, la successione delle richieste si può pensare come una catena lungo la quale si propaga l'informazione. Gli anelli della catena sono i nodi che contengono le richieste in ordine di arrivo. Ciascun nodo è normalmente passivo e si attiva alla ricezione di informazione dal suo predecessore; l'attivazione gli comporta contestualmente l'incarico di trasferire a sua volta informazione al successore, che diventa così il nodo attivo. La condizione di attivazione è istantanea e coinvolge progressivamente tutti i nodi lungo la catena, secondo l'ordine delle richieste. È evidente però che ci deve essere un innesco a questa propagazione di informazioni, chiaramente in coincidenza del primo nodo della catena.
Le richieste che si possono definire come portatrici di informazione, quindi in grado di soddisfare altre richieste, sono le scritture (W) e le letture (R) soddisfatte. In base

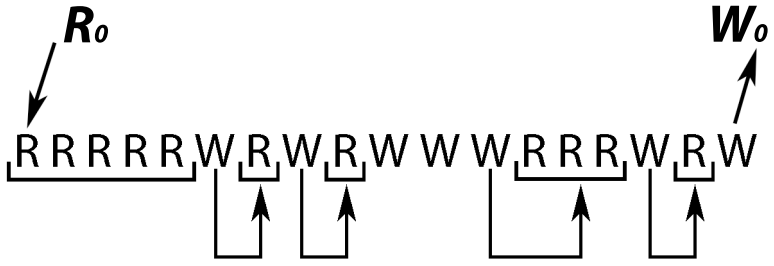


Figura 4.2: Caso meno favorevole di sequenza: entrambi gli accessi richiesti

alla tipologia della prima richiesta della sequenza è dunque possibile dover coinvolgere la cella di memoria indirizzata. Se essa è una richiesta di lettura, è necessario acquisire il valore direttamente dalla cella di memoria, per soddisfarla e creare un primo portatore di informazione. Se la sequenza si apre invece con una richiesta di scrittura, ciò non è necessario perché essa è già portatrice di informazione.
L'altra situazione in cui è necessario accedere alla cella di memoria si verifica quando l'ultima informazione propagata proviene da una scrittura (W). In questo caso si aggiorna il valore scritto nella cella con quello dell'ultima scrittura della sequenza, per rendere consistenti le informazioni nel sistema.
Le figure 4.1 e 4.2 riportano esempi di configurazione delle sequenze. Le richieste di lettura consecutive sono raggruppate per evidenziare come la modalità a catena della propagazione, attraverso cui sono soddisfatte in successione a partire dalla prima del gruppo. Le frecce sottostanti indicano la fonte dell'informazione propagata. L'eventuale freccia superiore indica invece la scrittura conclusiva, da cui aggiornare la cella di memoria con l'opportuno valore, indicato con W_0.
La successione in 4.1 illustra l'avvio della propagazione a partire da una richiesta (W). La configurazione rappresentata rientra nella situazione più favorevole, in cui è necessario un solo accesso, in tal caso per l'aggiornamento finale. È intuitivo comprendere che ogniqualvolta la sequenza si apra con un richiesta di scrittura è necessario un solo accesso alla cella di memoria. Altri casi favorevoli sono costituiti da sequenze omogenee, o tutte di scrittura o tutte di lettura.
La fig. 4.2 è invece un esempio del caso meno favorevole poiché per acquisire il valore R_0 e trasferire W_0 richiede entrambi gli accessi descritti in precedenza.
In entrambe le figure si vedono delle richieste di scrittura che sono seguite da altre richieste di scrittura e non soddisfano quindi alcuna richiesta di lettura successiva. Esse corrispondono ad una serie di sovrascritture al valore nella cella.
La chiave di lettura di questa strategia è proprio quella di evitare le operazioni ininfluenti quali le scritture che vengono sovrascritte e le letture di un valore già letto. Questa è proprio l'intuizione che consente di minimizzare le letture/scritture al contenuto della cella. Secondo quanto descritto il numero di accessi si riduce a uno nel caso migliore e due nel caso peggiore. In generale quindi l'ordine degli accessi è costante e indipendente dal numero di richieste che compongono la sequenza.
I passi finali sono analoghi al modello precedente, con il ritorno delle richieste di lettura soddisfatte ai nodi contenenti il request packet corrispondente. Con lo svuotamento della mesh, questi verranno inseriti nelle apposite code, da cui i processori potranno successivamente prelevarli.

4.2 Il nodo: nuove funzionalità

In questo nuovo modello di memoria *pipelined* il concetto di nodo viene sostanzialmente rivisto rispetto al modello precedente. Esso viene definito in maniera più articolata, con l'introduzione di nuove strutture e funzionalità.
In particolare, ciascun nodo della mesh è visto congiuntamente come componente di memoria primaria e unità di elaborazione. Proprio in virtù di questo duplice comportamento, a seconda delle varie fasi nella gestione delle richieste di accesso alla memoria, il nodo può assumere ruoli diversi e svolgere operazioni che non si limitano alla semplice memorizzazione di informazioni.
Dal punto di vista strutturale, ciascun nodo è dotato al proprio interno di un banco di μ celle di memoria primaria (*primary memory*). Il parametro μ è completamente indipendente dal numero di nodi M e lascia la libertà di impostare o meno una qualsiasi relazione con questa grandezza.
Ogni cella mantiene informazione e può essere destinataria, anche in questo caso, di due diversi tipi di richiesta da parte di un processore: *Write* (W), per scrivere o aggiornare l'informazione al suo interno, o *Read* (R), per acquisire l'informazione in essa contenuta. Banalmente, nel caso limite per $\mu = 1$ si ritorna alla struttura del modello precedente.

