

Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Fisica

Anno Accademico 2000 – 2001

Tesi di Laurea

ApeNEXT: architettura ed algoritmi di LGT

Candidato

Toni Giorgino

Relatore

Chiar.mo Prof.
Raffaele Tripiccione

Indice

1	Introduzione	3
2	Scenario e motivazioni	5
2.1	Le teorie di campo	5
2.2	Il caso non abeliano	7
2.3	Simulazioni numeriche su grande scala	7
2.4	L'importance sampling	8
2.5	Il calcolo del propagatore	9
2.6	Il quenching	10
2.7	Verso la QCD completa	11
2.8	Altre richieste	13
3	Macchine per la QCD	16
3.1	Fisica computazionale	16
3.2	Il problema forma la macchina	17
3.3	Calcolo parallelo	18
3.4	Il modello SIMD	19
3.5	Limiti del parallelismo implicito	21
3.6	Influenza della tecnologia	23
3.7	Calcolatori per la fisica	25
3.7.1	Macchine dedicate	26
3.7.2	Aggregati	27
3.7.3	Supercalcolatori commerciali	29
3.8	Il panorama internazionale	30
3.8.1	Il CP-PACS	31
3.8.2	La Columbia QCDSP	32
3.8.3	In europa: APEmille	32
3.9	La prossima generazione	34
3.9.1	Columbia University: QCDOC	34
3.9.2	apeNEXT	34
4	L'operatore di Dirac e l'architettura di apeNEXT	36
4.1	L'operatore di Dirac sul reticolo	37
4.2	Il dimensionamento della macchina dall'analisi di D	38
4.2.1	Calcolo di R	38
4.2.2	Calcolo di ρ	40
4.3	La memoria	43
4.4	Il register file	44
4.5	Il processore matematico	44
4.5.1	Effetto del pipelining sull'efficienza floating point	45

4.6	La rete di comunicazione	48
4.6.1	Comunicazioni sincrone	48
4.6.2	Comunicazioni con prefetch	51
4.6.3	Compensazione nei siti di bordo	52
4.6.4	Altre caratteristiche	53
4.7	La generazione degli indirizzi	53
4.8	Sincronizzazione e controllo globale	54
4.9	Il microcodice	55
4.10	Architettura del nodo di apeNEXT	58
4.10.1	Il register file	58
4.10.2	Il collegamento tra memoria e rete	59
4.10.3	La memoria fisica	61
4.10.4	L'unità floating point	62
4.10.5	La rete di trasmissione	63
4.10.6	Lo stack delle condizioni	67
4.10.7	Le condizioni globali	68
4.10.8	La cache istruzioni	69
4.10.9	Lo slow control	72
4.11	Conclusioni	74
5	Codice e misure	75
5.1	Obbiettivi e metodo	75
5.2	Quantità misurate	76
5.2.1	L'efficienza del microcodice	77
5.2.2	L'efficienza della rete	77
5.2.3	L'efficienza della memoria	78
5.2.4	L'efficienza netta	79
5.2.5	Il tempo del calcolo	80
5.2.6	Il campo di riferimento	80
5.3	Gli strumenti a disposizione	80
5.4	Il programma	81
5.5	La prima versione: comunicazioni sincrone	84
5.6	Utilizzo della AGU	84
5.7	Riduzione dei fifo stretch	86
5.8	Il prefetch completo	88
5.8.1	Il register windowing	91
5.8.2	L'importanza del prefetch	91
5.9	Riduzione dei memory stretch	91
5.10	Considerazioni finali	95
5.10.1	Il rapporto con i codici di produzione	95
5.10.2	La gestione della fifo nei linguaggi di alto livello	95
5.10.3	Ulteriori modifiche	95
	Conclusioni	97
A	Appendice	98
A.1	Il modello MIMD	98
A.2	Misurare in flops	98
A.3	Rappresentazioni approssimate	99
A.4	Le inizializzazioni nel codice dell'operatore di Dirac	101

Capitolo 1

Introduzione

La descrizione teorica tra i costituenti fondamentali della materia è basata attualmente sul cosiddetto Modello Standard. Il modello è basato sul concetto di “invarianza di gauge”, la generalizzazione della proprietà formale osservata per la prima volta nelle equazioni di Maxwell. Le previsioni del modello standard sono state sottoposte a molte stringenti verifiche sperimentali, soprattutto nella parte che riguarda le interazioni debole, elettromagnetica, e quella forte ad alte energie. La forza forte è descritta da una teoria di gauge nota come QCD (cromodinamica quantistica); ad energie dell'ordine del GeV o più basse non è nota alcuna approssimazione realistica che permetta di trattare analiticamente la QCD ed ottenerne previsioni quantitative.

Le tecniche di simulazione numerica al calcolatore sono l'unica possibilità attualmente accessibile per estendere le verifiche della teoria a questo dominio. La complessità dei metodi numerici e delle risorse richieste ha motivato nel passato molti gruppi di ricerca a progettare delle macchine dedicate allo scopo di affrontare simulazioni di così grande scala.

In questa tesi verrà esposto il lavoro svolto dal candidato nel gruppo Ape dell'INFN di Pisa. APE è un progetto portato avanti dall'Istituto Nazionale di Fisica Nucleare in collaborazione con DESY e la Université Paris-Sud. Il progetto ha lo scopo di realizzare una apparecchiatura ottimizzata per la simulazione numerica delle teorie di gauge su reticolo (Lattice Gauge Theories, LGT).

In un progetto complesso in cui molte scelte vengono prese con largo anticipo, e non sono poi modificabili, è essenziale avere una stima precisa e dettagliata del comportamento della macchina sui problemi tipici ai quali sarà applicata.

Il contributo principale del candidato al progetto è stato l'esecuzione di una serie di misure che hanno permesso di dare una stima realistica dell'efficienza della futura generazione di macchine APE, attualmente in fase di progetto, su un tipico kernel di calcolo, l'applicazione dell'operatore di Dirac in un quadrispazio discreto. Le misure dettagliate che sono state effettuate sono state descritte nel quinto capitolo.

La possibilità ed opportunità della costruzione della macchina sono sostenute dall'attuale panorama della ricerca internazionale nel campo delle teorie di gauge su reticolo ed in particolare sulla QCD. Il secondo ed il terzo capitolo illustrano queste motivazioni; il quarto capitolo mostra come la forma del particolare problema fisico da affrontare determini l'architettura ottimale della macchina, detti altre scelte di progetto, ed in ultima analisi fissi il carattere ed il metodo seguito nel lavoro.

Il sesto capitolo contiene il risultato finale di questa tesi, che è una stima molto accurata delle prestazioni di apeNEXT su uno dei kernel di calcolo più rilevanti per le simulazioni di LGT.

Capitolo 2

Scenario e motivazioni

Il lavoro di tesi svolto si è inserito nell'ambito dello sviluppo della prossima generazione di macchine APE, denominata apeNEXT. Il progetto apeNEXT mira a dotare i membri della collaborazione di una potenza di calcolo dell'ordine di 10 Tflops (1 Tflops corrisponde a 10^{12} operazioni in virgola mobile al secondo) ciascuno, nei prossimi 2-3 anni. Questa è la potenza di calcolo, secondo recenti stime, necessaria per lo svolgimento di simulazioni di QCD sul reticolo con fermioni dinamici e realistici parametri fisici.

2.1 Le teorie di campo

Il Modello Standard è la teoria attualmente accettata per la descrizione delle interazioni fondamentali, esclusa quella gravitazionale. Dal modello sono state estratte predizioni quantitative per una grande quantità di fenomeni, che ne hanno verificato la validità in quasi tutti i regimi accessibili sperimentalmente.

Un dominio in cui questo non è stato possibile fino ad alcuni anni fa è quello delle interazioni forti ad energie basse (dell'ordine del GeV o inferiori). Il modello standard prevede che l'interazione forte sia descritta da una teoria di gauge $SU(3)$; la predizione di proprietà quali masse ed elementi di matrice della materia ordinaria, inclusi mesoni ed adroni, dipende dalla possibilità di calcolare valori di aspettazione di opportune osservabili sullo stato di vuoto.

Rivediamo qui molto brevemente alcuni passaggi che portano alla formulazione delle teorie di gauge su reticolo. Quando l'equazione del moto di una teoria di campo è espressa nella forma

$$\hat{L}\phi = 0, \tag{2.1}$$

dove ϕ è una funzione d'onda (eventualmente con degli ulteriori indici) definita in ogni punto del quadrispazio, se ne può scrivere la più generica soluzione

$$\phi(x_\mu) = \sum_j a_j f_j(x_\mu) + \sum_k b_k f_k^*(x_\mu).$$

Le f_i e g_i sono un set completo di soluzioni della (2.1) e le somme sono eventualmente intese come integrali dove lo spettro fosse continuo.

A questa funzione si associa un operatore corrispondente $\hat{\phi}$ definito come $\hat{\phi} = \sum_j \hat{a}_j f_j(x) + \sum_k \hat{b}_k f_k^*(x_\mu)$; le \hat{a}_j e \hat{b}_k sono operatori su uno spazio di Hilbert definito costruttivamente a partire da uno stato di vuoto $|0\rangle$. Se si richiede che gli operatori (di Fock) obbediscano ad opportune regole di commutazione, si vede che dallo stato di vuoto è possibile costruire uno spazio sul quale sarà definito $\hat{\phi}$.

In teoria di campo il valore medio di qualsiasi osservabile $\langle 0 | \mathcal{O}(x_1, x_2, \dots, x_n) | 0 \rangle$ sullo stato di vuoto può essere ricavata conoscendo il cosiddetto propagatore, definito come

$$p(x_1, x_2) \equiv \langle 0 | T \hat{\phi}^*(x_1) \hat{\phi}(x_2) | 0 \rangle,$$

dove T è l'operatore che ordina temporalmente i suoi fattori. Dalla definizione si può dimostrare che p (che è una funzione locale di due coordinate) è la funzione di Green dell'operatore \hat{L} , cioè vale

$$\hat{L} p(x_1, x_2) = \delta^4(x_1 - x_2) \dots \quad (2.2)$$

dove i puntini indicano che ci possono essere altre funzioni δ (di Kronecker) se la ϕ possiede degli indici discreti.

Per ottenere tutti i valori di aspettazione desiderati ci si propone quindi di calcolare i propagatori

$$p_{i,j}(x_1, x_2)$$

per ogni particella a cui siamo interessati. Gli indici delle variabili discrete (ad esempio spin e colore) saranno nel seguito sottintesi. Il propagatore non può essere calcolato numericamente dalla sua definizione.

Il problema può essere affrontato esprimendo p nella sua forma alternativa di path integral dovuta a Feynman:

$$\langle 0 | T \hat{\psi}^*(x_1) \hat{\psi}(x_2) | 0 \rangle = \int D\phi(x_\mu) e^{\frac{iS(\phi)}{\hbar}} \phi^*(x_1) \phi(x_2), \quad (2.3)$$

dove $\int D\phi(x_\mu)$ indica l'integrale al variare indipendente di tutti i gradi di libertà del sistema, cioè su tutti i possibili percorsi classici. L'azione S è un funzionale fornito dalla teoria che associa un numero a ciascuna di queste configurazioni. In questa forma il problema può essere trattato numericamente, almeno in linea di principio, se si impone che le coordinate x_μ possano assumere solo un insieme discreto di valori.

La discretizzazione più semplice sostituisce il 4-volume infinito con uno finito, suddiviso da una griglia cartesiana in $N = L^3 T$ ipercubi (siti del reticolo), con opportune condizioni al contorno (ad es. periodiche). Il path integral in $D\phi$ diventa così un integrale multiplo sulle N variabili (complesse) $d\phi^{(1)} \dots d\phi^{(N)}$, che sono il valore di ϕ su ciascuno dei punti del reticolo (lattice).

2.2 Il caso non abeliano

Il calcolo di una osservabile può essere espresso in maniera analoga alla (2.3) nel caso della teoria delle interazioni forti, a meno di difficoltà di cui non ci occupiamo [1]:

$$\langle \phi^* \phi \rangle = \int [DA_\mu(x)] [D\psi(x)] [D\bar{\psi}(x)] e^{-S[A_\mu, \psi]/\hbar} \phi^* \phi \quad (2.4)$$

Nella azione compaiono la costante di accoppiamento forte e le masse dei quark. L'esponente è stato reso reale per mezzo di una continuazione analitica $t \rightarrow -i\tau$, $\tau \in \mathbf{R}$; in questa formulazione i percorsi lontani da quello classico si cancellano perché moltiplicati per un esponenziale piccolo invece che oscillante. Questa caratteristica desiderabile permette di restringere il campionamento dell'integrando alle regioni dello spazio delle configurazioni che danno un contributo maggiore al path integral (*importance sampling*).

I path integral nelle variabili fermioniche possono essere svolti analiticamente. La discretizzazione della (2.4) è stata proposta per la prima volta da Wilson [2]. Sul reticolo i campi di gauge $A_\mu(x)$ sono scritti in funzione di matrici U_μ complesse appartenenti al gruppo $SU(3)$. Si associa una matrice a ciascuno dei segmenti orientati che congiungono il sito x a quello $x + \hat{\mu}$. Il risultato è che per conoscere il propagatore di un quark si risolve l'integrale multiplo, al variare di ogni componente delle matrici di gauge, di ogni sito:

$$\langle \phi^* \phi \rangle = \int \phi^* \phi \det \mathcal{D}[U, m] e^{-S[U]/\hbar} \prod_{n, \mu} dU_\mu(n) \quad (2.5)$$

dove \mathcal{D} è l'operatore di Dirac sul reticolo, che nel limite continuo diventa il consueto operatore differenziale $(\gamma_\mu \partial^\mu + m)$. m è la massa del quark considerato e l'indice di sito $n = 1 \dots L^3 T$ ha preso il posto delle coordinate continue x_μ .

2.3 Simulazioni numeriche su grande scala

Questa sezione introdurrà l'approccio numerico alla simulazione di problemi QCD nella pratica. Si dice talvolta che queste simulazioni sono svolte “da principi primi”. Ci sembra interessante rivedere brevemente quali parametri entrano nelle misure estratte da una di queste simulazioni.

Le masse dei quark e la costante di accoppiamento forte. Questi sono i soli parametri liberi della teoria. Le stesse “masse” dei quark sono parametri che non possono essere osservati direttamente; essi sono stati ottenuti [3, 4] tramite procedure di fit in modo da far corrispondere le predizioni di simulazioni LQCD con i valori sperimentali.

Le dimensioni dello spazio e la spaziatura del reticolo. Questi parametri sono introdotti nella discretizzazione e hanno necessariamente valori finiti nelle simulazioni al computer. Nella pratica si richiede che lo spazio sia sufficientemente grande da contenere un multiplo della lunghezza Compton del più leggero stato rilevante nel calcolo ed allo stesso tempo che il passo del reticolo non introduca un limite superiore inaccettabile allo spazio dei momenti.

La maggior parte degli sforzi su simulazioni di QCD nei prossimi anni saranno rivolti a ridurre le incertezze su alcune grandezze già determinate [5]. È naturale, quindi, che nella costruzione di macchine dedicate alla QCD ogni sforzo sia rivolto a massimizzare il numero di configurazioni analizzabili, a parità di tempo macchina impiegato.

2.4 L’importance sampling

La formulazione path integral nel tempo immaginario mette in luce una analogia tra la teoria di campo e la meccanica statistica. Il path integral, anche se discretizzato, si traduce in un numero proibitivamente elevato di integrazioni (una variabile di integrazione per ogni componente di U , per ogni sito, per ogni direzione).