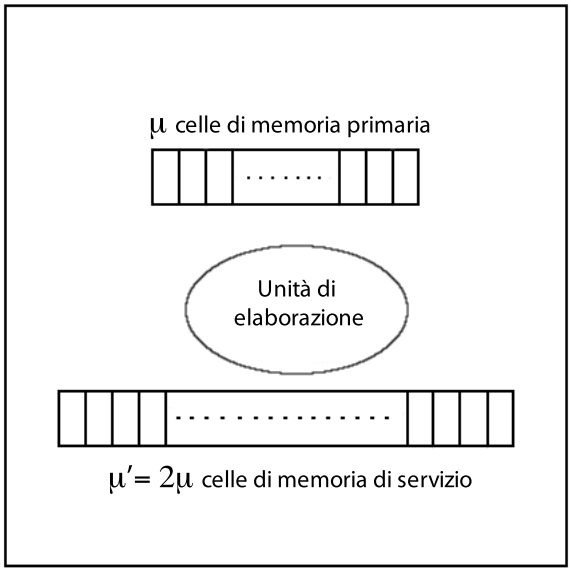


Figura 4.3: Layout del nodo

Questi primi aspetti sono quelli che caratterizzano il nodo come componente di memoria primaria.
All'interno di un nodo risiede inoltre un'unità di elaborazione, che per svolgere le proprie funzioni necessita di una memoria di appoggio, costituita da un banco di $\mu^{'} = 2\mu$ celle. Questa memoria di servizio (*working memory*) è la locazione in cui mantenere temporaneamente i messaggi che giungono al nodo e che eventualmente devono poi essere inoltrati. Essa però è soprattutto la sede in cui memorizzare le informazioni, acquisite sempre attraverso i messaggi, necessarie all'elaboratore per contribuire a soddisfare in maniera complessivamente efficiente nel sistema le varie richieste di accesso alla memoria. La funzionalità risulta particolarmente evidente nella successiva sezione relativa agli algortimi.
Considerando il layout del nodo, in Fig. 4.3, e ipotizzando di disporre di banchi di memoria sequenziali al suo interno, è necessario esprimere in forma parametrica i costi per le comunicazioni tra nodi vicini. Essi dipendono almeno dalla dimensione lineare del banco di memoria, pertanto, se i banchi occupano un layout di area proporzionale a μ, possiamo formularli come $\Omega(\sqrt{\mu})$.
L'unità di elaborazione presente in ciascun nodo è in grado di eseguire operazioni di confronto tra dati, ricevuti e memorizzati, per valutare se e dove un messaggio deve essere inoltrato e per produrre un ordinamento delle richieste stesse sui nodi della mesh. A queste si aggiungono le ovvie funzionalità di ricezione/trasmissione di un messaggio, scrittura delle celle di memoria primaria, acquisizione del loro contenuto e creazione di un messaggio.
Il nodo come elaboratore rimane tuttavia privo della facoltà di eseguire operazioni da *Unità aritmetico-logica* (*ALU*), che rimangono prerogativa dei processori della macchina. Introdurre una tale potenzialità in ciascun nodo porterebbe ad una sovrapposizione di ruoli che renderebbe necessaria una ridefinizione dell'intera architettura di calcolo. L'elaboratore in ciascun nodo è pertanto un processore dedicato, con funzionalità mirate alla sola gestione delle richieste di accesso in memoria.

4.3 Considerazioni strutturali

In questo modello la taglia della memoria non è più identificabile con il numero di nodi della mesh. Essa diventa funzione anche del parametro μ e la capacità complessiva si esprime quindi come $MEM(M, \mu) = M\mu$. Essa, pur nell'ipotesi di taglia costante dei nodi, è potenzialmente molto superiore rispetto al modello precedente, a parità di taglia della mesh. L'aumento è legato al parametro μ, quindi dipende interamente dal *cluster* di celle inserite nel nodo.
Il numero di richieste che entrano in memoria è lo stesso del modello precedente, quindi, raggruppando più celle in cluster, aumenta la probabilità che un nodo sia

raggiunto da un elevato numero di messaggi. Questo rischio è superato attraverso la strategia di risoluzione.
La struttura di questo modello annulla anche la corrispondenza biunivoca tra celle di memoria e nodi.

4.4 Mappatura delle celle di memoria nei nodi

L'introduzione di un *cluster* di celle di memoria primaria in ciascun nodo pone dunque l'esigenza di definire una mappa che attribuisca un indirizzo univoco a ciascuna cella e, contestualmente, sancisca la corrispondenza tra ciascuna locazione di memoria e il proprio nodo di riferimento. Il passaggio è fondamentale per permettere ai processori di indirizzare i request packet alle locazioni di effettivo interesse e agli altri nodi di indirizzare i messaggi per acquisire o aggiornare i valori nelle celle di memoria.
In questo modello la memoria contiene $M\mu$ celle indirizzabili, pertanto, considerando per comodità i due parametri come potenze di 2, sono necessari $\log_2 M\mu$ bit per realizzare l'indirizzamento. Si considerano quindi le celle numerate in row-major da 0 a $M\mu - 1$, come avveniva nel modello precedente per l'indicizzazione dei nodi.
La via più semplice per mappare le celle nei nodi consiste nel considerare gli M *bit* più significativi dell'indirizzo di una cella come indice del nodo in cui è contenuta e i μ bit meno significativi come indice di cella all'interno del nodo.

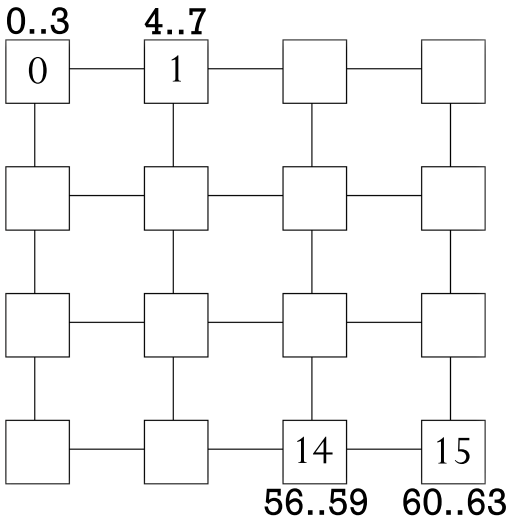


Figura 4.4: Schema di mappatura

In figura 4.4 è illustrato lo schema di mappatura in caso di memoria a 64 locazioni, con $M = 16$ e $\mu = 4$. La numerazione delle celle in row-major è riportata in decimale; si nota che l'indice di un nodo è proprio il corrispondente in decimale degli M bit più significativi della rappresentazione binaria di una cella contenuta in esso.

4.5 Implementazione della strategia

Note le condizioni operative, con la definizione della struttura e delle funzionalità dei componenti, è possibile attuare la strategia fondante del modello. Il processo di implementazione si esplica attraverso la scelta dell'algoritmo più adatto a ciascuna fase prevista dalla strategia.
Gli algoritmi utilizzati sono riportati in dettaglio e analizzati nella sezione 4.7.
La strategia richiede innanzitutto di suddividere le richieste e creare le sequenze da elaborare. Lo strumento più adatto per realizzare questo processo è un ordinamento sui nodi della mesh. La chiave dell'ordinamento è costituita dal nodo di destinazione della richiesta. Affinché il modello soddisfi le richieste secondo la semantica definita, il metodo di ordinamento utilizzato deve essere stabile. Questa proprietà è equivalente ad un ordinamento secondo una chiave più ampia, che prevede come chiave secondaria l'indice del nodo d'ingresso. La relazione d'ordine implicita nella numerazione dei nodi permette di soddisfare le richieste con la stessa chiave primaria, in accordo ai criteri di precedenza discussi in 2.4. I metodi di ordinamento in parallelo che si possono utilizzare sono il sorting bitonico [5] o l'algoritmo di Schnorr e Shamir [7], che, nel modello classico di mesh, risolve il problema in tempo ottimo. Dopo l'esecuzione, le sequenze risultano consecutive sulle righe della mesh in ordine crescente di indirizzo e sono pronte per l'elaborazione.
Il *core* della strategia si realizza invece tramite due algoritmi specifici, progettati appositamente. *Preprocessing* (algoritmo 4.7.1) implementa la primissima fase del procedimento: l'eliminazione delle sovrascritture inutili. Tramite la comunicazione tra nodi adiacenti, il codice provvede ad eliminare le richieste di scrittura intermedie in sequenze di scritture allo stesso indirizzo.
Il soddisfacimento vero e proprio delle singole richieste avviene poi nell'*Algoritmo di propagazione* (algoritmo 4.7.2), che realizza l'effettiva elaborazione all'interno delle singole sequenze sui nodi della mesh, secondo lo schema della catena. L'algoritmo eventualmente acquisisce in fase iniziale il valore dalla cella di memoria in questione e trasferisce tra i nodi adiacenti le informazioni (*bypassando* i nodi svuotati dalla fase precedente), realizzando così l'attivazione consecutiva dei nodi. Riconosciuto un cambio di destinazione tra richieste adiacenti, dunque il termine di una sequenza, l'algoritmo, nelle opportune condizioni, procede all'aggiornamento del valore nella cella primaria di destinazione.
A questo punto, la fase conclusiva dell'implementazione ricalca quanto esposto nel

modello precedente. Le richieste di lettura sono effettivamente soddisfatte con il recapito dei valori acquisiti ai nodi che contengono i request packet corrispondenti. Come nel modello precedente, questo passaggio si esplica con l'esecuzione dell'algoritmo per le permutazioni parziali. Il processo si completa con l'algoritmo che svuota la mesh in parallelo sulle colonne e inserisce le richieste soddisfatte nelle code corrispondenti, da cui i processori prelevano i valori di interesse.