L’analogia con la meccanica statistica suggerisce che per conoscere il valore “macroscopico” di una funzione termodinamica (osservabile) è tuttavia sufficiente misurarla percorrendo lo spazio delle fasi vicino alla configurazione di minima azione. Nella pratica il path integral è eseguito in maniera approssimata al variare di U_μ per mezzo di metodi misti stocastici-deterministici, come l’algoritmo di Metropolis e simili [6].

Le tecniche numeriche della QCD su reticolo sono basate su questo approccio. Lo studio degli errori di un calcolo eseguito secondo il metodo Montecarlo è fuori dallo scopo del lavoro. Tuttavia è evidente che i risultati di una simulazione statistica sono soggetti a tre potenziali fonti di errore:

Errori sistematici, per esempio dovuti ad effetti di bordo ed alla dimensione finita del reticolo, oppure alla possibilità di esplorare lo spazio delle configurazioni vicino ad un minimo soltanto locale dell’azione.

Errori statistici: l’errore sul risultato sarà tanto più piccolo quante più configurazioni indipendenti e significative si riescono ad analizzare.

Errori dovuti ad approssimazioni adottate Sono essenzialmente due: per le masse dei quark si assumono valori più grandi di quelli fisici in modo da rendere il problema trattabile; e l'approssimazione cosiddetta “quenched”, che sarà descritta nel seguito.

Tutte le cause di errori elencate ad eccezione del quenching possono essere controllate efficacemente. Tipicamente si compiono delle estrapolazioni al variare di a e m verso i limiti corretti, a patto di avere abbastanza punti vicini alla regione di interesse. Lo studio sistematico di questi parametri per ricavare lo spettro delle masse previste dalla teoria nell'approssimazione quenched è un compito formidabile che è stato completato di recente [7]. L'obiettivo che si vuole raggiungere nei prossimi anni è realizzare uno studio analogo, in cui l'approssimazione non controllata del quenching sia rimossa.

2.5 Il calcolo del propagatore

Nella equazione (2.5) compare il determinante di \mathcal{D} , che è un operatore lineare che agisce su uno spazio di dimensione finita $L^3 \times T \times 3 \times 4$, cioè una matrice con $O(10^{15})$ elementi. (Il fattore 3 tiene conto del numero di colori ed il 4 delle componenti dello spin.) Scrivendo \mathcal{D} in forma di matrice, la proprietà (2.2) diventa

$$D[U, m]_{n, n', s, s'} \phi_{n', l, s', r} = \delta_{n, l} \delta_{s, r} \quad (2.6)$$

e può essere usata per calcolare il propagatore ϕ , perché è equivalente a $\phi = D^{-1}$. La lettera U indica la configurazione di campi di gauge che si sta provando, gli indici n sono gli indici di sito che hanno preso il posto delle coordinate \mathbf{x} ed \mathbf{x}' , s rappresenta gli indici discreti di spin e colore. A causa delle invarianze per traslazioni e di gauge si considera il propagatore ad un indice $p_{0,i}(x_1 - x_2)$ anziché quello a due indici $p_{j,i}(x_1, x_2)$: ponendo $l = r = 0$ consideriamo una sola riga di D^{-1} .

Un problema di interesse reale potrà essere studiato in un reticolo delle dimensioni dell'ordine di 40^4 , così la dimensione della matrice sarà $N \times N$, con $N \sim 3 \cdot 10^7$. L'inversione numerica diretta di una matrice di queste dimensioni è un compito formidabile: il più veloce metodo di inversione noto richiede un numero di operazioni dell'ordine di N^α , con $\alpha \simeq 2.38$ [8]. Il calcolo del propagatore corrispondente ad una sola configurazione su uno spazio delle dimensioni indicate richiederebbe quindi circa $7 \cdot 10^{17}$ operazioni complesse in virgola mobile, che impegnerebbero una macchina da 1 Tflops con efficienza pari ad uno per circa 8 giorni.

L'inversione della matrice viene eseguita con metodi approssimati iterativi, quale ad esempio il gradiente coniugato o il minimo residuo [9]. Questi metodi cercano la soluzione dell'equazione

$$M \phi = k$$

dove M è la matrice da invertire, ϕ è la soluzione cercata e k è un vettore assegnato. Indichiamo con ϕ_n la soluzione approssimata dopo n passi, con r_n il residuo definito da

$$r_n = k - M \phi_n$$

e con (ϕ, ψ) il prodotto hermitiano di due vettori ϕ e ψ . I seguenti passi realizzano ad esempio l'algoritmo del minimo residuo:

$$\begin{aligned}\phi_{n+1} &= \phi_n + \frac{(Mr_n, r_n)}{|Mr_n|^2} r_n, \\ r_{n+1} &= \frac{(Mr_n, r_n)}{|Mr_n|^2} Mr_n\end{aligned}$$

L'utilità del metodo sta nel fatto che in queste formule la matrice M è usata solo per moltiplicare vettori: il metodo approssimato ne ottiene l'inversa applicandola ripetutamente ai residui.

La maggior parte delle operazioni numeriche di una simulazione nell'approssimazione quenched sono impiegate per l'inversione della matrice che realizza l'operatore di Dirac. Come si è visto in questa sezione, l'inversione viene ottenuta con un metodo approssimato come quello descritto; in conclusione il tempo di calcolo è quasi interamente impiegato nella *applicazione* dell'operatore di Dirac sul reticolo.

2.6 Il quenching

Nell'equazione (2.5) compare il determinante di \mathcal{D} . Il calcolo numerico (non diretto) di questo determinante è possibile, ma richiede una quantità di operazioni numeriche ingente, perché deve essere ripetuto ad ogni passo del processo che aggiorna i campi U_μ . Questa è la ragione per cui, fino ad oggi, la maggior parte delle simulazioni sono state fatte con una approssimazione nota convenzionalmente come *quenching*, che consiste nel sostituire questo determinante con una costante.

L'approssimazione quenched si è dimostrata fruttuosa ed i valori delle grandezze misurate mediante simulazioni sono stati trovati coincidenti con a quelli sperimentali entro margini del 20% circa. Non è possibile, però, studiare sistematicamente l'errore introdotto da questa assunzione.

Lo scenario sta cambiando in questi anni: la potenza di calcolo che si stima che potrà essere accessibile ai fisici per condurre simulazioni di LGT nei prossimi anni suggerisce che la ricerca nei prossimi anni potrà essere orientata verso simulazioni in cui questa approssimazione è rimossa. Questo tipo di simulazioni prende il nome di full-QCD ovvero *unquenched*, o anche con fermioni dinamici. Un problema

full-QCD ha un costo computazionale tipicamente $10^3 - 10^5$ volte più grande del corrispondente unquenched [10].

Anche il calcolo numerico del determinante non può essere eseguito in maniera diretta e viene ricondotto all'inversione dell'operatore di Dirac. L'applicazione dell'operatore, quindi, è la parte che domina computazionalmente anche nelle simulazioni full-QCD.

2.7 Verso la QCD completa

Nel 1990 N. H. Christ ha passato in rassegna gli articoli pubblicati su riviste di fisica nucleare ed ha stimato approssimativamente la quantità di operazioni numeriche che erano state necessarie agli autori per ottenere i risultati presentati [11]. I risultati dello studio sono presentati nel grafico in figura 2.1, che mostra la più grande potenza di calcolo che è stata impiegata dai fisici teorici anno per anno per una singola simulazioni di QCD su reticolo. (I punti successivi al 1990 sono stime del gruppo APE.)

La retta mostra i risultati di un fit esponenziale sui punti mostrati, esclusi i primi due. Il fit mostra che la scala delle simulazioni è approssimativamente *raddoppiata ogni anno* dal 1985 ($\tau_2 = 11.9 \pm 0.7$ mesi). Se questo andamento rimarrà invariato nei prossimi anni, l'extrapolazione del grafico nel futuro mostra che nell'anno 2003 il costo delle simulazioni raggiungerà la scala del 20 Tflops-mese. È quindi sensato ipotizzare che in quell'anno la ricerca potrà contare su un certo numero di macchine con potenze dell'ordine dei 5 Tflops ciascuna ⁽¹⁾.

La figura indica approssimativamente il numero totale di operazioni richieste per una misura. Qualsiasi numero di operazioni può senz'altro essere raggiunto facendo lavorare per un tempo sufficientemente lungo calcolatori poco veloci, ma questa soluzione non è realistica. Una misura via simulazione verrà difficilmente iniziata se non può essere conclusa in qualche mese. Di conseguenza possiamo assumere che l'unico modo di accedere a misure da 2 Tflops-mese sia avere a disposizione calcolatori di almeno 2 Tflops.

Si pone a questo punto la questione di notevole interesse, se questa potenza di calcolo renderà possibile l'eliminazione della approssimazione quenched.

La collaborazione SESAM/T χ L ha di recente pubblicato delle stime sul costo computazionale delle simulazioni full-QCD, che sono riassunte nella formula semiempirica data da G \ddot{u} sken et al. [3]:

$$N = 1.7 \cdot 10^7 \cdot (L^3 T)^{\frac{4.55}{4}} \cdot a^{-7.25} \cdot m_{ps}^{-2.7} \quad \text{Tflops} \cdot \text{mese} \quad (2.7)$$

¹ Una breve discussione sul possibile abuso di questa unità di misura è data in appendice a pagina 98.

dove a è il passo reticolare (in fm), L e T sono le dimensioni spaziali e temporali dello spazio simulato (in fm) ed $m_{ps} = m_\pi/m_\rho$ è il rapporto tra le masse delle particelle π e ρ .

La formula dà una stima approssimata del numero di operazioni aritmetiche reali necessarie per generare una configurazione statisticamente indipendente con algoritmi del tipo HMC (Hybrid Montecarlo), al variare dei parametri indicati.

Il rapporto m_π/m_ρ è un dato di uscita delle simulazioni e dipende delle masse dei quark assunte nelle simulazioni. Le masse dei quark non sono note sperimentalmente, mentre lo è il rapporto $m_\pi/m_\rho \simeq 0.1785$. Questo è il motivo per cui la formula è stata espressa in funzione di questa grandezza.

Idealmente si vorrebbero condurre simulazioni in cui le masse dei quark fossero fissate e tali da produrre il rapporto corretto per m_π/m_ρ ; l'altro parametro libero della teoria, la costante di accoppiamento forte, è noto sperimentalmente. In questo regime si potrebbe determinare le proprietà di ogni adrone e mesone per mezzo di simulazione, con precisione limitata solo dall'accuratezza della teoria. Purtroppo questa possibilità non è ancora alla portata dei calcolatori della attuale generazione, né prevedibilmente lo sarà ancora per parecchi anni.

Il motivo può essere compreso osservando l'andamento della figura 2.2. Le curve in figura sono ottenute dalla formula 2.7 data sopra e mostrano il numero di Tflops-mese ⁽²⁾ necessari per condurre una simulazione unquenched, assumendo un reticolo di 4 fermi di lato, al variare delle masse dei quark (presenti implicitamente in ascissa). Le quattro curve corrispondono a valori diversi del passo reticolare a . Lo studio al variare di a è necessario perché si desidera estrapolare i risultati verso il limite del continuo $a \rightarrow 0$.

Le curve nella regione $m_\pi/m_\rho \simeq 0.1785$, in cui i parametri della teoria assumono valori fisici, indicano un costo computazionale di $O(10 - 1000)$ Tflops-mese. Non solo questa potenza è al momento irraggiungibile agli enti di ricerca, ma è probabilmente sottostimata: l'analisi delle proprietà di una particella richiede infatti un reticolo di dimensioni almeno quattro volte la lunghezza Compton della stessa: $\lambda \sim \hbar/(mc)$. Le linee orizzontali ed il punto delimitano il limite sinistro delle regioni in cui questa condizione è soddisfatta. Nella regione "fisica" di m_π la dimensione $L = 4$ fm, $T = 2L$ assunta in figura è largamente insufficiente e va quindi aumentata, con la conseguente ulteriore crescita del tempo di calcolo richiesto.

Dal confronto delle due curve 2.2 e 2.1 si deduce che nel 2003 alla ricerca sulla QCD sarà disponibile una potenza di calcolo dell'ordine dei 20 teraflops. La retta orizzontale dei 20 Tflops-mese interseca le curve dei costi delle simulazioni unquen-

²Un Tflops-mese corrisponde a circa $2.59 \cdot 10^{18}$ operazioni aritmetiche in virgola mobile (la precisione ed il tipo di operazione non sono specificati). Talvolta la stessa unità viene citata come Tflops-mese *sostenuti*, per sottolineare che il numero di operazioni calcolate è il risultato dovuto ad un mese di lavoro ininterrotto di una macchina da un Tflops, con efficienza pari ad uno.

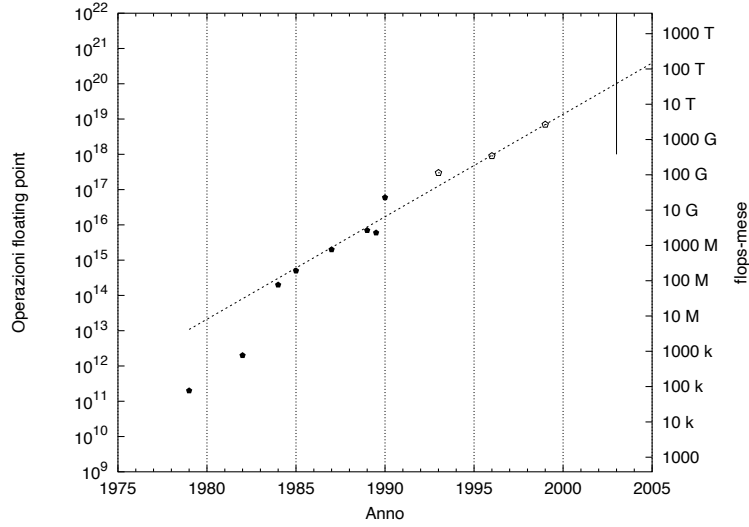


Figura 2.1: *Quantità di operazioni in virgola mobile impiegate per simulazioni di QCD su reticolo negli anni passati.* Il grafico mostra una stima approssimativa del più grande numero di operazioni in virgola mobile usate in una singola simulazione di QCD su reticolo in funzione del tempo. I punti in nero sono dati in [11], quelli più recenti vengono da stime del gruppo APE [12]. La retta è un fit dei dati mostrati, esclusi i primi due, con una funzione esponenziale. Il tempo di raddoppio è $\tau = 11.9 \pm 0.7$ mesi.

ched; la potenza di calcolo prevista sarà sufficiente ad affrontare problemi full-QCD nei seguenti domini:

- Sistemi fisici di dimensioni di $L \simeq 4$ fm e passo reticolare $a \simeq 0.1 \dots 0.05$ fm (corrispondenti a reticoli di $\sim 48^3 \times 96$, a valori di m_{ps} simili a quelli delle attuali simulazioni.
- Masse dei quark tali da avere $m_\pi/m_\rho \simeq 0.35$. Questo rapporto corrisponde a masse dei quark leggere abbastanza da potere estrapolare da esse dei valori fisici [7].

2.8 Altre richieste

Sebbene lo studio della teoria QCD completa sia lo scopo principale della prossima generazione di calcolatori, le simulazioni quenched continueranno ad avere una notevole importanza in particolare per lo studio della fenomenologia dei quark pesanti. Uno studio realistico in questo caso richiede reticoli dell'ordine dei 100^4 . Per ogni sito occorre considerare i valori di almeno due propagatori: di conseguenza il limite principale per le simulazioni quenched non sarà la potenza di calcolo ma la disponibilità di una quantità sufficiente di memoria.