4.6 I messaggi

La strategia e gli algoritmi che la mettono in atto si fondano su un continuo scambio di informazioni tra nodi adiacenti. Queste giungono ai nodi attraverso l'invio continuo di messaggi, di cui è opportuno definire il formato nei vari casi.
I messaggi creati a partire dai request packet contengono le informazioni sufficienti per eseguire l'ordinamento iniziale e il routing conclusivo. Nel mezzo si pongono una serie di necessità informative, che dipendono esclusivamente dai *request packet* iniziali e si manifestano negli algoritmi progettati appositamente per questo contesto.
Questa sezione analizza in dettaglio i messaggi utilizzati proprio da *Preprocessing* e *Propagazione*. La trattazione è ad alto livello e si focalizza sulle informazioni necessarie da scambiare. Per non appesantire eccessivamente l'esposizione, le informazioni relative al nodo mittente e al nodo destinatario sono omesse dal formato del messaggio e considerate implicite.
I primi messaggi in fase di preprocessing sono le interrogazioni da parte dei nodi iniziali delle righe e le relative risposte. Il messaggio richiesta, inviato dal primo nodo di ogni riga al vicino inferiore, è:

$$\langle \textit{istruzione lettura} \rangle$$

La risposta che i destinatari forniscono ai nodi mittenti è invece:

$$\langle \textit{tipo richiesta contenuta} \rangle$$

Essa permette ai nodi iniziali delle righe di far conoscere all'ultimo nodo la richiesta contenuta nel suo successore.
Il messaggio successivo è il principale per l'obiettivo della procedura. Esso si propaga sulle righe a partire dal primo nodo e il suo formato è:

$$\langle \textit{indirizzo richiesta nodo}, \textit{tipo richiesta}, \textit{indirizzo richiesta } s, \textit{tipo richiesta } s \rangle$$

I primi due campi contengono la destinazione e il tipo di richiesta dell'ultimo nodo visitato. Ogni nodo attraversato li aggiorna secondo il proprio contenuto. Gli ultimi due campi sono invece destinati al nodo finale di ciascuna riga. Essi contengono la

destinazione e il tipo di richiesta del nodo iniziale della riga sottostante, considerato il successore dell'ultimo nodo. Esso, ricevuto il messaggio, li memorizza al proprio interno ed esaurisce la propagazione del messaggio. I campi *indirizzo* contengono uno dei possibili indirizzi di memoria mentre i campi *tipo richiesta* hanno solo due possibilità: lettura (R) o scrittura (W).
L'ultima tipologia di messaggio che può essere utilizzata in questa procedura è la notifica di cancellazione. Per cancellare una condizione di sovrascrittura, un nodo invia un messaggio

$$\langle \textit{istruzione cancellazione richiesta} \rangle$$

al proprio predecessore.
Nell'algoritmo di propagazione sono necessari messaggi attraverso cui trasferire informazione lungo i nodi della catena, da un nodo al suo successore. Essi sono in formato:

$$\langle \textit{indirizzo ultima richiesta},\ \textit{valore ultima richiesta},\ \textit{provenienza del dato} \rangle$$

I primi due campi rappresentano l'indirizzo di destinazione e il valore relativi all'ultima richiesta soddisfatta. Il primo campo può contenere un qualsiasi indirizzo della memoria, mentre l'altro può contenere qualsiasi dato che possa essere mantenuto in memoria centrale. Il nodo che riceve confronta l'indirizzo della propria richiesta con l'istanza del primo campo e, se coincidono, acquisisce l'istanza del secondo.
Il terzo campo, invece, indica se l'informazione che si sta propagando in avanti sulla riga proviene da una scrittura (W) attraversata o da un prelevamento dalla cella di memoria, cioè da una lettura (R) soddisfatta. Quest'ultimo campo ha pertanto un dominio di due soli valori possibili: R o W. Esso è necessario per capire quando si deve aggiornare la cella di memoria in questione, inviando al nodo corrispondente il dato.
Le istanze di un messaggio, aggiornato dopo una richiesta di lettura, sono dunque:

$\langle \textit{indirizzo } R,\ \textit{valore } R,\ R \rangle$, con dato prelevato dalla cella,
$\langle \textit{indirizzo } R,\ \textit{valore } R,\ W \rangle$, con dato prelevato da scrittura precedente.

Dopo una richiesta di scrittura, il messaggio è invece:

$$\langle \textit{indirizzo } W,\ \textit{valore } W,\ W \rangle$$

Per prelevare il dato da una cella di memoria non contenuta in esso, e soddisfare quindi una richiesta di lettura che apre una sequenza, un nodo invia un messaggio:

$$\langle \textit{indirizzo cella},\ \textit{istruzione lettura} \rangle$$

al nodo che contiene la cella. Questo gli risponde con un messaggio:

$\langle valore\ contenuto \rangle$

Qualora un nodo riscontri la necessità di aggiornare una cella di memoria primaria non contenuta in esso, invia un messaggio

$\langle indirizzo\ cella,\ valore\ richiesta,\ istruzione\ scrittura \rangle$

al nodo che contiene la cella in questione.
I nodi mittenti risalgono all'indice del nodo che contiene la cella d'interesse tramite la mappa definita in precedenza (sezione 4.4).

4.7 Algoritmi e analisi di complessità

4.7.1 Ordinamento sulla mesh

Rispetto alla funzione di indicizzazione dei nodi, il problema di ordinamento è concepito come spostamento del $j - esimo$ elemento più piccolo nel nodo indicizzato con j, per ogni $j = 0,\ 1,\ ...,\ N - 1$.
In base alla contestualizzazione del problema, la modalità di indicizzazione dei nodi segue lo schema *row-major*.
I metodi principali per ordinare secondo questa indicizzazione sono degli adattamenti dell'ordinamento bitonico di Batcher [1]. In generale essi raggiungono prestazioni inferiori rispetto ad uno schema *shuffle-major*. Una versione nota è quella di Orcutt [6], che risolve il problema in tempo $O(\sqrt{M} \log \sqrt{M})$.
Un algoritmo che si presta a risolvere meglio il problema dell'ordinamento in questa forma è quello di Nassimi e Sahni [5]. Attraverso un diverso adattamento del *bitonic sort*, essi ottengono un algoritmo che si avvicina molto alle prestazioni dell'ordinamento shuffle-major di Thompson e Kung [10]. L'ordine di complessità è infatti il medesimo, con il problema risolto in un una quantità $O(\sqrt{M})$ di passi di computazione. Questo avviene tuttavia a scapito di un maggior numero di passi di instradamento e di scambi a livello di registri del processore.
Sempre in questi termini, il problema è risolvibile anche in una modalità alternativa, con la possibilità di sfruttare algoritmi ottimi nel modello classico di mesh, come quello di Schnorr e Shamir [7], che ordina gli elementi della mesh in tempo ottimo secondo lo schema *row-major snake-like*. Si rende tuttavia necessario spendere poi del tempo aggiuntivo per portare gli elementi in *row-major*, in modo che siano utilizzabili dagli algoritmi successivi.
Le differenze di tempo sono però effettivamente valutabili solo a parità di modello della mesh e quindi di clock, secondo quanto esposto sulle condizioni di confrontabilità degli algoritmi in 2.8.