Riassumo quindi gli obbiettivi previsti per le simulazioni LQCD di grande scala negli anni 2003 – 2006, che saranno:

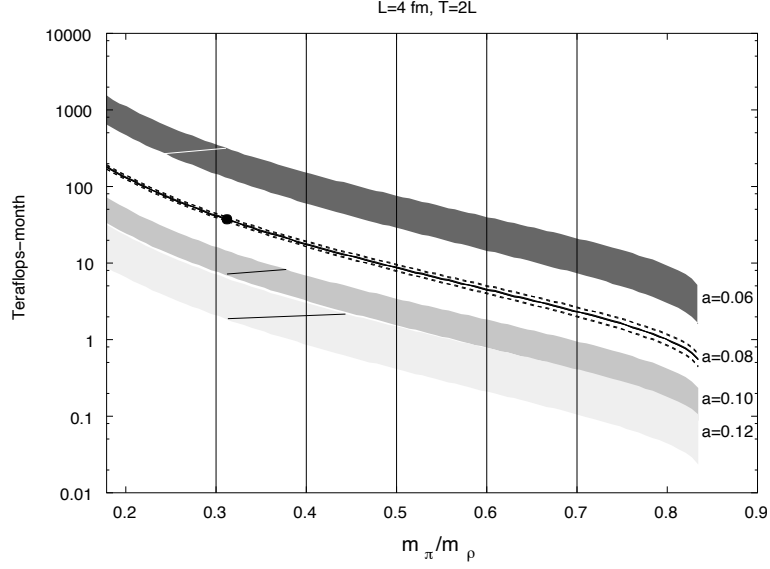


Figura 2.2: Stime del costo computazionale per la generazione di 100 configurazioni di vuoto indipendenti in full QCD, al variare del rapporto tra le masse dei quark e del passo reticolare a . La dimensione del lato del reticolo è assunta $L = 4$ fm.

- Simulazioni full-QCD su reticoli di dimensioni dell'ordine di $48^3 \times 96$ e passo reticolare $a = 0.1 \dots 0.05$ fm. Le masse dei quark impiegate saranno più vicine a valori realistici, con $m_\pi/m_\rho \simeq 0.35$. Questi obiettivi non saranno entrambi raggiunti nella stessa simulazione.
- Simulazioni in approssimazione quenched su reticoli dell'ordine di 100^4 .

Le motivazioni qui riportate per la realizzazione di sistemi di calcolo di prestazioni adatte allo studio delle LGT sono assai simili alle conclusioni di un commissione, nominata dall'ECFA per portare a termine uno studio sulle risorse di supercalcolo che saranno richieste nei prossimi anni dai fisici europei del campo e sulle opportunità scientifiche che le risorse potrebbero aprire (referenza [5]).

Il comitato nominato dall'ECFA ha fatto propria la stessa stima che è stata presentata nelle sezioni precedenti ed ha concluso il proprio lavoro con la seguente raccomandazione:

Future strategy must be driven by the requirements of the physics research program. From the Technology Review in section 8 we conclude that it is both realistic and necessary to aim for machines of $O(10$ Tflops) processing power by 2003. As a general guide, such machines will enable results to be obtained in unquenched simulations with a similar precision to those currently found in quenched ones.

L'interesse del comitato è evidentemente sostenuto dalla necessità di coordinare gli sforzi europei in modo da fare fronte alla forte collaborazione-competizione

internazionale nel ramo, di modo da mantenere la posizione di riguardo che hanno tenuto i fisici europei nei due decenni passati.

Capitolo 3

Macchine per la QCD

L'obiettivo di simulare un quadrispazio discreto e quello di raggiungere la potenza di calcolo indicata sopra, insieme alla valutazione delle tecnologie correnti, condizionano fortemente l'architettura delle macchine dedicate a questo scopo. La scelta di organizzare un reticolo tridimensionale di nodi di calcolo appare naturale, come lo è stata nei progetti precedenti dello stesso gruppo (APE 100 ed APEmille). Il terzo capitolo della tesi dà una panoramica di come il calcolo parallelo è stato adattato a problemi di fisica.

3.1 Fisica computazionale

Ci sono numerosi campi della ricerca pura che traggono benefici dalla simulazione numerica a mezzo di supercalcolatori. Qui cito tre modelli di esempio che possono essere affrontati per simulazione numerica, allo scopo di dare una visione intuitiva del carattere generale del problema.

Il modello di Ising è stato proposto originariamente come modello per spiegare il ferromagnetismo dei metalli e la sua scomparsa ad alte temperature ed è stato successivamente generalizzato in varie direzioni.

Nella sua formulazione originale prevede un sistema di N particelle, ciascuna delle quali può essere in uno stato di spin $s_i = \pm 1$. Le particelle sono disposte su un anello monodimensionale ed ognuna di esse è soggetta ad una interazione con le sue due prime vicine della forma

$$H = -J \sum_{\{i,j\}} s_i s_j$$

La notazione $\{i, j\}$ indica che nella somma vanno considerate tutte le coppie di primi vicini, cioè, nel caso monodimensionale, quelle per cui $|i - j| = 1$. Per $J > 0$ l'energia del sistema è tanto minore quante meno “frontiere” esistono tra domini di spin opposto.

Il modello di Ising in una dimensione è risolubile analiticamente; in ogni caso è possibile usare metodi numerici (Metropolis ad esempio) per generare un numero arbitrario di configurazioni indipendenti che campionano l'insieme canonico [13].

Il modello Lattice-Boltzmann. Lo studio del moto di un fluido a grandi numeri di Reynolds è di interesse pratico oltre che di ricerca. L'equazione di Boltzmann può essere discretizzata su un reticolo tridimensionale ed integrata nel tempo per ottenere (ad esempio) $\mathbf{V}(\mathbf{x}, t)$ e le sue funzioni di correlazione. L'equazione che governa il sistema è una equazione differenziale alle derivate parziali; anche per risolvere la sua corrispondente discreta si integrano le equazioni del moto calcolando il valore di $\mathbf{V}(\mathbf{x}, t)$ in funzione di $\mathbf{V}(\mathbf{x}', t - \Delta t)$ (e le altre quantità negli stessi punti), con \mathbf{x}' “lontano” da \mathbf{x} di al più un passo reticolare lungo uno dei tre assi.

La Qcd. Come è stato accennato nel capitolo precedente, la soluzione numerica di un propagatore $\phi(0, x_\mu)$ nel quadrispazio si ottiene mediante la ripetuta applicazione dell'operatore di Dirac allo stesso. L'operatore è locale perché è essenzialmente una somma di quattro derivate $\gamma_\mu \partial^\mu$ e, discretizzato su un reticolo a quattro dimensioni, accoppia solo i siti primi vicini come nell'esempio precedente.

I tre casi sono stati scelti perché tutti presentano qualche tipo di località. Essa deriva – nel primo caso – dalla forma della interazione a primi vicini; negli altri due dalla località degli operatori differenziali, scritti nello spazio diretto.

3.2 Il problema forma la macchina

La decisione se una simulazione numerica su grande scala sia praticamente affrontabile richiede la considerazione di un certo numero di grandezze che dipendono dal particolare calcolo che si vuole affrontare:

La quantità di memoria necessaria è proporzionale al numero di grandezze di ingresso, di uscita e temporanee di cui è necessario tenere traccia durante ogni passo del calcolo della soluzione.

Il numero di operazioni aritmetiche è il conteggio del numero di volte che occorrerà sottoporre una coppia di valori (per lo più in virgola mobile) ad una operazione numerica quale somma, differenza o prodotto. Altre operazioni numeriche, quali quoziente, esponenziazione oppure calcolo di funzioni trigonometriche, vanno generalmente considerate in modo speciale, se sono presenti in numero considerevole. Ciò accade perché spesso il calcolo di queste

funzioni più complicate viene ricondotto per motivi semplicità a sequenze talvolta molto lunghe di operazioni aritmetiche più semplici (prodotti, somme ed estrazioni di bit).

Complessità dell'accesso ai dati I modelli presi più sopra come esempi hanno in comune la caratteristica che il procedimento con cui accedono ai dati è alquanto regolare. Ancora più importante è il fatto che il compito può essere scomposto dividendo il reticolo fisico in maniera naturale tra più processori, per ognuno dei quali la sequenza di accesso ai dati è sostanzialmente la stessa.

Idealmente si vorrebbe costruire (ed usare!) una macchina in cui queste tre caratteristiche siano “le migliori possibile”. Questa richiesta è vaga ed irragionevole perché in prima approssimazione la attuale tecnologia può fornire prestazioni arbitrariamente elevate, pagando un opportuno prezzo sia economico che di complessità.

La strategia da seguire è evidentemente quella di bilanciare gli aspetti menzionati (ed altri ancora) in modo che nessuno dei tre limiti molto più degli altri la soluzione *del problema di riferimento*. Se la macchina non fosse bilanciata, alcune delle sue componenti sarebbero sistematicamente sottoutilizzate; il costo della risorsa sprecata sarebbe potuto essere impiegato per la costruzione di un numero maggiore di macchine.

3.3 Calcolo parallelo

È ragionevole e corretto immaginare che una macchina digitale in cui tutte le operazioni in virgola mobile siano affidate ad un singolo componente possa raggiungere prestazioni essenzialmente limitate dalla frequenza alla quale si può temporizzare il chip. I comuni processori commerciali a scopo generico sono supportati da grandi interessi commerciali e sono quanto di più veloce la tecnologia permetta di realizzare su un singolo componente di silicio. Eppure il più veloce processore commerciale ha una potenza di circa 1 Gflops, tre-quattro ordini di grandezza inferiore all'obiettivo previsto per la QCD completa nel capitolo precedente.

Un modo naturale per realizzare una macchina di alte prestazioni è di raggruppare un numero elevato di processori semplici e di affidare a ciascuno di essi la soluzione di una parte del problema. Se, per esempio, si considera uno spazio discretizzato in maniera regolare di forma parallelepipedica (eventualmente in più dimensioni), lo si può dividere in un certo numero di parallepiedi più piccoli; per risolvere il problema originale si potrà applicare lo stesso algoritmo trovato per lo spazio “fisico” ad ognuno dei domini più piccoli (figura 3.3).

Questa idea si basa sull'osservazione che i sistemi che si desiderano studiare, come quelli elencati sopra, presentano una dinamica quasi del tutto *omogenea* nello

spazio. (L'omogeneità è completa quando si impongono le condizioni periodiche al contorno, come si fa abitualmente.)

Possiamo dire, in altri termini, che abbiamo osservato il parallelismo implicito nella fisica e nel suo modello e lo abbiamo parzialmente riprodotto nell'hardware digitale che li deve simulare. È evidente che una parte della simulazione può comunque richiedere l'esecuzione di operazioni *globali*, come ad esempio la somma del quadrato di una grandezza su tutto lo spazio, oppure il controllo che una qualche misura di errore locale sia ovunque minore di una soglia fissata. I casi come questi, le comunicazioni remote, le condizioni al contorno ed in generale in tutte le situazioni in cui il parallelismo viene a mancare andranno considerati con attenzione e trattati a parte.

Più in alto si è sottolineato una altra caratteristica comune ai tre problemi: la loro *località*. La località non è una condizione necessaria per la scomposizione (“parallelizzazione”) di un modello, ma è una caratteristica altamente desiderabile. La figura 3.1 prova a visualizzarne il motivo: le informazioni sui vari siti reticolari sono conservate in diversi nodi di calcolo, che sono dispositivi elettronici fisicamente distanti tra loro. La trasmissione di dati da uno all'altro (accesso remoto) impiega un tempo circa un ordine di grandezza più grande della lettura di un dato locale. Se le interazioni accoppiano siti a distanza maggiore di uno, la frazione di accessi remoti sale considerevolmente. Inoltre una interazione a grande raggio accoppierà realisticamente anche i siti posti in diagonale e ciò impone una forma di comunicazione tra processori cosiddetta non-primo vicino (figura 3.2).

3.4 Il modello SIMD

Un compromesso ragionevole per problemi di QCD sembra quindi essere quello di rispettare la struttura tridimensionale dello spazio che si vorrà simulare, organizzando una rete di nodi di calcolo disposti come una rete cubica e collegati (almeno) da una rete di sei collegamenti bidirezionali per nodo. Ognuno dei nodi sarà inoltre dotato di una sua memoria locale, nella quale saranno memorizzati i campi, i risultati ed ogni altra variabile necessaria nel corso del calcolo; ciascun nodo possiede ovviamente una unità dedicata al calcolo delle operazioni aritmetiche.

Secondo una classificazione tradizionale delle macchine parallele, l'architettura descritta, o “modello di esecuzione”, è conosciuta con il nome di SIMD. La sigla è la contrazione di Single Instruction Multiple Data; essa deriva dalla caratteristica di queste macchine di eseguire un *singolo* programma assegnato, facendolo eseguire contemporaneamente con più dati di ingresso (diversi).

Nell'architettura descritta la potenza di calcolo scala linearmente con il numero di processori, perché tutti sono attivi contemporaneamente. Questa è la sua prima

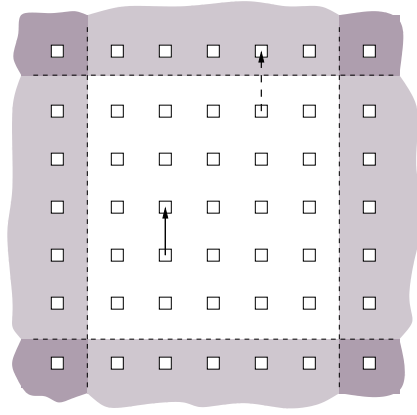


Figura 3.1: *Ripartizione dei campi tra processori*. Nella memoria locale di ogni processore è memorizzato lo stesso numero di siti di reticolo, disposti con la stessa geometria. Se, nel corso della spazzata, occorre utilizzare il valore di una variabile memorizzata all'interno dello stesso processore, si accederà all'opportuno indirizzo di memoria locale, calcolato a partire dalle coordinate del sito richiesto (freccia nera). Per aggiornare i siti posti al bordo occorrerà raggiungere la memoria di un processore primo vicino (freccia tratteggiata). I siti "appartenenti" alla memoria di processori primi vicini sono ombreggiati in chiaro, quelli non primi vicini in scuro. La figura è una sezione della macchina a $z = \text{costante}$. L'asse t dei campi è interamente locale e non è mostrato.

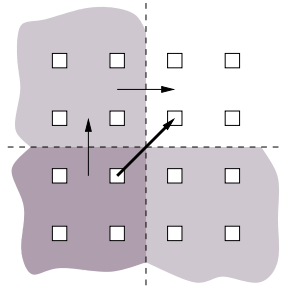


Figura 3.2: *Comunicazioni a secondo vicino*. Anche una interazione a primo vicino, se "obliqua" nella discretizzazione adottata, richiede di attraversare due volte la frontiera tra processori quando si sfrutta il modello di calcolo "implicitamente parallelo" descritto nel testo. (Questo vale quando sia il modello che la macchina abbiano una topologia bidimensionale.)

caratteristica desiderabile; la seconda è la sua semplicità, perché è relativamente semplice controllare un insieme magari largo di componenti ($\sim 10^3 - 10^4$ nel caso di apeNEXT) con l'assunzione che si comporteranno in maniera quasi omogenea.

La condivisione dei dati tra processori viene realizzata attivando quando occorre delle reti di trasmissione internodo. La tecnologia di collegamento tra i nodi è estremamente importante; basti ricordare che la comunicazione tra processori è necessaria, ad esempio, quando si valutano le grandezze delle interazioni (anche a primi vicini) sentite da siti “al bordo” della porzione di reticolo assegnata ad ogni processore. Per paragone, la nota tecnologia di trasmissione *ethernet* ha una larghezza di banda insufficiente di diversi ordini di grandezza alle tipiche applicazioni.

Nel capitolo successivo si mostrerà in un certo dettaglio come le richieste di calcolo della QCD abbiano imposto anche il dimensionamento del network internodo adottato in apeNEXT.

Un modello di calcolo parallelo alternativo a quello visto e molto noto è indicato con la sigla MIMD (Multiple Instruction Multiple Data); ne è dato un breve cenno in appendice. Il confronto tra diversi modelli di calcolo (MIMD, memoria condivisa, cluster ecc.) non è negli scopi di questo lavoro. Verrà invece fatto un rapido review delle macchine parallele costruite in tempi recenti per le necessità di calcolo numerico della fisica teorica.

3.5 Limiti del parallelismo implicito

Non per tutti i problemi è noto un metodo di soluzione numerica regolare. È sufficiente ad esempio considerare le operazioni necessarie per calcolare la trasformata di Fourier di un campo.

La trasformata di Fourier (normale oppure veloce) è una tecnica potente e praticamente onnipresente. La sua implementazione in un calcolatore massicciamente parallelo quali quelli dedicati alla QCD costituisce, però, una sfida notevole per il motivo che verrà brevemente esposto.

Supponiamo di voler calcolare la trasformata discreta di Fourier di un campo scalare bidimensionale (a valori reali o complessi) su un array di processori organizzati ad anello (figura 3.4). Il modo più efficiente per realizzare la trasformata discreta di Fourier in questa topologia sembra essere quello noto di scomporre la trasformata 2D in due trasformate 1D separate da una trasposizione; la strategia è nota in letteratura come “transpose algorithm” [14].

Un modo sensato di affrontare il problema è distribuire il campo in modo che un asse sia interamente locale ai processori: il valore del campo nel punto di coordinate i, j è memorizzato sul processore n , con $n \sim j$ ma che non dipende da i . La

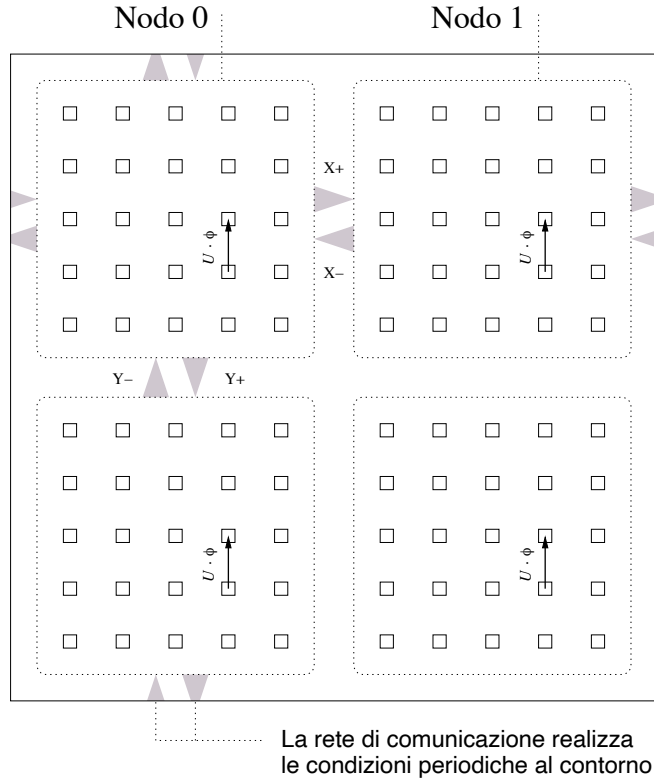


Figura 3.3: *Calcolo massicciamente parallelo*. Un grande numero di nodi di calcolo esegue un calcolo aritmetico (qui ho posto ad esempio il prodotto di una matrice di link con il valore del propagatore in un punto). Ciascun processore esegue lo stesso calcolo, in posizioni corrispondenti e lavora accedendo ai dati conservati nella sua memoria locale. Il campo “fisico” è suddiviso in parti uguali nelle memorie locali dei nodi di calcolo. La spazzata e l’aggiornamento del campo saranno completi quando ciascun processore avrà completato (sequenzialmente) il lavoro nella propria porzione di reticolo. Le spazzate avvengono di solito in maniera pressoché sincronizzata e finiscono approssimativamente allo stesso ciclo di clock. La rete di comunicazioni, mostrata in grigio per il nodo in alto a sinistra, consente di realizzare lo scambio bidirezionale delle informazioni tra processori adiacenti nelle tre dimensioni.

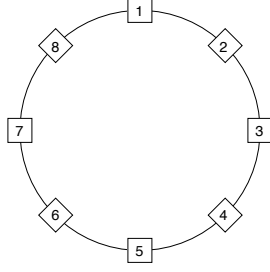


Figura 3.4: Array di otto celle, con topologia ad anello

trasformata lungo l'asse i non richiede comunicazioni locali e può essere eseguita efficientemente in parallelo su tutti i processori.

Per eseguire la trasformata lungo l'asse j occorre invertire i due assi, cioè eseguire una trasposizione del campo e questo richiede essenzialmente una comunicazione tutti-a-tutti, che non può essere parallelizzata in maniera banale. Sono state studiate, però, delle strategie che rendono il compito omogeneo tra i processori e quindi adatto all'architettura SIMD [15].

Sorge a questo punto un altro problema: le macchine per il calcolo scientifico hanno di solito una topologia a toro (almeno) tridimensionale e non ad anello. Occorre quindi numerare i processori del toro in sequenza e metterli in corrispondenza con quelli dell'anello. Come conseguenza, i processori che nell'anello sono primi vicini in una macchina con topologia 3D possono essere fisicamente lontani (e viceversa). La sequenza di comunicazioni necessarie per eseguire una trasformata di Fourier su una macchina SIMD è di conseguenza notevolmente complicata.

Il problema è stato esposto (in maniera semplificata) per mostrare un caso in cui l'architettura data-parallel *non* è quella naturale per la soluzione di un problema.

Data l'importanza del problema, l'implementazione efficiente della FFT su calcolatori massicciamente paralleli è stata oggetto di un certo numero di studi [16]. Parallelamente al lavoro su apeNEXT l'autore, assieme al Dr. Federico Toschi, ha condotto uno studio sulla implementazione della trasformata veloce di Fourier bidimensionale su un calcolatore parallelo dedicato (APEmille). I risultati ottenuti saranno esposti in un apposito lavoro.

3.6 Influenza della tecnologia

Nella costruzione di uno strumento dedicato al calcolo scientifico si pone l'accento sulla efficienza nella soluzione del problema assegnato. Nello stesso tempo si richiede che la costruzione dello strumento sia ragionevole in termini di tempo e risorse umane ed economiche necessarie nelle fasi di progetto, sviluppo e costruzione. En-

trambi questi vincoli a loro volta sono fortemente dipendenti dalle tecnologie attuali e dai rispettivi costi.

Per dare una idea di quanto sia articolato lo studio preliminare della architettura di una macchina dedicata con prestazioni allo stato dell'arte, dò un cenno di alcuni degli aspetti tecnologici che vengono valutati.

Tecnologia dei semiconduttori. Le operazioni in virgola mobile vengono eseguite da una unità sincrona – la *floating point unit* – che ditte specializzate (dette *foundries*) essenzialmente “stampano” in wafer di silicio. Le fonderie possiedono in generale tecnologie proprietarie che determinano la dimensione delle porte logiche, la tensione di alimentazione, l'area massima di un chip e quindi, in ultima analisi, la frequenza di lavoro della unità floating point. La tecnologia del silicio è in costante evoluzione. ApeNEXT verrà realizzato con una tecnologia che permette di realizzare transistor di lunghezza $0.18\ \mu\text{m}$. La larghezza di gate (λ) è un parametro chiave per identificare una tecnologia elettronica, in quanto la quantità di transistor che possono essere disposti per unità di area cresce come $1/\lambda^2$, mentre la velocità di commutazione di un transistor cresce come $1/\lambda$.

Tecnologia delle memorie. La quantità di memoria a stato solido che può essere integrata in un nodo di calcolo è finita ed ovviamente se ne vuole avere una quantità sufficiente ad affrontare il problema tipico. Delle memorie dinamiche vanno anche considerate la latenza ed il costo.

Tecnologia delle reti. In maniera analoga, il recente rinnovato interesse per le reti di comunicazione ha reso disponibili una quantità di tecnologie di trasmissione dati su cavo a velocità dell'ordine dei $500 \cdot 10^6$ bit/s, a regime. Lo sviluppo da zero di un link con queste prestazioni è già di per se una sfida tecnologia formidabile.

Tool di sintesi. Durante le fasi di progetto i progettisti danno dapprima una descrizione abbozzata delle parti logiche che comporranno la macchina. In seguito la descrizione di ciascuna parte viene raffinata e dettagliata sempre di più, fino a che un complesso software è in grado di trasformare il modello in una *netlist* che può essere spedita alla fonderia. È essenziale che la fase di sintesi sia eseguita da un tool affidabile, ovvero che non introduca artefatti nel tradurre le intenzioni dei progettisti.

Strumenti di simulazione dell'hardware. Ad ogni successiva modifica è essenziale sottoporre il modello ad un esteso test che verifichi che il comportamento

sia quello atteso, in più condizioni possibili (è impossibile testare ogni stato di una macchina digitale). Le simulazioni pre- e post- sintesi sono permesse dallo stesso software di cui al punto precedente. La sintesi di un modello deve essere verificabile e la simulazione deve essere fedele a quello che sarà il comportamento del chip quando sarà stampato (e non più modificabile).

Ambiente software. La disponibilità di linguaggi di programmazione che si prestano alla rapida scrittura di programmi di appoggio non è meno importante, ma è spesso trascurata. La necessità di automatizzare semplici operazioni ripetitive si fa sentire in ogni fase, dallo sviluppo (quando ad esempio si raccolgono i dati di uscita di un programma prelevandoli dai modelli delle memorie) alla produzione (quando ad esempio si vuole calcolare fuori linea una funzione di correlazione su un campo di velocità precedentemente calcolato e salvato). Di recente si sono resi disponibili sistemi operativi per PC affidabili e dotati di sorgente che hanno reso molto più facile lo sviluppo di una interfaccia efficace tra gli utenti finali e lo strumento.

Questo lavoro non approfondirà le interessanti questioni tecnologiche menzionate; si concentrerà invece su macchina e progetto visti in rapporto al problema che li motivano.

3.7 Calcolatori per la fisica

Il grafico in figura 3.5, presentato in [17], dà una interessante rappresentazione dell'attuale scenario delle macchine impiegate nella fisica computazionale. Il grafico è costruito a partire dai valori approssimati di prezzo e dalla performance di picco di nove macchine che sono state usate per simulazioni di QCD negli anni 1996-2002. (La performance reale di macchine dedicate può essere diverse anche di un fattore 2 – 4 rispetto a quella di picco e dipende dalla dimensione del problema da affrontare. Nel grafico sono ignorate le differenze di precisione numerica, dimensione della memoria e di efficienza).

L'ordinata di ogni punto è proporzionale alla potenza di calcolo del sistema considerato; la sua ascissa al suo costo per Mflops. Le linee diagonali rappresentano i sistemi di uguale costo.

La prima caratteristica evidente dalla figura è che sei sistemi appartengono alla fascia di costo che va da 1 a 10 milioni di dollari. Il motivo evidente è che questa è la cifra per cui un singolo progetto di ricerca scientifica può attendersi di essere finanziato.

Intorno alla ordinata dei 200–400 Gflops si trovano le macchine della generazione attuale di supercalcolatori per la fisica. La differenza di prezzo per flops si spiega

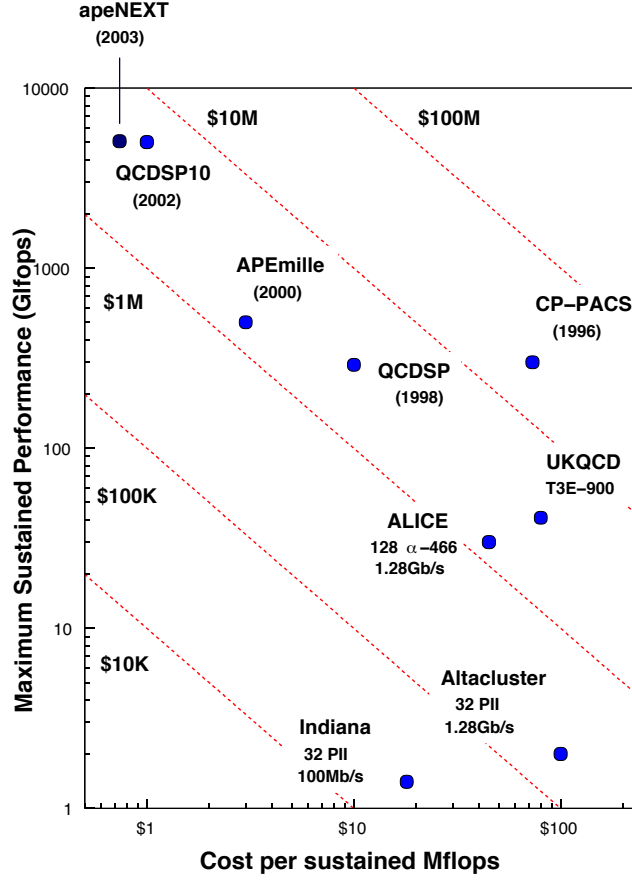


Figura 3.5: Prestazione di picco in funzione del prezzo per ciascun Mflops in alcuni sistemi usati per QCD su reticolo. Il grafico è quello dato in [17], con l'aggiunta del punto relativo ad apeNEXT. Le linee diagonali indicano il costo totale del sistema. Il QCDSP10 ha cambiato nome ed è ora noto come QCDOC.

in parte per le diverse date dei tre progetti. È evidente poi il notevole prezzo del CP-PACS, che ha messo a disposizione della ricerca giapponese una potenza paragonabile a quella attualmente impiegata in Europa ed USA, con un anticipo di circa tre anni, a costo superiore di un fattore di circa dieci.

La futura generazione, nella parte alta della figura, sarà caratterizzata da macchine dotate di potenze di picco dell'ordine di 5 Tflops per macchina; ci sono due progetti attualmente in fase di avanzato sviluppo: il progetto europeo apeNEXT, successore di APEmille, e QCDOC, sviluppato alla Columbia University dallo stesso gruppo che ha realizzato la QCDSP.

3.7.1 Macchine dedicate

È un dato di fatto che una frazione considerevole dei cicli macchina impiegati in simulazioni di QCD nel passato sia stata fornita da calcolatori di grandi prestazioni progettati e costruiti appositamente per affrontare il problema in esame.

In tempi recenti numerosi gruppi universitari si sono dedicati alla costruzione di questi calcolatori dedicati, cercando di sfruttare lo stato dell'arte delle tecnologie disponibili di memoria, calcolo e comunicazioni. La figura 3.6 mostra le potenze di picco che sono state raggiunte nei calcolatori dedicati costruiti in tempi recenti per applicazioni di LGT. La caratteristica evidente della figura è l'aumento esponenziale della potenza di picco; il fit dei punti indicati mostra questa potenza negli ultimi anni è andata raddoppiando ogni 15 mesi circa. L'incremento di prestazioni dei calcolatori dedicati è stato migliore di quello descritto dalla legge di Moore¹.