4.7.2 Preprocessing delle richieste

La procedura di *preprocessing* rappresenta una fase chiave per evitare gli accessi inutili alle celle di memoria, secondo la descrizione precedente. Nell'attuazione della strategia risolutiva esposta, il compito specifico di questo algoritmo è l'eliminazione delle scritture intermedie consecutive indirizzate ad una stessa cella, mantenendo solo la richiesta finale.
Un esempio in estrema sintesi è visualizzato nelle figure 4.5 e 4.6. La raffigurazione è molto essenziale e mira a far comprendere gli effetti della procedura, a prescindere dai formalismi precisi dei messaggi nei nodi. La figura 4.5 rappresenta una mesh di taglia $N = 16$ prima dell'esecuzione dell'algoritmo; essa contiene una sequenza di richieste di accesso alle celle di memoria, ordinate sui nodi dall'algoritmo precedente. Nella figura 4.6 si vede come, dopo l'esecuzione dell'algoritmo, la mesh risulti priva delle richieste di scrittura adiacenti indirizzate allo stesso nodo. La condizione di adiacenza per i nodi terminali di una riga va confrontata sui nodi iniziali della riga successiva, considerati i loro successori.
Le operazioni nell'algoritmo, svolte in parallelo sfruttando la topologia a mesh della memoria, sono:

1. Su tutte le righe della mesh, i nodi iniziali interrogano il vicino inferiore (se esiste), per acquisire tipo e indirizzo della richiesta in esso contenuta. Questa

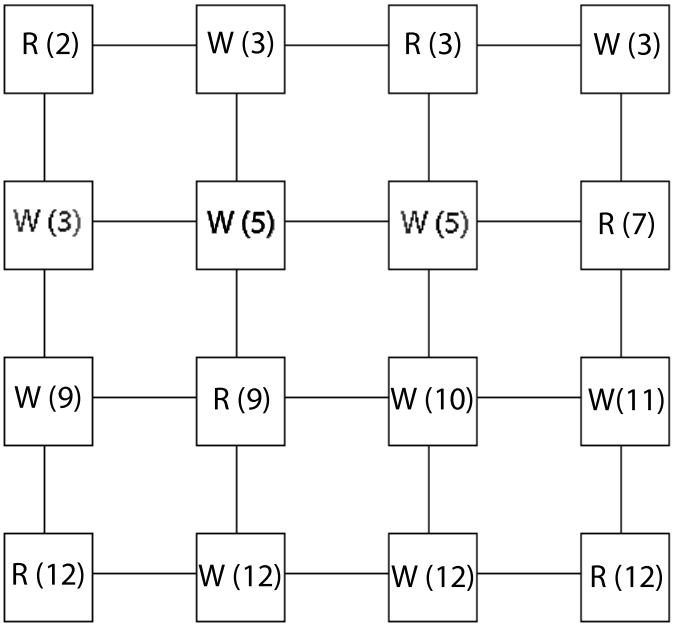


Figura 4.5: Mesh iniziale

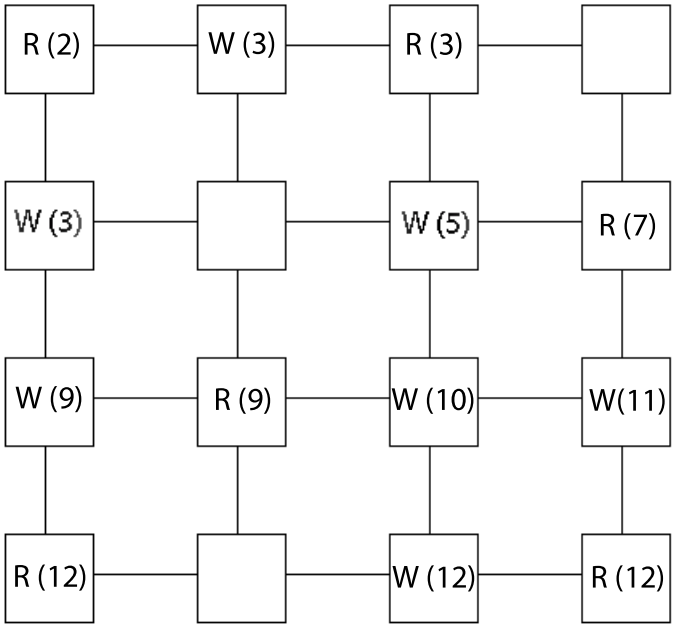


Figura 4.6: Mesh dopo il preprocessing

informazione è destinata al nodo finale della riga: esso si svuota se la propria richiesta e quella del nodo interrogato, considerato il suo successore, sono scritture ad uno stesso indirizzo. Ogni nodo iniziale crea poi un messaggio con l'indirizzo e il tipo della propria richiesta e, se acquisite, le informazioni ottenute dal vicino inferiore. I messaggi sono poi inoltrati ai vicini destri sulle righe.

2. Iterativamente per tutte le colonne della mesh, tutti i nodi su una colonna, in parallelo, inviano al predecessore una notifica di cancellazione se riscontrano una coincidenza tra il tipo e l'indirizzo della propria richiesta e quella portata dal messaggio. In questo caso il nodo predecessore rimuove la propria richiesta e si svuota.
   Ciascun nodo, poi, aggiorna il messaggio con il tipo e l'indirizzo della propria richiesta e lo inoltra in avanti, al vicino sulla riga.

3. Su tutte le righe della mesh, i nodi finali, ricevuto il messaggio, confrontano le informazioni trasportate con quelle della richiesta che contengono. Se il nodo finale e il suo predecessore contengono entrambi una richiesta di scrittura indirizzata alla stessa cella, il nodo finale invia una notifica al predecessore, affinché cancelli la propria.
   L'unità di elaborazione del nodo finale scrive poi nella propria memoria di

servizio le informazioni che il nodo iniziale ha acquisito appositamente dal vicino inferiore e trasmesso nel messaggio. Questa conoscenza relativa al suo successore sarà utile al nodo finale nella fase di propagazione in avanti, come richiamato in seguito.
In termini più immediati però, queste informazioni sono utilizzate dall'ultimo nodo per sapere se cancellare o meno la propria richiesta. Questo avviene nel caso in cui l'ultimo nodo e il primo della riga successiva contengano entrambi una richiesta di scrittura allo stesso indirizzo di memoria.

Lo pseudocodice completo della procedura è riportato nell'algoritmo 4.7.1

Algorithm 4.7.1 Procedura di *Preprocessing*

Require: *richieste di accesso alla memoria ordinate per indirizzo, in row − major sulle righe della mesh*
Ensure: *eliminazione sequenze di richieste W consecutive allo stesso indirizzo*

for all *nodo sulla prima colonna della mesh* **do**
if ($\exists$ *vicino inferiore*) **then**
acquisisci l'indirizzo e il tipo di richiesta del vicino inferiore
inoltra al vicino sulla riga il messaggio:
$\langle$*indirizzo richiesta nodo*, *tipo richiesta*, *indirizzo richiesta s*, *tipo richiesta s*$\rangle$
else
inoltra al vicino in avanti sulla riga il messaggio:
$\langle$*indirizzo richiesta nodo*, *tipo richiesta nodo*$\rangle$
end if
end for

$i = 1$ {→contatore di colonna, inizializzato alla seconda}

while ($i < \sqrt{M} - 1$) **do**
for all *nodi sulla colonna i* **do**
if ((*indirizzo richiesta nodo* = *indirizzo richiesta messaggio*)
$\wedge$ (*tipo richiesta messaggio* = W) $\wedge$ (*tipo richiesta nodo* = W)) **then**
invia al predecessore una notifica che ne cancella la richiesta W
{→ svuota il predecessore}
end if
aggiorna messaggio con la richiesta nel nodo
il nodo inoltra il messaggio aggiornato in avanti sulla riga
end for
$i = i + 1$
end while

for all *nodo sull'ultima colonna della mesh* **do**
if ((*indirizzo richiesta nodo* = *indirizzo richiesta messaggio*) $\wedge$
(*tipo richiesta messaggio* = W) $\wedge$ (*tipo richiesta nodo* = W)) **then**
invia al predecessore una notifica che ne cancella la richiesta W
{→ svuota il predecessore}
end if
scrivi nella memoria di servizio le informazioni *indirizzo richiesta successore*
e *tipo richiesta successore* contenute nel messaggio ricevuto
end for

if ((*indirizzo richiesta nodo* = *indirizzo richiesta s*) ∧
(*tipo richiesta s* = *W*) ∧ (*tipo richiesta nodo* = *W*)) **then**
  cancella la richiesta W nel nodo
  {→ svuota ultimo nodo}
end if

La prima fase richiede un numero costante di passi per l'acquisizione delle informazioni sul vicino inferiore da parte dei nodi sulla prima colonna (ad eccezione dell'ultimo). Ciascun nodo poi esegue una quantità costante di operazioni per preparare il messaggio che inoltra nel passo successivo.
Nella seconda fase, ciascun nodo su una colonna esegue ancora una quantità costante di operazioni di elaborazione e poi inoltra il messaggio al vicino sulla riga. La sequenza si ripete per $\sqrt{M} - 2$ colonne.
In fase finale, tutti i nodi sull'ultima colonna eseguono un numero superiore di operazioni di elaborazione rispetto ai nodi sulle colonne precedenti, tuttavia l'ordine di queste operazioni rimane costante. Anch'essi inviano nel caso peggiore la notifica al proprio predecessore.
Complessivamente questa procedura richiede $O(\sqrt{M})$ passi per le operazioni di elaborazione e $O(\sqrt{M})$ passi per il transito dei messaggi. In virtù della formulazione parametrica dei costi di comunicazione tra nodi vicini, la complessità della procedura di preprocessing è dominata dagli $O(\sqrt{M})$ passi per il transito dei messaggi.

4.7.3 Algoritmo di propagazione

L'algoritmo di propagazione attua la fase finale e decisiva nella strategia per il soddisfacimento di tutte le richieste di accesso alla memoria.
L'idea di questo algoritmo è propagare sulle righe ed eventualmente sulle colonne della mesh le informazioni utili a soddisfare le richieste indirizzate alla stessa locazione di memoria, in modo da ridurre al minimo gli accessi alle celle di memoria primaria.
La strategia alla base dell'algoritmo prevede di sfruttare il più possibile la topologia a mesh della memoria, che permette la concorrenza nelle operazioni, sulle righe e colonne, e negli accessi.
L'algoritmo interviene in seguito alla fase di *preprocessing*, durante le quale, secondo la strategia descritta in precedenza, possono essere rimosse alcune richieste di scrittura. Pertanto, esso opera sulla mesh che potenzialmente può contenere dei nodi vuoti, all'interno della sequenza ordinata di richieste di accesso alla memoria.
I passi dell'algoritmo sono:

1. Ciascun nodo contenente una richiesta di lettura interroga il nodo predecessore in merito all'indirizzo della richiesta in esso contenuta. I nodi iniziali sulle

righe della mesh, ad ovvia eccezione del primo, considerano come vicino l'ultimo nodo della riga superiore e procedono ad interrogarlo.
Una volta ricevuta la risposta, ogni nodo verifica se l'indirizzo ricevuto coincide con il proprio. I nodi in cui gli indirizzi risultano diversi richiedono l'informazione in caso di lettura ai nodi che contengono la cella indirizzata. Questa è l'unico momento in cui può avvenire una lettura vera e propria di una locazione di memoria.
Le operazioni si svolgono in maniera concorrente mediante scambio di messaggi.
Una volta acquisito il valore, l'unità di elaborazione del nodo lo scrive nella memoria di servizio e le richieste di lettura sono soddisfatte. Esse possono eventualmente soddisfare a loro volta analoghe richieste successive indirizzate alla stessa cella.

2. In ogni riga, se presente, il primo nodo contenente una richiesta di scrittura o di lettura soddisfatta crea il messaggio per la propagazione, con contenuto diverso a seconda della richiesta iniziale.

3. A partire dai nodi individuati al passo precedente, i messaggi vengono propagati in avanti sulle righe in maniera concorrente. Ogni nodo che riceve confronta l'indirizzo della propria richiesta con quello contenuto nel messaggio. Se coincidono e il nodo contiene una richiesta di lettura, l'unità di elaborazione scrive il valore del messaggio nella memoria di servizio del nodo e la richiesta è soddisfatta; altrimenti, se il nodo contiene una richiesta di scrittura allo stesso indirizzo, l'elaboratore aggiorna il valore contenuto nel messaggio con quello della scrittura. Nel caso in cui nel nodo sia presente una richiesta indirizzata ad una cella diversa, l'unità di elaborazione ristruttura completamente il messaggio, sia che contenga una richiesta di lettura, che risulta sicuramente soddisfatta, che una di scrittura.
In quest'ultima condizione di indirizzi differenti, se il messaggio ricevuto contiene un valore proveniente da una precedente richiesta di scrittura, l'elaboratore memorizza nel nodo i valori per aggiornare successivamente la cella riferita dal messaggio. Questa istruzione rappresenta concettualmente l'ultimo passaggio della strategia illustrata in precedenza. Ciascun nodo inoltra poi il messaggio eventualmente aggiornato in avanti sulla riga, al vicino destro.
La fase si ripete ciclicamente finchè la propagazione raggiunge i nodi finali delle righe.

4. L'ultimo nodo di ogni riga, una volta raggiunto dal messaggio, esegue istruzioni diverse a seconda dell'indirizzo della richiesta contenuta nel suo successore, il primo nodo della riga successiva. I nodi in questione sono già a conoscenza

di queste informazioni grazie ai messaggi ricevuti durante la fase di preprocessing.
Se l'ultimo nodo è vuoto, significa che il successore, se non vuoto a sua volta, contiene una richiesta di scrittura. Se questa è indirizzata ad una cella diversa da quella riferita dal messaggio e quest'ultimo contiene un valore proveniente da una precedente richiesta di scrittura, l'elaboratore dell'ultimo nodo memorizza nel nodo stesso i dati necessari ad aggiornare la cella indirizzata dal messaggio, che termina così il proprio avanzamento. La memorizzazione dei dati per l'aggiornamento avviene analogamente se l'ultimo nodo è raggiunto da un messaggio contenente indirizzo diverso e valore proveniente da una precedente richiesta di scrittura. Se si verifica però solo la differenza tra indirizzi, il comportamento dell'ultimo nodo dipende dalla richiesta del successore: se è una lettura allo stesso indirizzo della richiesta dell'ultimo nodo (quindi non soddisfatta), questo invia un opportuno messaggio al nodo iniziale della riga stessa.
Se invece il successore non contiene questa richiesta e l'ultimo nodo contiene una richiesta di scrittura ad una cella diversa, l'unità di elaborazione memorizza nel nodo i valori per il successivo aggiornamento della cella.
Nel caso in cui l'ultimo nodo contenga una richiesta di lettura al medesimo indirizzo di quello del messaggio ricevuto, l'unità di elaborazione scrive nella memoria di servizio il valore del messaggio e la richiesta è soddisfatta. Se il successore poi contiene a sua volta una richiesta di lettura allo stesso indirizzo (quindi non soddisfatta) l'ultimo nodo gli inoltra il messaggio nella modalità descritta nella situazione precedente. Nel caso in cui la richiesta dell'ultimo nodo sia invece una scrittura all'indirizzo contenuto nel messaggio, quest'ultimo viene aggiornato e propagato solo se il successore contiene anch'esso una richiesta di lettura allo stesso indirizzo; altrimenti l'unità di elaborazione dell'ultimo nodo mantiene all'interno del nodo stesso il valore della scrittura, per aggiornare successivamente l'indirizzo in questione.
Le istruzioni vengono eseguite in maniera concorrente su tutte le righe.