È interessante osservare che, secondo quanto mostrato nel capitolo precedente, le *richieste* di potenza di calcolo da parte dei fisici raddoppiano *più rapidamente* di quanto faccia la potenza dei calcolatori dedicati. In altre parole, i fisici usano tutta la potenza di calcolo disponibile anno per anno. Ipotizzo che l'aumento di operazioni per simulazione mostrato nella figura 2.1 sia dovuto ai tre seguenti fatti:

1. L'aumento della potenza dei calcolatori, secondo quanto visto nel paragrafo precedente.
2. La potenza resa disponibile dalla costruzione di macchine più grandi motiva pochi studi di grande scala piuttosto che molti di piccole dimensioni. La figura 2.1 mostra il numero di flop della *più grande* simulazione dell'anno; è possibile che negli anni la dimensione delle simulazioni sia cresciuta a scapito del loro numero.
3. Nei primi anni del calcolo i tempi delle simulazioni erano limitati più di adesso. Le attuali simulazioni allo stato dell'arte possono occupare macchine per mesi, mentre non era lo stesso in passato.

I punti 2. e 3. dovrebbero spiegare i diversi tempi caratteristici delle due curve. Le due curve sono state sovrapposte nella figura 3.7 che mostra, per così dire, la domanda e l'offerta di potenza di calcolo nello stesso grafico. (Si noti la diversa unità di misura per le due curve). Dal grafico si deduce che nel passato il tempo macchina concesso alle simulazioni di QCD su reticolo è aumentato e che la durata di una misura tipica, nel 1990, era nell'ordine di un mese.

3.7.2 Aggregati

I due punti più in basso nel grafico ed il sistema di nome ALiCE, sono rappresentanti di una categoria di macchine relativamente nuova: gli aggregati di calcolatori commerciali, o *cluster*. Un cluster è un insieme di PC o workstation collegati da una rete di comunicazione e supportati da qualche software che permetta lo scambio di dati tra programmi che contemporaneamente affrontano un singolo problema.

¹ La potenza di calcolo dei processori digitali programmabili raddoppia ogni 18 mesi.

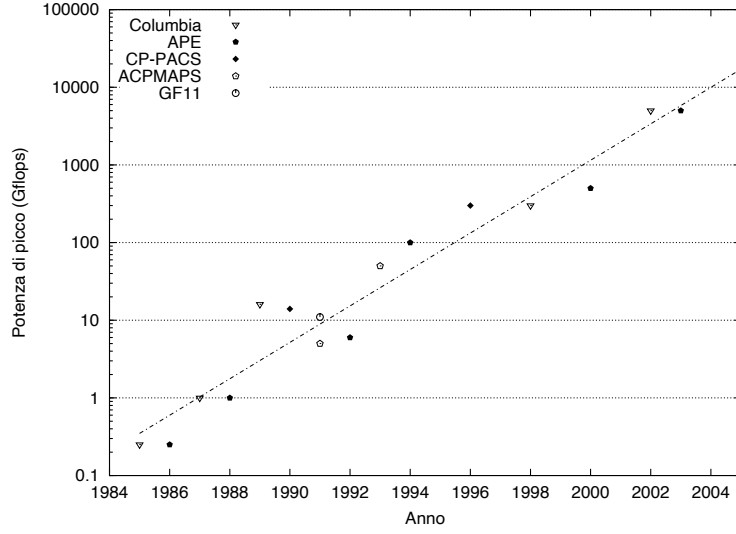


Figura 3.6: *Macchine per la QCD*. Il grafico mostra l'aumento negli anni della potenza di picco dei calcolatori dedicati alla QCD. I dati sono stati ottenuti dagli articoli [18] e [17]. Il fit con una legge esponenziale rivela che l'andamento è $\sim 2^{t/\tau}$, $\tau = 15.4 \pm 1.0$ mesi, più ripido di quello previsto dalla nota legge di Moore.

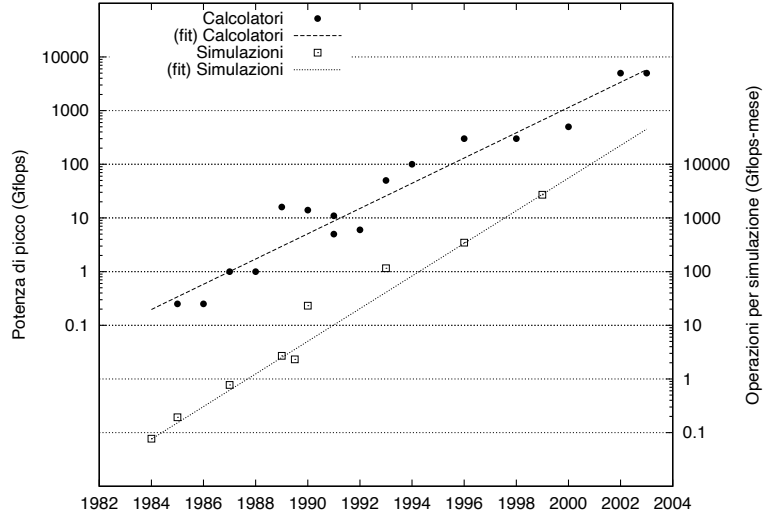


Figura 3.7: *Macchine e richieste di calcolo per QCD a confronto*. La figura riassume i dati forniti in questo capitolo e nel precedente. Si noti che le due curve sono misurate in diverse unità.

Per quanto non nuova, i cluster sono diventati una opzione per il supercalcolo solo di recente, essenzialmente a causa di due motivi. Il primo è che un cluster può essere realizzato in maniera relativamente semplice assemblando pezzi reperibili sul mercato di scala. Il passaggio dei dati tra processi può essere realizzato da software noti e documentati come MPI (una libreria per lo scambio di messaggi); la parte debole del progetto è che una rete di connessione sufficientemente bilanciata con il problema da affrontare² finisce per rappresentare una parte rilevante del costo delle macchine.

Il secondo vantaggio è che – entro dei limiti – è semplice aggiornare un cluster composto da hardware commerciale seguendo l'andamento del prezzo di mercato dello stesso: la potenza di calcolo dei processori commerciali raddoppia ancora ogni 18 mesi e potere avvantaggiarsi di questo andamento senza dover riprogettare l'intero sistema è una potenzialità sicuramente attraente.

D'altra parte, non è ancora stato realizzato alcun cluster in grado raggiungere potenze di picco anche dell'ordine dei 100 Gflops. Il motivo è probabilmente da un lato la difficoltà tecnica di aggregare $O(10^3)$ sistemi tecnicamente complicati e di grande consumo ($\gtrsim 50$ W per processore); dall'altro dal fatto che la necessità di connettere un così grande numero di nodi impone l'uso di un qualche tipo di switch dedicato che costituisce una frazione dominante del costo finale della macchina. Entrambi questi motivi non sono veri impedimenti tecnici, ma fanno aumentare il costo previsto per Mflops a livelli tali da rendere l'approccio Tflops-cluster assai meno attraente rispetto a cluster più piccoli ed allo sviluppo di un sistema *custom* di pari prestazioni.

3.7.3 Supercalcolatori commerciali

L'ultimo punto del grafico è una macchina Cray T3E-900 in dotazione al progetto inglese UKQCD. Questo sistema è interessante perché è l'unico supercalcolatore commerciale considerato: essi non sono molto popolari per ricerche di fisica.

I supercalcolatori commerciali sono progettati ponendo l'accento sulla generalità e sulla facilità di uso. Il primo requisito impone dei compromessi nell'architettura e nel software che in ultima analisi rendono il loro prezzo per flops molto maggiore di una equivalente macchina costruita per affrontare un solo problema ben preciso.

D'altra parte, la facilità di uso non è un requisito importante: una campagna di misure può impiegare una macchina per tempi dell'ordine dei mesi e non è assurdo pensare che la fase di preparazione dell'esperimento duri altrettanto; di conseguenza la semplicità di programmazione è un vantaggio, ma non è essenziale per macchine impiegate nelle simulazioni numeriche di grande scala.

² Il capitolo 4 affronterà la questione in dettaglio

Infine, queste macchine hanno avuto negli anni recenti incrementi di potenza inferiori rispetto sia a sistemi commerciali di fascia bassa, che a calcolatori dedicati.

La figura 3.8, tratta da [19], mostra le potenze di picco recentemente raggiunte da calcolatori commerciali fino al 2000. Le date si riferiscono all'anno di consegna o di costruzione. I simboli vuoti rappresentano supercalcolatori commerciali (indicati come vettoriali); i simboli piccoli pieni sono calcolatori paralleli commerciali; quelli grandi sono calcolatori dedicati alla QCD.

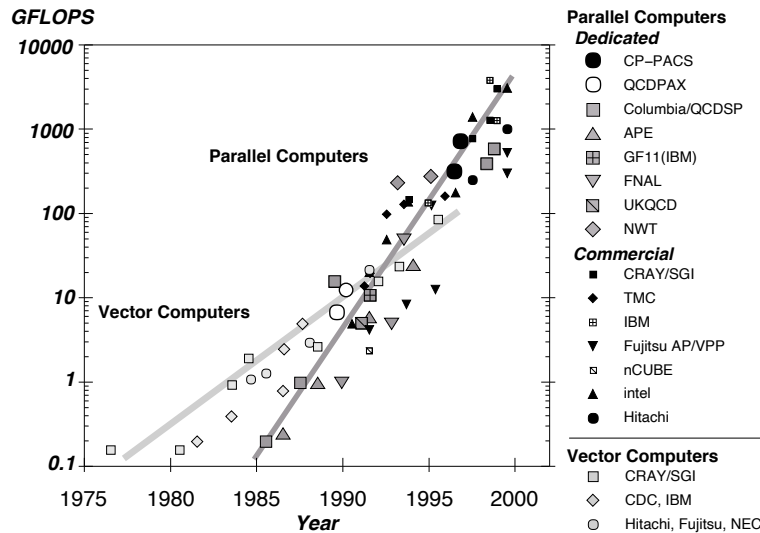


Figura 3.8: Potenze di picco raggiunte fino al 2000 da calcolatori commerciali e dedicati (da [19]).

3.8 Il panorama internazionale

È interessante notare che tutte le macchine dedicate progettate per essere applicate a problemi di fisica sono state costruite primariamente per calcoli di QCD su reticolo. Il motivo di questa scelta è in parte il grande interesse che c'è intorno al modello standard, che ha permesso di raccogliere abbastanza supporto da permettere la realizzazione di queste che sono spesso sfide tecnologiche ed organizzative di difficoltà non trascurabile; per un'altra parte conta la semplicità di principio del calcolo da effettuare.

Una eccezione interessante è stato il caso delle macchine Grape [20]. I calcolatori Grape (Gravity Pipe) sono stati progettati da un gruppo della università di Tokio per integrare numericamente le equazioni del moto di sistemi astrofisici ad N corpi, sottoposti alla interazione gravitazionale. Questo problema si presta alla costruzione di una macchina dedicata per motivi analoghi alla QCD: il modello teorico da simulare è consolidato; ed inoltre il problema è parallelizzabile in modo naturale.

La forma della forza gravitazionale tra una coppia di corpi è $F(\mathbf{x}_1, \mathbf{x}_2) \sim |\mathbf{x}_1 - \mathbf{x}_2|^{-2}$ e va valutata per $O(n^2)$ coppie di corpi ad ogni passo della integrazione delle equazioni del moto. Questo fa sì che in queste macchine l'efficiente calcolo dell'inverso della radice di un numero reale abbia l'importanza che in QCD hanno i prodotti matrice-vettore e l'aritmetica su grandezze complesse.

In questa sezione saranno brevemente descritte le architetture delle macchine dedicate attualmente in uso. Esse hanno fornito una frazione consistente dei flops dedicati alla ricerca sulla QCD a partire dal 1996 circa e continueranno probabilmente ad essere impiegate a tempo pieno per la produzione di risultati fino all'anno 2002. Quella che sarà descritta è talvolta indicata come la terza generazione di macchine dedicate.

3.8.1 Il CP-PACS

Questo progetto congiunto dell'università di Tsukuba e dell'industria Hitachi è partito in Giappone nel 1991 con lo scopo di sviluppare un calcolatore massicciamente parallelo per la ricerca nella scienza computazionale, con l'accento sulla QCD.

La macchina, nella sua configurazione definitiva che è stata completata nell'ottobre 1996, consiste di 2048 nodi di calcolo e 128 nodi di ingresso/uscita, collegati da una rete di comunicazione in una topologia tridimensionale $8 \times 17 \times 16$.

In ciascun nodo di calcolo è presente un processore dedicato, sviluppato a partire da un microprocessore commerciale HP PA-RISC1.1, che ha una potenza di picco di 300 Mflops [21] ed una memoria locale di 64 – 256 MB. Una particolarità interessante che è stata introdotta nel processore è la gestione *slide windowed* dei registri, adottata per nascondere la latenza degli accessi alla memoria: è possibile “precaricare” dati in registri con ampio anticipo sull'utilizzo degli stessi registri. Un memory controller mantiene traccia delle richieste di precaricamento e le onora nell'ordine. (Lo studio del preloading e del windowing sull'operatore di Dirac per apeNEXT è stato parte del lavoro svolto in questa tesi e verrà illustrato nei capitoli successivi).

La rete di comunicazione del CP-PACS consiste in dispositivi di collegamento del tipo detto *hyper crossbar*. Ciascun nodo può comunicare direttamente con quelli che nella macchina hanno uguale coordinata x oppure y oppure z . La comunicazione può essere disomogenea e la macchina è MIMD (su ogni nodo anzi è presente un sistema operativo locale).

La macchina raggiunge efficienze dell'ordine del 30 – 50% su un codice compilato e del 60% sullo stesso codice scritto in assembler³ ed ottimizzato a mano.

³ Un programma in forma assembler è composto da una sequenza di simboli (istruzioni) che viene interpretata dall'hardware. Una sequenza di istruzioni assembler che svolge un compito di qualche utilità è necessariamente composta da un numero molto grande di passi elementari.

3.8.2 La Columbia QC DSP

Lo sviluppo del QC DSP è partito presso l'università Columbia nel 1993. Lo scopo di questo progetto era costruire una macchina con il massimo rapporto potenza sostenuta per dollaro su applicazioni QCD e che potesse scalare fino ad una potenza complessiva dell'ordine dei Tflops [22].

Lo scopo è stato raggiunto unendo un grande numero di nodi di calcolo semplici e di basso consumo, realizzati usando processori commerciali prodotti dalla Texas Instruments. Il processore impiegato è un DSP (Digital Signal Processor) da cui il nome della macchina⁴ ed ha una potenza di picco di 50 Mflops su operandi a 32 bit.

Al processore (TMS320C31) è affiancato in ogni nodo un chip dedicato, che gestisce le comunicazioni remote e l'interfaccia con la memoria. La rete di comunicazione collega l'insieme dei processori in una griglia a quattro dimensioni; ogni processore può comunicare con i suoi primi vicini. Attualmente negli Stati Uniti sono installate tre macchine di questo tipo, che totalizzano una potenza di picco di circa 1 Tflops.

3.8.3 In europa: APEmille

APEmille è il progetto della collaborazione APE per l'attuale generazione di calcolatori dedicati. Il gruppo APE è nato circa 10 anni fa all'interno dell'Istituto Nazionale di Fisica Nucleare. In anni recenti al gruppo si sono aggiunte altre istituzioni di ricerca europee, DESY (Deutsches Elektronen-Synchrotron) e l'università di Paris-Sud. Le prime macchine APEmille sono state consegnate nel 2000; attualmente ci sono macchine installate nelle sedi INFN di Roma, Pisa, Roma II, per un totale di circa 1 Tflops, e DESY-Zeuthen (400 Gflops).

Una macchina APEmille è organizzata come una rete tridimensionale di nodi di calcolo; ciascun nodo consiste di un processore aritmetico e della sua memoria dati locale. Ciascun gruppo di otto nodi è controllato da un processore di controllo e lavora in modalità SIMD, con indirizzamento locale: i nodi possono accedere alle loro memorie ad indirizzi calcolati localmente.

I processori aritmetici possono accedere direttamente alle memorie dati dei nodi remoti per mezzo di una rete di comunicazione. Gli accessi remoti non hanno bisogno di un trattamento speciale da parte del programma. Un accesso alla memoria di indirizzo $r+l$, dove r è una costante stabilita in hardware, sarà automaticamente tradotto in un accesso all'indirizzo l della memoria di un nodo adiacente. Le comunicazioni nelle tre dimensioni della topologia della macchina, in entrambi i versi, si possono realizzare sommando all'indirizzo locale la opportuna costante $r_1 \dots r_6$. La

Comunemente queste sequenze sono generate in modo largamente automatico da software chiamati "compilatori" a partire da rappresentazioni equivalenti, ma più maneggevoli da esseri umani.

⁴ I DSP sono sviluppati commercialmente come dispositivi di basso costo in grado di fornire una accettabile capacità di calcolo ad apparecchiature elettroniche di vario tipo.

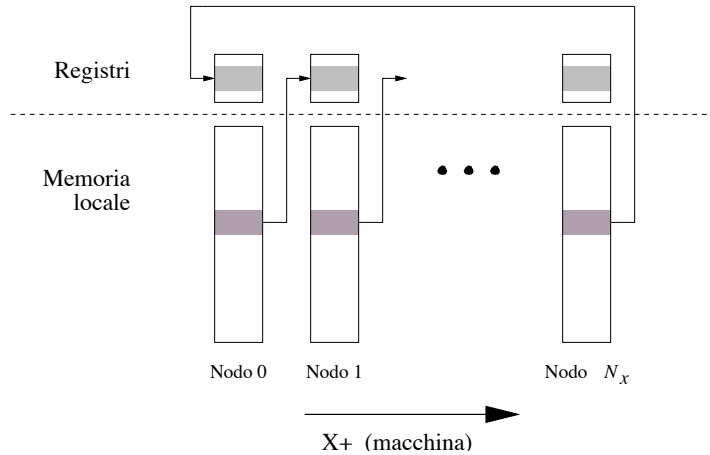


Figura 3.9: *Comunicazioni remote nel calcolo parallelo*. In APEmille il modello SIMD è rispettato anche nelle comunicazioni remote. Ogni processore copia una parte della memoria, locale o remota, nel suo register file per potere successivamente eseguire operazioni aritmetiche sugli operandi contenuti in essa. L'accesso può essere indifferentemente locale o remoto, purchè segua il pattern regolare mostrato in figura: ciascun nodo comunica (nel caso più semplice) con il suo primo vicino nella direzione X^- per prelevare i dati richiesti.

rete di comunicazione è chiusa in forma di 3-toro in modo da ottenere le condizioni periodiche al contorno sul reticolo fisico.

La macchina è rigidamente sincronizzata: i nodi di calcolo di una partizione sono allineati temporalmente a meno di una frazione del ciclo di clock, che è 66 MHz. Ad ogni ciclo di clock il processore può calcolare una *operazione normale* $a \cdot b + c$ su grandezze complesse contenute in registri (ce ne sono 512). La potenza di calcolo di picco su numeri complessi è quindi di 528 Mflops (132 Mflops su numeri reali in singola precisione e 66 Mflops in doppia precisione).

Gli accessi alle memorie remote avvengano in maniera parallela e corrispondente (figura 3.9). Dal momento che il sincronismo deve essere conservato e che gli accessi remoti impiegano più tempo di quelli locali, durante un accesso remoto il resto della macchina inclusa l'unità floating point viene interamente congelato. Di conseguenza gli accessi remoti sono causa di una perdita di efficienza; ogni accesso remoto dura circa tre volte il tempo di un accesso locale (45 ns contro 15 ns circa per parola reale, latenze escluse).

La macchina raggiunge efficienze dell'ordine del 50% su programmi tipici scritti in un linguaggio chiamato TAO e noto perché impiegato nelle precedenti macchine realizzate dal gruppo.

3.9 La prossima generazione

3.9.1 Columbia University: QCDOC

L'università della Columbia ha iniziato lo sviluppo di una macchina che dovrebbe raggiungere la scala di potenze dell'ordine dei 10 teraflops [23]. L'obiettivo del progetto è di avere macchine funzionanti intorno all'anno 2002 con un rapporto potenza/prezzo di circa un dollaro per Mflops, circa un decimo della macchina precedente realizzata dallo stesso gruppo.

Anche la QCDOC adotta la strategia di unire un grande numero di processori di piccola potenza e di basso consumo; la novità rimarchevole è che il nodo di calcolo e tutte le componenti accessorie verranno impresse in un singolo chip, che dà il nome alla macchina. La soluzione è stata resa possibile dalla collaborazione con la IBM che ha fornito numerose componenti già sviluppate (*standard libraries*), da cui provengono la maggior parte delle componenti del nodo di calcolo.

L'unità aritmetica è un PowerPC dotato di una unità in virgola mobile da 1 Gflops; ogni nodo ha 24 porte di comunicazione con una larghezza di banda di 500 Mbit/s ciascuna. Il processore include 32 Mbit di memoria su chip, che sono sufficienti per le simulazioni full-QCD. La velocità di lettura della memoria prevista è di 64 Gbit/s.

L'università ha sviluppato in proprio i blocchi di collegamento (controller) tra processore, memoria e link. La topologia complessiva dei link della macchina è quella di toro a 6 dimensioni. Questa forma in qualche misura insolita è stata scelta per poter partizionare la macchina in unità più piccole di diverse grandezze, ciascuna di esse ancora a forma di toro con quattro dimensioni. La macchina complessiva può essere suddivisa in forme toroidali diverse, facendo corrispondere gli assi fisici $\hat{x}_0 \dots \hat{x}_3$ ad opportuni indici di nodo, senza la necessità di ricollegare i link seriali per dividere realmente la macchina in più parti, come invece era richiesto nella QCDSF.

La macchina completa sarà prevedibilmente composta da $2^3 \times 8^2 \times 16 = 8192$ nodi, per una potenza di picco di 8 Tflops. La efficienza attesa su codici di QCD ottimizzati è del 50%.

3.9.2 apeNEXT

L'obiettivo principale per la costruzione di apeNEXT è quello di rendere disponibile nei prossimi 2 – 4 anni risorse di calcolo sufficienti per simulazioni full QCD su reticoli di dimensioni dell'ordine di $24^3 \times 48$. Questo scopo corrisponde al raggiungimento di una potenza di picco dell'ordine di 3 – 6 Tflops in doppia precisione e quantità di memoria dell'ordine di 10 Tbit. Il progetto mira a raggiungere una

potenza integrata tra i membri della collaborazione di 30 Tflops nell'anno 2003, ad un costo compreso tra 0.5 ed 1 euro per Mflops.

L'architettura di apeNEXT, su cui si è basato il lavoro del candidato, è descritta in dettaglio nel capitolo successivo.

Capitolo 4

L'operatore di Dirac e l'architettura di apeNEXT

Le prestazioni della macchina saranno essenzialmente determinate dalla sua efficienza nel calcolo del cosiddetto operatore di Dirac della teoria discretizzata sul reticolo; la forma dell'operatore determina la topologia delle connessioni internodo, la larghezza di banda delle stesse relativa alla potenza di calcolo ed altri parametri. Il quarto capitolo introduce le scelte di progetto elencate.

Nei capitoli precedenti è stato mostrato che lo scopo di una simulazione di LGT è la misura di una osservabile sullo stato di vuoto e che questa sia computazionalmente dominata dal calcolo del propagatore di uno o più quark. Il propagatore è calcolato invertendo l'operatore $\mathcal{D} + m_q$; l'inversione avviene applicando ripetutamente l'operatore stesso ad una configurazione di prova, che viene fatta convergere alla soluzione cercata. La forma dell'operatore di Dirac sul reticolo che è stata assunta in questo lavoro è quella data in [21] (K è una costante che dipende dal passo del reticolo e dalla massa del quark; gli indici di spin e colore non sono indicati):

$$D_{n,n'} = \delta_{n,n'} - K \sum_{\mu} \left\{ (1 - \gamma_{\mu}) U_{n\mu} \delta_{n+\hat{\mu},n'} + (1 + \gamma_{\mu}) U_{n'\mu}^+ \delta_{n-\hat{\mu},n'} \right\} \quad (4.1)$$

Il calcolo dell'operatore di Dirac costituisce la parte di gran lunga più consistente nel conteggio del numero totale di operazioni aritmetiche della tipica simulazione sia quenched che full-QCD. Le macchine dedicate alla QCD sono costruite in modo che l'applicazione dell'operatore ad un campo dato avvenga con la massima efficienza. In questo capitolo sarà discusso, in particolare, come queste considerazioni hanno motivato le scelte architetturali di apeNEXT.

Gli altri algoritmi che compongono una simulazione, accennati in precedenza (conjugate gradient, stima dello scarto della soluzione, bagno termico alle matrici di gauge) contribuiscono generalmente ad una frazione sufficientemente esigua del

tempo di calcolo totale da giustificare la decisione di non prenderli in considerazione nella stima dell'efficienza. La collaborazione CP-PACS riporta le percentuali date in tabella 4.1 per una simulazione quenched [24]. Si stima che in una simulazione full-QCD la frazione di tempo impiegata nel solver di Dirac sia approssimativamente doppia.

Passo	Frazione (%)
Update	30
Gauge-fixing	8
<i>Solver</i>	43
Measurement	19

Tabella 4.1: *Breakdown dei tempi di calcolo.*

4.1 L'operatore di Dirac sul reticolo

La matrice $D_{n,n'}$ ha dimensione $12N \times 12N$, con N pari al numero di siti. Essa, però, è estremamente sparsa, perché ha elementi non nulli solo tra i siti primi vicini nelle quattro dimensioni. Per questo il prodotto con un vettore dato viene eseguito in parallelo, spezzandolo su più processori, ciascuno dei quali percorre la sua parte di vettore ed applica ad essa, elemento per elemento, il prodotto per gli elementi di matrice non nulli corrispondenti.

Il calcolo dell'operatore di Dirac viene quindi eseguito con il metodo diretto (da cui il termine “algoritmo”) descritto nel riquadro 4.1. (La sequenza di operazioni descritta si intende estesa a quattro dimensioni.) Questa sequenza è estremamente importante per le discussioni che seguiranno in questo lavoro; d'ora in avanti per brevità verrà indicata con il simbolo \mathcal{D} .

Supponiamo di avere un quadrispazio discretizzato in N punti reticolari. L'applicazione dell'operatore di Dirac si riduce essenzialmente all'applicazione dell'algoritmo dato per ognuno degli N siti. Dal momento che \mathcal{D} si applica in maniera identica a ciascun sito, diciamo che per applicare l'operatore dobbiamo compiere N *iterazioni* oppure *loop* della procedura \mathcal{D} (le coordinate del sito centrale sono parametri che variano ad ogni iterazione).

L'aggiornamento del campo è completo quando il calcolo mostrato è stato eseguito in successione in ognuno dei punti del reticolo. Ai siti di bordo si impongono le condizioni periodiche al contorno. Quando l'aggiornamento è finito, si dice che è stata completata una *spazzata* del campo.

In figura 4.2 è mostrato graficamente il pattern di accesso ai dati necessario per il calcolo dell'operatore in un sito, nel caso di un campo bidimensionale (il passo del reticolo è assunto pari ad 1). Ogni quadrato rappresenta lo spinore associato a ciascun sito del reticolo, quindi 3×4 variabili complesse. Ogni freccia che congiunge

L’algoritmo \mathcal{D} .

Note: Indico con T uno spinore temporaneo (variabile di appoggio per semplificare la notazione) e con $\phi(x, y)$ il valore dello spinore nel sito di coordinate intere (x, y) . Trascuro eventuali fattori numerici. Le U sono le matrici di gauge $SU(3)$ associate ai link numerati come in figura 4.2. I passi si intendono eseguiti in maniera sequenziale ed estesi nel modo naturale a quattro dimensioni. Il punto indica il consueto prodotto tra matrici.

1. $T_1 = (1 - \gamma_1) \cdot U_1 \cdot \phi(x + 1, y)$
2. $T_2 = T_1 + (1 + \gamma_1) \cdot U_2 \cdot \phi(x - 1, y)$
3. $T_3 = T_2 + (1 - \gamma_2) \cdot U_3 \cdot \phi(x, y + 1)$
4. $T_4 = T_3 + (1 + \gamma_2) \cdot U_4 \cdot \phi(x, y - 1)$
5. – 8. \dots
9. $T_9 = T_8 + \phi(x, y)$
10. memorizza il risultato: $\phi'(x, y) = T_9$

Riquadro 4.1: Un passo della procedura che applica l’operatore di Dirac sul reticolo.

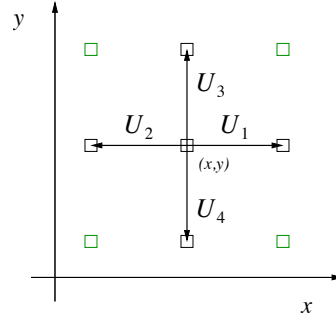


Figura 4.2: Pattern di accesso ai dati in \mathcal{D} (nel piano xy).

due quadrati rappresenta una matrice $SU(3)$, quindi 9 grandezze complesse. Il pattern mostrato in figura è talvolta indicato con il nome di “croce”.

4.2 Il dimensionamento della macchina dall’analisi di D

Nelle sezioni successive saranno riviste in maniera abbastanza generica le considerazioni che hanno portato alle scelte architetturali della macchina, con particolare attenzione a come queste sono collegate al problema che si è deciso di ottimizzare.

4.2.1 Calcolo di R

È evidente innanzitutto che il calcolo ha la forma tipica di una derivazione numerica. A parte il sito centrale, ogni iterazione di \mathcal{D} utilizza i valori di ϕ negli otto siti primi vicini di quello che si sta calcolando ed i valori di altrettante matrici U , associate a ciascuno dei bracci della (iper) croce. Nell’attuale tecnologia, i valori di ciascun

elemento delle grandezze dette vengono conservati in unità dette “memorie”; questo fatto motiva le considerazioni che seguono.

In questa sezione ed in quelle successive ricaveremo i parametri R e ρ dalla forma dell’operatore. Essi dimensionano (in ultima analisi) la memoria e rete di comunicazione. Come si è detto, in questo lavoro non analizzerò invece l’influenza dei fattori tecnologici sull’architettura di apeNEXT; essi sono debitamente descritti nella referenza [25].