5. I nodi iniziali delle righe che hanno ricevuto il messaggio dai nodi terminali della riga stessa lo inoltrano in avanti sulla colonna (cioè al vicino inferiore), in modo che raggiunga i successori considerati al punto precedente. Ricevuto il messaggio, questi eseguono una duplice azione: avviano la propagazione sulla riga per soddisfare tutte le richieste di lettura all'indirizzo del messaggio (ancora inevase) e propagano a loro volta il messaggio in avanti sulla colonna, per soddisfare eventuali altre richieste al medesimo indirizzo, sulla riga sottostante.
   Se il messaggio trasmette un valore proveniente da una scrittura, nel nodo

in cui si arresta la propagazione su una riga (dove si riscontra cioè la prima assenza di corrispondenza tra indirizzo della richiesta nel nodo e indirizzo nel messaggio), vengono mantenuti i dati per l'aggiornamento della cella di memoria indirizzata dal messaggio.

6. I nodi in cui sono memorizzati i valori per l'aggiornamento della memoria primaria provvedono a trasmetterli alle celle in questione. Essi inviano, in maniera concorrente, le opportune notifiche di scrittura. Questa è l'unica fase in cui possono essere effettivamente scritte le locazioni di memoria. Al termine dell'aggiornamento questi valori sono rimossi dai nodi in cui erano mantenuti.

L'intero algoritmo è riportato in pseudocodice in 4.7.2

Algorithm 4.7.2 Algoritmo di propagazione in avanti sulle righe

Require: *richieste di accesso alla memoria ordinate per indirizzo, in row − major sulle righe della mesh, sovrscritture eliminate*
Ensure: *richieste di accesso alla memoria soddisfatte*
valori nelle celle di memoria aggiornati

Esecuzione in parallelo:

for all *nodi* **do**
  if *nodo contiene richiesta R* **then**
    interroga il nodo predecessore
    if (*indirizzo richiesta predecessore* $\neq$ *indirizzo R del nodo*) **then**
      Acquisisci il valore dal nodo contenente la cella
      Memorizza il valore nel nodo
    end if
  end if
end for

for all *primo nodo non vuoto su una riga della mesh* **do**
  if *nodo contiene una richiesta R* **then**
    if (*richiesta R soddisfatta*) **then**
      crea il messaggio:$\langle$*indirizzo di R, valore di R, R*$\rangle$
    else
      Avanza al primo nodo contenente R soddisfatta o W
      if *nodo raggiunto contiene R soddisfatta* **then**
        crea il messaggio:$\langle$*indirizzo di R, valore di R, R*$\rangle$
      else
        crea il messaggio:$\langle$*indirizzo di W, valore di W, W*$\rangle$
      end if
    else
      crea il messaggio:$\langle$*indirizzo di W, valore di W, W*$\rangle$
    end if
  end if
end for

while *not*(*fine riga*) **do**
  A partire dai nodi individuati inoltra in avanti sulle righe i messaggi
  for all nodi che ricevono il messaggio **do**
    if *nodo vuoto* **then**
      inoltra il messaggio in avanti al vicino sulla riga

```
      else
        if (indirizzo messaggio ≠ indirizzo richiesta nel nodo) then
          if (tipo operazione messaggio = W) then
            mantieni nel nodo le informazioni per l'aggiornamento
            {→aggiornamento memoria primaria in attesa}
          end if
          if nodo contiene richiesta R then
            aggiorna il messaggio in:⟨indirizzo di R, valore acquisito, R⟩
          else
            aggiorna il messaggio in:⟨indirizzo di W, valore di W, W⟩
          end if
        else
          if nodo contiene richiesta R then
            memorizza nel nodo il valore contenuto nel messaggio
            {→ richiesta soddisfatta}
          else
            aggiorna il messaggio in:⟨indirizzo di W, valore di W, W⟩
          end if
        end if
      end if
      inoltra il messaggio in avanti al vicino sulla riga
    end for
  end while

  for all nodo sull'ultima colonna della mesh do
    if (nodo vuoto) then
      if (indirizzo W successore ≠ indirizzo messaggio) ∧
      (tipo operazione messaggio = W) then
        mantieni nel nodo le informazioni per l'aggiornamento
        {→aggiornamento memoria primaria in attesa}
      end if
    else
      if (indirizzo messaggio ricevuto ≠ indirizzo richiesta nel nodo) then
        if (tipo operazione messaggio = W) then
          mantieni nel nodo le informazioni per l'aggiornamento
          {→aggiornamento memoria primaria in attesa}
        end if
        if (successore richiede R allo stesso indirizzo dell'ultimo nodo
        {→ non soddisfatta}) then
          if nodo contiene richiesta R {→ soddisfatta} then
            invia al nodo iniziale della riga il messaggio:⟨indirizzo R, valore R, R⟩
```

```
          else
            invia al nodo iniziale della riga il messaggio:⟨indirizzo W, valore W, W⟩
          end if
        else
          if nodo contiene richiesta W then
            mantieni nel nodo le informazioni per l'aggiornamento
            {→aggiornamento memoria primaria in attesa}
          end if
        end if
      else
        if (successore richiede R allo stesso indirizzo dell'ultimo nodo
        {→ non soddisfatta}) then
          if nodo contiene richiesta R then
            memorizza nel nodo il valore contenuto nel messaggio
            {→ richiesta soddisfatta}
            invia al nodo iniziale della riga il messaggio ricevuto
          else
            invia al nodo iniziale della riga il messaggio:⟨indirizzo W, valore W, W⟩
          end if
        else
          if nodo contiene richiesta R then
            memorizza nel nodo il valore contenuto nel messaggio
            {→ richiesta soddisfatta}
          else
            mantieni nel nodo le informazioni per l'aggiornamento
            {→aggiornamento memoria primaria in attesa}
          end if
        end if
      end if
    end if
  end for

  for all nodo sulla prima colonna della mesh do
    if (nodo contiene R non soddisfatta) ∧ (riceve messaggio dal vicino superiore)
    then
      while (∃ sulla colonna richieste R all'indirizzo nel messaggio) do
        propaga il messaggio in basso sulla colonna
      end while
    end if
    if nodo ha ricevuto il messaggio then
      memorizza il valore nel nodo
```

```
      {→ richiesta soddisfatta}
      while (∃ sulla riga richieste R all'indirizzo nel messaggio) do
        propaga il messaggio in avanti sulla riga
      end while
      if (∃ successore non vuoto all'ultimo nodo di propagazione) then
        if (indirizzo richiesta successore ≠ indirizzo messaggio) ∧
        (tipo operazione messaggio = W) then
          mantieni nel nodo le informazioni per l'aggiornamento
          {→aggiornamento memoria primaria in attesa}
        else
          if (tipo operazione messaggio = W) then
            mantieni nel nodo le informazioni per l'aggiornamento
            {→aggiornamento memoria primaria in attesa}
          end if
        end if
      end if
    end if
  end for

  for all nodi con aggiornamenti della memoria primaria in attesa do
    trasmetti i valori memorizzati alle celle indirizzate per la scrittura
    x {→aggiornamenti memoria primaria completati}
    cancella i valori dai nodi
  end for
```