Il parametro R è definito dal rapporto

$$R = \frac{\text{operazioni aritmetiche}}{\text{operandi richiesti}}$$

ed è una caratteristica del problema da risolvere. Il calcolo dell’operatore di Dirac come descritto in precedenza è realizzato, a meno del termine centrale, per mezzo di una sequenza di otto operazioni $\phi_i^{'s} = \gamma_{sr} U_{ij} \phi_j^r$, una per ogni braccio della croce. Gli indici i e j variano sui colori $1 \dots 3$, s ed r variano sulle componenti dello spinore $0 \dots 3$.

Nel calcolo del numero di operazioni aritmetiche richieste per braccio ignoro i prodotti per le matrici γ , poiché sono in prima approssimazione equivalenti ad un riordino degli indici di spin eventualmente moltiplicati per $\pm 1, \pm i$. L’ordinario prodotto riga per colonna $U_{ij} \phi_j$ richiede tre moltiplicazioni e due somme tra grandezze complesse. Noi siamo interessati a tutte e tre le componenti del prodotto ($i = 1 \dots 3$) ed a tutte quattro le componenti dello spin ($s = 0 \dots 3$), quindi il numero complessivo di operazioni aritmetiche complesse richieste per ciascun braccio è di $3 \cdot 3 \cdot 4 = 36$ moltiplicazioni e $2 \cdot 3 \cdot 4 = 24$ addizioni complesse; vale a dire un totale di $36 \cdot 6 + 24 \cdot 2 = 264$ operazioni reali. I dati in ingresso sono i nove elementi complessi di U ed i 12 di ϕ , per un totale di 42 operandi reali. Da ciò si calcola che per l’operatore di Dirac vale

$$R \simeq \frac{264}{42} \simeq 6.3$$

dove il simbolo di “circa” ci ricorda che abbiamo trascurato il punto centrale, i prodotti per le matrici γ , l’eventuale coniugazione degli U e la somma/differenza tra i risultati di ogni braccio. È possibile raffinare il calcolo, ma ciò non è particolarmente interessante in questa sede.

Il valore di R può essere anche determinato in modo diretto analizzando i codici di calcolo in modo da contarvi il numero di operazioni e di accessi in memoria. Vedremo in particolare che il codice che calcola l’operatore di Dirac realizzato dall’autore contiene nel loop interno 342 operazioni normali complesse, carica dalla memoria 182 operandi complessi e ne memorizza 12 di risultato. Da questi valori si calcola $R = (342 \cdot 8)/(194 \cdot 2) \simeq 7.05$. La discrepanza rispetto al valore stimato in

precedenza si spiega in parte per le approssimazioni adottate ed in parte perché nel codice si deve eseguire una operazione normale complessa (che nella stima precedente contava come 8 operazioni aritmetiche) anche quando ne sarebbe sufficiente una più semplice, quale una somma (che contava come due operazioni). Il motivo di questa limitazione verrà esposto tra breve.

Per inciso, il valore di R serve anche a classificare approssimativamente i problemi numerici in una scala convenzionale. Se la soluzione richiede molti calcoli su pochi dati di ingresso ovvero, equivalentemente, se un insieme di dati di ingresso fornisce operandi ad un numero molto maggiore di operazioni aritmetiche, il problema è indicato come *number crunching*. Le applicazioni di QCD, in cui $4 < R < 8$, sono classiche rappresentanti di questa categoria. Nell'ordinamento opposto di R piccolo si adotta la denominazione *data intensive*. Le applicazioni che hanno carattere statistico appartengono tendenzialmente a questa seconda categoria; il calcolo della media di un largo set di valori, ad esempio, ha $R = 1$.

Le componenti di una matrice $SU(3)$ non sono tutte indipendenti, quindi sarebbero sufficienti in linea di principio 5 valori complessi per rappresentarla. È stato provato nella pratica che se si sceglie di mantenere memorizzate solo (ad esempio) le componenti sopra diagonali, ne deriva una complicazione nei calcoli successivi che non giustifica il risparmio di memoria.

4.2.2 Calcolo di ρ

Un altro parametro tipico di notevole interesse per i nostri scopi è ρ , definito come il rapporto

$$\rho = \frac{\text{operandi remoti}}{\text{operandi usati}}$$

La quantità al numeratore dipende dalla particolare ripartizione dei campi tra i processori che si sta considerando. Essa è il numero di dati in ingresso che *non sono* disponibili nella memoria locale del processore che eseguirà l'operazione e devono pertanto essere trasmessi dalla rete internodo.

In questa sezione calcoliamo il valore di ρ per lo stesso algoritmo dato ed adottiamo nel calcolo le stesse approssimazioni usate per R .

Dimostriamo che il valore di ρ dipende linearmente da n , la dimensione *spaziale* del reticolo assegnato ad ogni processore. Assumiamo realisticamente di simulare uno spazio fisico in quattro dimensioni con la forma di cubo di lato $L = 4$ fm, discretizzato con passo reticolare $a = 0.08$ fm; questo corrisponde ad un reticolo ipercubico con $N = 4/0.08$ siti di lato su ciascun asse. Analizziamo la ripartizione del reticolo su una macchina dotata di P processori.

La topologia della macchina apeNEXT è tridimensionale e la assumiamo in prima approssimazione dotata di un uguale numero di processori in ogni direzione

(cubica). Poiché molte macchine APE sono composte di multipli di 8 processori, supponiamo di avere un reticolo di 48^4 punti su una macchina di $8 \times 8 \times 8$ nodi. La topologia tridimensionale suggerisce che uno dei quattro assi dello spazio fisico sia tenuto interamente locale ad ogni nodo; convenzionalmente si sceglie l'asse temporale. Il reticolo assegnato ad ogni processore sarà quindi un (iper) parallelepipedo con i tre assi spaziali lunghi $n = 48/8 = 6$ e quello temporale lungo $N = 48$ siti.

Consideriamo il numero di operandi richiesti dall'applicazione dell'algoritmo non ad ogni sito dell'intero quadrispazio, ma solo ad una sua sezione a $t = \text{costante}$, che chiamerò per semplicità “fetta”. (Ciascuno di questi sottospazi richiederà lo stesso numero di operandi e di operandi remoti; a noi interessa solo il rapporto tra queste due grandezze, quindi il numero di fette – ovvero la dimensione del quadrispazio nella direzione t – non entra nel calcolo di ρ .) Una fetta contiene $n^3 \simeq 216$ siti reticolari; una spazzata che la percorre utilizzerà quindi circa $8 \cdot 42 n^3$ operandi reali. ⁽¹⁾ (Il fattore 8 tiene conto del numero di braccia per sito.) Aggiungiamo a questi ancora altri $24 n^3$ operandi, dovuti al termine di massa, prima trascurato; il numero di grandezze reali utilizzate in una spazzata del sottospazio è in definitiva circa $360 n^3$.

Lo schema in figura 4.3 aiuta a visualizzare il fatto che non tutti gli operandi sono memorizzati nello stesso nodo che sta effettuando il calcolo: il sottospazio locale confina con $6 n^2$ siti primi vicini posti su processori remoti. Da ciascuno di questi siti è necessario prelevare un valore di ϕ (24 grandezze reali); in base a quanto mostrato nella figura 4.4 sarà anche necessario prelevare i valori delle matrici U da $3 n^2$ di essi. Di conseguenza in una spazzata del sottospazio si prelevano

$$24 \cdot 6 \cdot n^2 + 18 \cdot 3 \cdot n^2 = 198 n^2 \simeq 6912$$

operandi remoti. Si ottiene così che

$$\rho = \frac{360 n^3}{198 n^2} \simeq 1.8 n \simeq 10.8 \quad (4.2)$$

Calcoliamo ora il valore di ρ per una macchina i cui nodi siano disposti *secondo una topologia ad ipercubo*. In questo caso la forma del reticolo locale sarà anche esso un ipercubo di lato $n \times n^3$ e tutte le sue otto facce ⁽²⁾ sono affacciate a nodi vicini. Il numero di operandi remoti richiesti sarà perciò $8 \times n^3 \times 24 + 4 \times n^3 \times 18 = 264 n^3$.

Si ottiene quindi che

¹Notare qui che sto implicitamente conteggiando gli ϕ di ciascun sito per più di una volta, come se questi fossero “dimenticati” tra una iterazione e l'altra di \mathcal{D} . Questa assunzione è approssimativamente quello che avviene in realtà. Nell'unità aritmetica non c'è un numero sufficiente di locazioni temporanee (registri) per mantenere i valori già utilizzati, che quindi devono essere ricaricati dalla memoria per ogni passo della spazzata.

² Ogni faccia è un 3-cubo di lato n .

Da ogni faccia escono n^2 segmenti

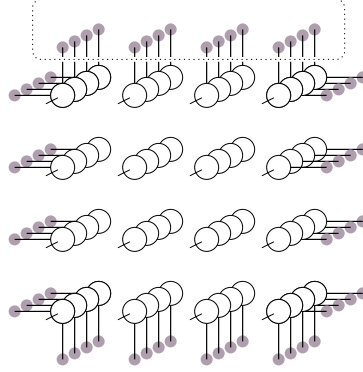


Figura 4.3: Schema per il calcolo del numero di operandi remoti che compaiono al denominatore di ρ . La figura mostra il caso $n = 4$. Il numero di dati remoti richiesti è proporzionale al numero di segmenti uscenti da ogni faccia del reticolo tridimensionale incluso in un nodo, come quello mostrato in figura ($t = \text{costante}$).

$$\rho_{4D} = \frac{360 n^4}{264 n^3} \simeq 1.36 n$$

dove al numeratore si è scritto il numero di operandi richiesti per la spazzata dell'intero ipercubo. A parità di reticolo locale, quindi, una macchina con una topologia a quattro dimensioni ha bisogno di una banda di trasmissione remota maggiore rispetto ad una a tre dimensioni.

Il valore di ρ svolge nei confronti della rete lo stesso ruolo che R aveva per la memoria. Assumiamo in una approssimazione molto rozza che un operando ogni ρ debba passare attraverso la rete; allora la velocità della rete (larghezza di banda w) può essere $1/\rho$ volte quella della memoria. Se la larghezza di banda fosse molto inferiore, la rete in condizioni stazionarie sarebbe causa di un ritardo minimo di

$$\frac{w}{\rho} - 1$$

operandi per ciclo. In questo calcolo abbiamo del tutto trascurato la latenza della rete ed inoltre abbiamo assunto che la trasmissione dei dati possa avvenire in contemporanea ai calcoli ed agli accessi ad altri operandi locali.

Queste sono assunzioni molto forti e l'unico modo per valutarne l'impatto è eseguire delle misure sulla simulazione di un vero codice in un modello dettagliato della macchina. I valori di R e di ρ calcolati sono confrontabili con le stime date in [26].

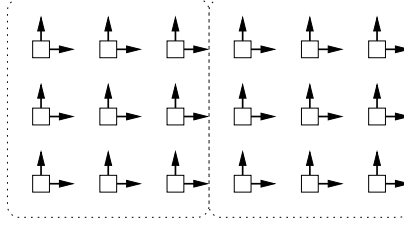


Figura 4.4: *Distribuzione delle variabili di gauge*. Ogni matrice di gauge è convenzionalmente memorizzata nella memoria dello stesso processore che contiene anche il sito reticolare da cui essa è *uscente*.

4.3 La memoria

La memoria è un dispositivo che può accedere ad un dato precedentemente memorizzato, identificato in maniera unica da una chiave detta “indirizzo”. L’accesso (lettura) o la scrittura del dato avvengono con un ritardo variabile (noto come *latenza*). Il numero di indirizzi utilizzabili è la “capacità di memoria” ed è misurata in parole.

Nel caso di apeNEXT, ad ogni indirizzo viene associata una parola di 128 bit, che è l’unità base dei calcoli aritmetici. Ciascuna parola viene interpretata come una coppia di valori interi oppure reali. I valori interi sono codificati nella notazione binaria a 64 bit con segno; la codifica delle grandezze floating point è accennata in appendice a pagina 99.

Le due scelte fondamentali da adottare a proposito della memoria di cui dotare una macchina dedicata come apeNEXT sono la sua dimensione e la sua velocità.

Il primo valore è essenzialmente proporzionale al numero grandezze numeriche alle quali la macchina potrà avere accesso in maniera veloce durante la fase di esecuzione dei programmi. I più recenti algoritmi per la full QCD richiedono l’uso di un numero di dati temporanei assai elevato; si è scelto di dotare i nodi di calcolo di una quantità di memoria sufficiente a tenere memorizzati $O(200n^4)$ feq⁽³⁾. Questa quantità potrebbe essere insufficiente per alcune simulazioni unquenched, in cui il numero di siti è notevolmente maggiore del caso full-QCD ($n \simeq 100$); in questi casi si può comunque ricorrere al supporto di memorie esterne più lente (“swap”).

La velocità di una memoria dinamica è limitata dalla tecnologia con è realizzata. Una moderna memoria può fornire il dato richiesto come indirizzo solo dopo un certo ritardo (variabile), noto come *latenza*. Le parole successive vengono invece fornite al ritmo di una per ciclo di clock. Il clock di apeNEXT ha un periodo di 5 ns; di conseguenza, trascurando la latenza iniziale, la memoria ha una banda passante di 25.6 Gbit/s ovvero $400 \cdot 10^6$ valori in virgola mobile per secondo.

³ Un feq (fermione equivalente) è la quantità di memoria necessaria per conservare 24 grandezze reali.

Vedremo tra breve anche che l'unità aritmetica ha una potenza di 1.6 Gflops. La banda passante della memoria è 1/4 della potenza di calcolo. Questo vuole dire che se tentassimo di eseguire un codice con il parametro $R < 4$, l'unità aritmetica sarebbe sottoutilizzata perché la memoria non sarebbe in grado di fornire operandi al ritmo necessario per tenerla occupata (memory contention).

4.4 Il register file

Una parte importante dell'architettura di apeNEXT è il *register file*. L'unità aritmetica non preleva i dati direttamente dalla memoria, perché tipicamente l'ordine sparso in cui si accede agli elementi di matrice nel prodotto riga per colonna renderebbe la loro lettura molto inefficiente. Gli operandi devono essere portati in un certo numero di aree temporanee note come registri; ciascun registro ha la stessa dimensione di una locazione di memoria e quindi contiene una grandezza complessa (oppure due reali) in doppia precisione.

Le scritture e le letture dei valori contenuti nel register file possono essere eseguite senza cicli di latenza, perché esso è realizzato con una tecnologia diversa dalle memorie dinamiche nello stesso pezzo di silicio dell'unità aritmetica. Nel processore apeNEXT sono presenti 256 registri a 128 bit. Ciascun registro può contenere, allo stesso modo di ogni parola in memoria, una grandezza complessa, oppure due grandezze reali indipendenti (modalità vettoriale), oppure due numeri interi. In ogni caso la precisione è di 64 bit.

I registri devono poter contenere valori interi per esempio per tenere traccia degli indici (di spincolor o di sito) ed eseguire i calcoli di indirizzo. I primi 64 registri sono collegati alla AGU, che sarà descritta in una sezione successiva.

Una caratteristica peculiare di apeNEXT è la possibilità di *mascherare* le scritture sul register file (ed in memoria): il programmatore può richiedere che le operazioni di scrittura di risultati vengano inibiti a seconda della verità di una condizione locale. Questo meccanismo è noto come *where()* e deve il suo nome all'istruzione che lo realizza nel linguaggio dedicato TAO.

Il codice che calcola una iterazione di \mathcal{D} si compone molto schematicamente dei passi: copia gli U e ϕ dalla memoria nei registri (load) \rightarrow esegui le operazioni aritmetiche tra registri caricati \rightarrow copia i risultati di nuovo in memoria all'indirizzo che contiene i risultati (store) \rightarrow passa al sito successivo. Il contenuto dei registri viene perso tra una iterazione e l'altra. Questa impostazione è comune a quasi tutti i codici di calcolo per la fisica.