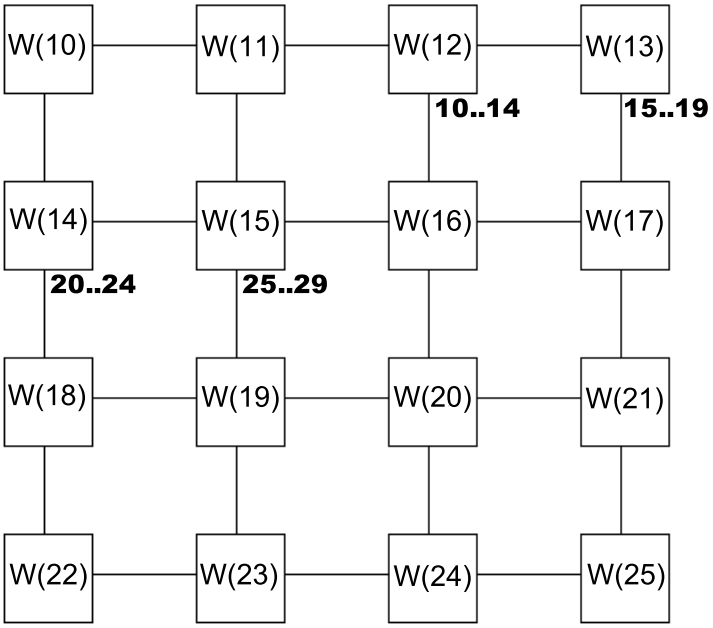


Figura 4.7: Caso sfavorevole per la propagazione

Si analizza a questo punto la complessità computazionale del procedimento.
La prima fase dell'algoritmo impiega innanzitutto $2\sqrt{M}$ passi per l'interrogazione dei predecessori da parte dei nodi sulla prima colonna. In base alle risposte, di seguito, può essere necessario prelevare i valori in memoria, direttamente dalle celle in cui sono memorizzati. Nel caso in cui ci siano più richieste di lettura a celle contenute nello stesso nodo, questo si trasforma in un problema di instradamento: un *routing* $1 - k$ simile a quanto visto nel modello precedente.
In questo caso tuttavia, in virtù della diversa strategia e della nuova configurazione della macchina, $k = \mu$. L'algoritmo di propagazione si avvale pertanto dell'algoritmo SHORTQROUTE di Sibeyn e Kaufmann [8], discusso in 3.5.1, che risolve così il problema in $O(\sqrt{\mu}\sqrt{M})$. Successivamente i nodi eseguono una quantità di ordine costante di operazioni per la memorizzazione dei valori al proprio interno. In questa fase la complessità è dominata dunque dalle operazioni di routing.
Nel caso peggiore, la seconda fase richiede $O(\sqrt{M})$ operazioni per individuare il primo nodo che può propagare informazione, seguite da un numero costante di istruzioni di elaborazione per creare il messaggio da inoltrare.
Durante la terza fase, di propagazione vera e propria, viene eseguita una quantità costante di operazioni per memorizzare il valore all'interno del nodo, intervenire eventualmente sul messaggio e infine inoltrarlo. Nel caso peggiore è necessario inoltre mantenere all'interno del nodo i dati provenienti da una scrittura finale, in attesa

del successivo aggiornamento della memoria primaria. Ciò richiede una quantità aggiuntiva di operazioni che è comunque costante.
Il ciclo si ripete per $O(\sqrt{M})$ iterazioni quindi, nel complesso, la terza fase richiede $O(\sqrt{M})$ passi per le operazioni di elaborazione e $O(\sqrt{M})$ passi per le comunicazioni. La quarta e la quinta fase coinvolgono una serie di operazioni che richiedono $O(\sqrt{M})$ passi.
Nel caso peggiore, durante l'ultima fase, si deve procedere all'aggiornamento delle celle di memoria con il valore proveniente dall'ultima scrittura di cui sono destinatarie. Questi valori sono temporaneamente memorizzati in vari nodi, che inviano le notifiche di aggiornamento. Se risultano coinvolte in maniera concorrente più celle in uno stesso nodo, riemerge nuovamente il problema del *routing* $1 - k$.
Le Fig. 4.7 e 4.8 rappresentano delle configurazioni limite per evidenziare le eventualità discusse.
In Fig. 4.7 è illustrato uno dei casi peggiori. Per illustrare la criticità della situazione con nodi raggiunti da molte richieste, senza eccedere nel numero di celle nel nodo, si considera eccezionalmente un valore di μ che non sia potenza di 2. La mesh, con parametri $M = 16$ e $\mu = 5$, contiene richieste di scrittura tutte destinate a celle diverse. In fase di aggiornamento, un nodo è destinatario di più messaggi. I nodi coinvolti sono contrassegnati dall'indicazione delle celle contenute in essi. Il problema del routing si manifesta due volte, all'inizio e alla fine dell'elaborazione.

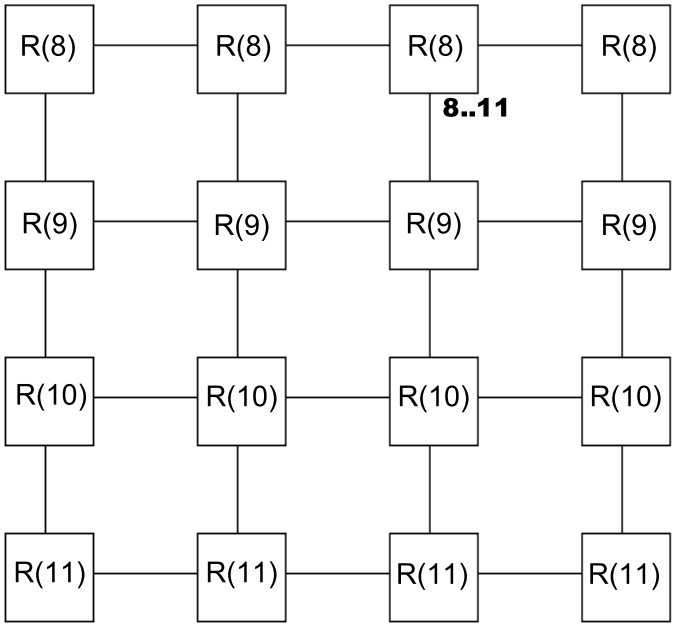


Figura 4.8: Caso favorevole per la propagazione

La Fig. 4.8 rappresenta invece una delle condizioni più favorevoli. In questo caso la mesh, con parametri $M = 16$ e $\mu = 4$, contiene solo richieste di lettura, con lo stesso destinatario sulle righe. Il problema di routing emerge solo alla prima iterazione, con l'acquisizione dei valori delle celle situate tutte nello stesso nodo.
In entrambi i casi si riesce comunque a racchiudere la complessità entro $O(\sqrt{\mu}\sqrt{M})$ passi.
In generale quindi, la complessità è funzione di entrambi i parametri di costruzione del modulo di memoria.
Considerando la successione tra preprocessing e propagazione sulle righe durante una risoluzione, l'ultima domina in termini di passi di computazione.

4.7.4 Permutazioni sulla mesh

Come anticipato, il sottoproblema è il *routing* $1 - 1$ già analizzato nel modello precedente. Il metodo per trattarlo è pertanto lo stesso, con un'alternativa da valutare tra gli algoritmi Leighton, Makedon e Tollis [4] e Sibeyn, Chlebus e Kauffman [9], che risolvono il problema in $2\sqrt{M} - 2$ passi e forniscono prestazioni ottime nel caso peggiore.

4.8 Considerazioni finali

Le prestazioni del modello precedente lasciavano ampi spazi di miglioramento, prontamente catturati da questa evoluzione nel modello.
L'aumento dell'efficienza è legato all'introduzione di una strategia che sfrutta di più e meglio la struttura e le funzionalità a disposizione. Le differenze e i vantaggi rispetto al modello precedente sono molteplici.
In questo modello tutti i nodi contribuiscono allo stesso modo alla soluzione del problema. Il carico computazionale è cioè distribuito in maniera sostanzialmente equa tra tutte le unità di elaborazione a disposizione, a prescindere dalle destinazioni delle richieste. L'intuizione della propagazione delle informazioni tra nodi contigui permette di virtualizzare il passaggio al nodo di destinazione per la maggior parte delle richieste da servire. Il nodo destinatario è coinvolto solo quando strettamente necessario e la maggior parte del lavoro per soddisfare le richieste è svolto al di fuori di esso, sfruttando le comunicazioni tra nodi vicini. In questo modo si risolve alla radice il problema della congestione, superando così la necessità di stratagemmi di spianamento.
Il nuovo parametro di costruzione μ introduce un grado di libertà in più, rispetto alla sola taglia della mesh, per intervenire sulla capacità della memoria. Esso permette inoltre di parametrizzare la banda.
Nello stesso tempo, però, è più probabile che il nodo sia destinatario di più messaggi

di richiesta. Poiché il modo di procedere in questo modello è diverso dal precedente e la maggior parte dello sforzo computazionale avviene all'esterno del nodo, il problema del *routing* $1 - k$ è stavolta sotto controllo. L'ordine M per il problema è esorcizzato: il parametro k nel caso peggiore è $O(\mu)$.
La strategia di lavorare al di fuori del nodo richiede un volume di messaggi nettamente superiore. Il flusso dei messaggi in transito sulla rete è però razionalizzato e ben stabilito rispetto al modello precedente; a parità di condizioni, i costi di comunicazione risultano quindi inferiori.
Considerando la complessità dell'intero procedimento risolutivo, essa è dominata dal routing durante la fase di propagazione sulle righe. Tutti gli altri passaggi hanno complessità di ordine $O(\sqrt{M})$.
Le prestazioni del modello sono ottime per le M richieste servite in maniera concorrente. La funzione di accesso $a(x)$ è $O(\sqrt{\mu}\sqrt{M})$ e, se si considera μ come un fattore costante, per mantenere valida l'ipotesi di taglia costante del nodo, essa diventa $O(\sqrt{M})$.
La funzione di accesso è tuttavia la medesima per qualsiasi locazione x della memoria, com'è ovvio essendo questa memoria non gerarchica. Nell'ottica di combinare elementi che sfruttino la località, un ulteriore avanzamento potrebbe essere quello di introdurre una gerarchia di livelli che, utilizzando meglio ciò che è consentito dai limiti fisici, indurrebbe ulteriormente alla realizzazione fisica.

Conclusioni

Man mano che l'evoluzione tecnologica conduce a macchine sempre più vincolate alla velocità dei segnali, il modello RAM diventa progressivamente inadeguato, anche nel contesto della computazione seriale. Le conseguenze sono duplici. Da un lato, le architetture diventano sempre più complesse, per sfruttare concorrenza e località, nell'obiettivo di approssimare ad un valore costante il tempo di esecuzione di un'istruzione della RAM ideale. Dall'altra parte, la progettazione degli algoritmi deve essere basata su modelli di calcolo che riflettano le nuove architetture, per esporre concorrenza e località. In linea di principio, macchine e algoritmi dovrebbero essere sviluppati congiuntamente, per raggiungere prestazioni ottime sotto i vincoli di riferimento.
Questo lavoro ha svolto un'indagine avanzata su una macchina di questo tipo, nell'ottica di potenziare le prestazioni in maniera mirata. Si è giunti alla definizione di una nuova architettura di memoria, pipelined e non gerarchica, che, considerando il potenziamento della banda di comunicazione tra memoria e processore, sfrutta la concorrenza e trae vantaggio dal parallelismo, nel rispetto della logica sequenziale.
Il progetto finale esibisce una latenza ottima $\theta(\sqrt{\mu}\sqrt{M})$ e una banda $\theta(\frac{\sqrt{M}}{\sqrt{\mu}})$. Esso è scalabile per macchine di taglia arbitraria, sotto i vincoli fisici fondamentali sulla taglia minima del dispositivo e sulla massima velocità del segnale. Questo è riconosciuto mostrando che il progetto si colloca nel piano in modo tale che il numero di porte logiche e la distanza geometrica che deve essere attraversata da ciascun segnale durante un passo logico di computazione siano indipendenti dalla taglia complessiva della macchina. Quindi il tempo fisico è effettivamente proporzionale al numero di passi logici.
A conclusione di un lavoro di ricerca come questo, secondo quanto prospettato anche nell'articolo [2], sorge spontaneo chiedersi a che punto si è giunti nel percorso di miglioramento delle prestazioni della macchina.
L'esplorazione è focalizzata sulla memoria, quindi, per valutare nel complesso l'incremento delle performance, deve essere analizzata anche la struttura dei processori: potrebbe manifestarsi la necessità di ridefinirla, per sfruttare appieno le potenzialità messe a disposizione dal nuovo modulo di memoria.
Una volta armonizzata l'intera architettura, sarebbe interessante riprendere gli stessi

test svolti nell'articolo, per quantificare l'eventuale incremento delle prestazioni in termini sia di parallelismo implicito degli algoritmi seriali che di parallelismo esplicito degli algoritmi paralleli.
Considerando poi in prospettiva il progetto presentato, quali sono gli ulteriori miglioramenti per una struttura di memoria nel piano?
L'avanzamento successivo è ovviamente rappresentato dallo sfruttamento della gerarchia congiuntamente alla concorrenza. L'orizzonte è quello di arricchire la struttura progettata con un'organizzazione pipelined gerarchica, per raggiungere nel piano la latenza ottima $a(x) = O(\sqrt{x})$ per l'indirizzo di memoria x.
Allargando le visione, un'estensione possibile è rappresentata dalla prosecuzione dell'indagine negli spazi anziché nel semplice piano. Un'idea interessante è senz'altro la distribuzione di uno stesso numero di locazioni di memoria su più dimensioni, con il confronto finale sulle prestazioni che si ottengono con le diverse strutture.
Un ulteriore versante da esplorare è quello dell'integrazione memoria-processore nella macchina, nell'ottica di aumentare sempre più le funzionalità dell'elaboratore all'interno del nodo, fino a svolgere le operazioni riservate all'ALU. La sfida del futuro è costituita dalla realizzazione di una struttura memoria-processore unificata all'interno dell'architettura di calcolo.

Bibliografia

[1] K.E. Batcher. "Sorting networks and their applications". In: *Proc. AFIPS 1968 SJCC*. A cura di AFIPS Press. Vol. 32. 1968, pp. 307–314.

[2] G. Bilardi, K. Ekanadham e P. Pattnaik. "On Approximating the Ideal Random Access Machine by Physical Machines". In: *J. ACM* 56 (2009).

[3] M. Kunde. "Routing and Sorting on Mesh-Connected Arrays". In: *VLSI Algorithms and Architectures: Proc. AWOC '88*. A cura di Springer-Verlag. Vol. Lecture Notes in Computer Science, 319. 1988, pp. 432–433.

[4] T. Leighton, F. Makedon e I. Tollis. "A $2n - 2$ Step Algorithm for Routing in an $n \times n$ Array With Constant Size Queue". In: *Proc. Symposium on Parallel Algorithms and Architectures*. A cura di ACM. 1989.

[5] D. Nassimi e S. Sahni. "Bitonic Sort on a Mesh-Connected Parallel Computer". In: *IEEE Trans. Computers* C-27 (1979), pp. 2–7.

[6] S.E. Orcutt. "Computer organization and algorithms for very-high speed computations". PhD. dissertation. Stanford Univ., 1974.

[7] C.P. Schnorr e A. Shamir. "An Optimal Sorting Algorithm For Mesh Connected Computers". In: *Proc. of the 18th ACM Symp. on Theory of Computing*. A cura di ACM. 1986, pp. 255–263.

[8] J.F. Sibeyn e M. Kaufmann. "Deterministic $1 - k$ Routing on Meshes, With Applications to Worm-Hole Routing". In: *Proc. of the 11th Symp. on Theoretical Aspects of Computer Science*. A cura di LNCS-Springer. 1994, pp. 237–248.

[9] J.F. Sibeyn, M. Kaufmann e B.S. Chlebus. "Deterministic Permutations Routing on Meshes". In: *Proc. 5th Symp. on Parallel and Distributed Proc.* A cura di IEEE. 1993.

[10] C.D. Thompson e H.T. Kung. "Sorting on a Mesh-Connected Parallel Computer". In: *CACM* 20 (1977), pp. 263–270.