4.5 Il processore matematico

È interessante notare che somme e prodotti complessi nel calcolo del prodotto ma-

triale $U \cdot \phi$ siano approssimativamente bilanciati, sia in numero che nella sequenza (nel prodotto riga per colonna si esegue prima la moltiplicazione, poi si accumula il risultato con i precedenti). Questo suggerisce di definire una cosiddetta “operazione normale” $n(a, b, c) = a \cdot b + c$. L’unità in virgola mobile di apeNEXT esegue in hardware il calcolo di operazioni normali su grandezze complesse⁴ per ogni ciclo di clock, che corrisponde ad una potenza di calcolo picco di 1.6 Gflops. Somme e prodotti tra grandezze in virgola mobile vengono in ogni caso ricondotti ad operazioni normali ponendo rispettivamente $a = 1$ oppure $c = 0$.

L’unità esegue le operazioni normali leggendo ciascun operando a, b, c da un registro (arbitrario) e memorizza il risultato n in un quarto registro, anche esso arbitrario. Ciascun operando può anche essere negato oppure sottoposto a coniugazione e questa operazione non richiede tempo aggiuntivo (si dice che si può eseguire una “coniugazione al volo”). Questo motivo chiarisce perché nel precedente calcolo di R le operazioni $z \rightarrow -z$ e $z \rightarrow z^*$ non siano state contate come operazioni aritmetiche.

L’unità in virgola mobile esegue una operazione normale complessa (corrispondente ad 8 operazioni reali) ogni ciclo di clock (5 ns), quindi ha una potenza di picco di 1.6 Gflops. I risultati delle operazioni sono disponibili con una latenza di 10 cicli, nota come *lunghezza di pipe*. Il nome deriva dal fatto che il calcolo viene scomposto nell’hardware nei suoi passi elementari, ciascuno dei quali viene completato in un ciclo di clock. Le varie sottounità che realizzano i passi elementari sono indipendenti, quindi se si ha un lungo vettore di ingresso ogni elemento potrà essere fornito alla unità floating point (in un registro) senza aspettare l’uscita dei precedenti. L’esistenza di una pipe ha importanti conseguenze per la efficienza dei programmi. Alla fine di questo capitolo verrà dato un esempio di come lo stesso calcolo possa essere realizzato in maniere equivalenti ma molto diverse per quanto riguarda il riempimento della pipe.

4.5.1 Effetto del pipelining sull’efficienza floating point

Nella sezione precedente è stato affermato che l’unità floating point è *pipelined*, cioè produce i suoi risultati al ritmo di uno per ciclo di clock, ma con una latenza fissa (10 cicli).

Quanto detto introduce il concetto essenziale di “dipendenza logica” tra una coppia di istruzioni. Due operazioni si dicono dipendenti se una richiede la conoscenza del risultato dell’altra per essere iniziata. Un esempio banale mostra che i due calcoli aritmetici $b = a + 1$ e $c = b + 1$ sono dipendenti.

Le dipendenze possono, però, essere ridotte formulando un problema in modo diverso. La scelta di una forma piuttosto che un’altra ha di solito conseguenze molto

⁴ oppure, a scelta del programmatore, su coppie di numeri reali.

evidenti sulla efficienza floating point. Questa sezione illustra questo concetto con un semplice esempio. Supponiamo di voler calcolare numericamente il valore del polinomio

$$\sum_{i=0}^n a_i x^i = a_n x^n + \dots + a_1 x + a_0$$

quando siano fissati i valori di tutte le variabili. È noto che una forma conveniente per eseguire la somma è quella di Horner (d'ora in avanti, per fissare le idee, assumiamo $n = 6$):

$$\sum_{i=0}^6 a_i x^i = ((a_6 x + a_5)x + a_4)x + \dots \quad (4.3)$$

che può essere anche espressa in modo ricorsivo:

$$P_n = a_n \quad P_{k-1} = P_k \cdot x + a_{k-1}$$

Il valore della somma cercato è P_0 . Questa forma ricorsiva è già in forma di algoritmo, perché ha una prescrizione iniziale ed ogni passo P_i viene calcolato dal risultato del passo precedente; la somma è quindi conclusa dopo aver valutato n operazioni normali.

Ci proponiamo ora di stimare quanti cicli di clock trascorrono dal primo passo (l'inizio del calcolo di P_{n-1}) al risultato (la fine del calcolo di P_0), sotto l'assunzione che l'unità aritmetica abbia una lunghezza di pipe finita. Lo svolgimento del calcolo in funzione del tempo è stato rappresentato in figura 4.5. Nell'asse orizzontale è mostrato il tempo. Ciascuna barra orizzontale rappresenta il calcolo di una operazione normale $a \cdot b + c$, che dura 10 cicli. Al ciclo 1 di ogni operazione normale la unità aritmetica richiede l'ingresso dei due fattori a e b , che nel nostro caso sono la costante x (in scuro) ed il valore P_i calcolato al passo precedente. Al ciclo 3 è richiesto l'addendo, la costante a_{i-1} . Il risultato è disponibile dopo 10 cicli. È evidente che l'operazione successiva non può partire fino a quando non è stata pagata per intero la latenza di tutte le operazioni precedenti.

Il costo del calcolo è quindi (trascurando fattori dell'ordine dell'unità) circa $10 \times n$ cicli. Se assumiamo di lavorare su valori complessi, una unità aritmetica con periodo di T di 5 ns (potenza di picco di 1.6 Gflops) starebbe fornendo con una potenza di

$$p = \frac{\text{operazioni}}{\text{tempo}} \simeq \frac{8 \cdot n}{10 \cdot n \cdot T} \simeq 0.16 \text{ Mflops}$$

che corrisponde ad una efficienza del 10% (il caso di peggiore uso della pipe).

La serie può essere scritta nella seguente forma:

$$\begin{aligned}
\sum_{i=0}^6 a_i x^i &= \left(\overbrace{(a_6 x^2 + a_4) x^2 + a_2}^{P_4} \right) x^2 + a_0 + \\
&+ \underbrace{\left(\overbrace{(a_5 x^2 + a_3) x^2 + a_1}^{P_3} \right) x}_{P_1}
\end{aligned} \tag{4.4}$$

che suggerisce di calcolarla come

$$\begin{aligned}
P_n &= a_n & P_{n-1} &= a_{n-1} \\
P_{k-2} &= P_k \cdot x^2 + a_{k-2} \\
\sum a_i x^i &= P_1 \cdot x + P_0
\end{aligned} \tag{4.5}$$

Il vantaggio di questa forma è che il calcolo di P_3 e di tutti i P di indice dispari non dipende da quello dei P pari e viceversa, a parte il risultato finale, che dipende da P_1 e P_0 . Lo schema di dipendenze che ne risulta permette di realizzare il diagramma dato in figura 4.6. Se trascuriamo nel conteggio le operazioni nell'ultima riga della (4.5), che è una approssimazione valida se n è grande, si ottiene che la stessa somma di prima è stata calcolata con efficienza circa doppia:

$$p \simeq \frac{8 \cdot n}{5 \cdot n \cdot T} \simeq 0.32 \text{ Mflops}$$

In figura c'è una linea tratteggiata verticale ed è attraversata da due operazioni. Il numero di operazioni attive a fissato t , diviso la latenza dell'unità, definisce una grandezza detta *riempimento della pipe*. In figura 4.5 il riempimento della pipe medio era ~ 0.1 e nel secondo esempio ~ 0.2 . Se la macchina non è arrestata da altre cause, questa misura coincide con l'efficienza dell'unità floating point.

Un altro modo di rappresentare le dipendenze è quello di creare un grafo orientato come quello di figura 4.7. Il tempo necessario per il calcolo è approssimativamente uguale ad $l \times g$. g è il numero massimo di nodi che è necessario attraversare per andare dal risultato a ciascuno degli operandi di ingresso.

Il ragionamento che ha portato dalla forma (4.3) alla (4.4) può essere ulteriormente esteso, dividendo la somma nelle tre serie P_{3k} , P_{3k+1} e P_{3k+2} e poi sommando i tre risultati finali. Se la serie da sommare contiene un numero di termini sufficientemente grande, la si può dividere anche in $4, 5, \dots, l$ somme parziali ed arrivare così ad avere una efficienza nell'impiego dell'unità floating point pari ad uno (sempre nel caso complesso).

Nel caso in cui si fosse interessati alla somma nel campo reale, l'uso di operazioni complesse sarebbe uno spreco di risorse: una normale complessa si compone di otto operazioni aritmetiche, delle quali si userebbe solo un risultato. Su apeNEXT si

può migliorare questa situazione sfruttando le operazioni cosiddette *vettoriali*, che agiscono contemporaneamente su coppie di grandezze reali. Questo parallelismo può essere usato per calcolare due serie indipendenti della stessa lunghezza, oppure i termini pari e quelli dispari allo stesso tempo. Nella pratica questo schema non è conveniente perché di solito le somme di polinomi (usate ad esempio per calcolare le funzioni trigonometriche) contengono pochi termini ed il passo finale di accumulo delle sottoserie diventa simile in complessità (e forma!) alla serie iniziale.

4.6 La rete di comunicazione

La rete di apeNEXT ha una capacità di trasmissione sostenuta, per ogni direzione dello spazio, per ogni verso, di 1.6 Gbit al secondo (pari a $25 \cdot 10^6$ parole reali), con una latenza dell'ordine dei 100 ns.

L'impiego principale della rete come si è visto è quello di realizzare degli accessi a memoria *remota* per copiarne il contenuto nei registri locali. In apeNEXT le comunicazioni tra memoria e register file non sono dirette, ma passano attraverso una FIFO (coda First In First Out). Il ruolo della FIFO o coda è di disaccoppiare l'istante dell'accesso in memoria da quello della scrittura sul register file. La FIFO, gestita dall'hardware, funziona da “serbatoio” temporaneo per contenere i dati arrivati dai nodi adiacenti o da quello locale, prima che il microcodice ne stabilisca la loro scrittura sui registri. Il modello di comunicazione che ne risulta è: **memoria** (locale o remota) \rightarrow FIFO \rightarrow registri.

Questo importante modello di comunicazione con *buffer* verrà descritto tra breve.

4.6.1 Comunicazioni sincrone

Ritornando ai valori dati all'inizio della sezione, il rapporto tra la larghezza di banda remota e quella locale è quindi

$$\frac{w_l}{w_r} = \frac{400 \cdot 10^6 \text{ parole/s}}{25 \cdot 10^6 \text{ parole/s}} = 16$$

Come è stato mostrato in precedenza i codici di calcolo tipici avranno $\rho = 10 \sim 15$, quindi la due larghezze di banda sono bilanciate sul problema esaminato.

Purtroppo questo non è sufficiente a garantire che le comunicazioni remote non introducano dei notevoli ritardi. Il modo più semplice di eseguire un accesso ad una memoria remota è quello di far partire una richiesta per un indirizzo a sul nodo r , prepararsi ad accogliere i dati negli opportuni registri e *fermare la macchina fino a quando i dati non arrivano*. Questo è quanto avviene in APEmille: la macchina va arrestata perché non vi è alcun disaccoppiamento tra la ricezione e la scrittura sul register file. La lista dei registri che ricevono i risultati della trasmissione è

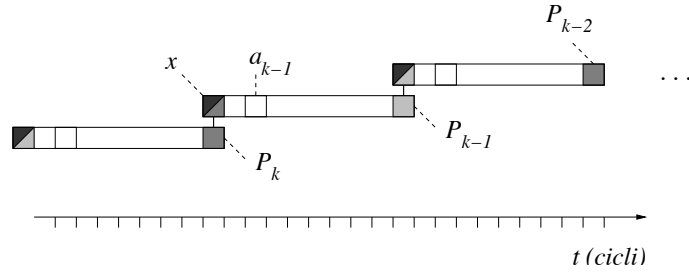


Figura 4.5: Calcolo di un polinomio mediante il metodo di Horner su una unità aritmetica di latenza $l = 10$: l'efficienza asintotica è $1/l$. L'unità è detta in pipe perché si può avviare una operazione aritmetica per ogni ciclo.

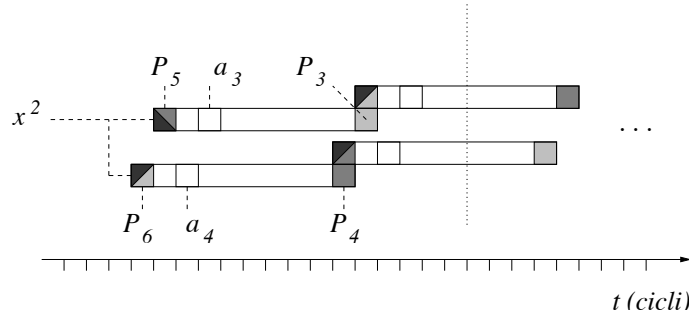


Figura 4.6: Il calcolo del valore di un polinomio in un punto, scritto nella forma (4.4), viene eseguito su una unità in pipe con efficienza doppia rispetto alla figura 4.5. Il riempimento della pipe, il numero di rettangoli che attraversano una linea a t costante, vale in media due.

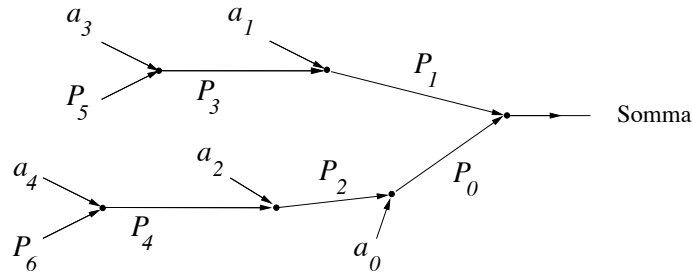


Figura 4.7: Determinare il costo del calcolo per mezzo di un grafo. La somma dell'equazione (4.4) viene rappresentata dal grafo in figura; ogni nodo è una operazione normale. (Gli operandi x^2 non sono mostrati). Il parametro g è il numero massimo di nodi da attraversare per andare dal risultato a ciascuno degli operandi di ingresso, che sono gli a_i , P_5 e P_6 . In questo disegno si vede che $g = 3 \simeq n/2$.

scritta nel microcodice e la sua esecuzione non può quindi proseguire fino a quando ciascuna parola non arriva a destinazione. (Il microcodice è una rappresentazione del codice da eseguire; esso descrive lo stato della macchina istante per istante, come sarà descritto in seguito in maggiore dettaglio.)

Questo modello di comunicazione purtroppo è causa di una notevole inefficienza: in media il tempo speso in comunicazioni remote sarà pari a quello speso in accessi alla memoria locale, però mentre la trasmissione è in corso la macchina è arrestata. Inoltre ad ogni comunicazione remota è associata una latenza, che è dovuta al tempo impiegato dal dato trasmesso a propagarsi lungo il cavo, al ritardo della memoria remota e molti altri fattori. Alla frequenza a cui lavora apeNEXT un ritardo di 100 ns corrisponde a molti cicli di macchina; nella soluzione descritta sopra il tempo di latenza si somma a quello di trasmissione.

Uno studio recente [27] ha quantificato l'importanza di questo fenomeno: la figura 4.8 mostra il calo dell'efficienza nell'applicazione di un operatore di Dirac al variare della frazione di operandi remoti, scritto come $f = 1/\rho$ (sull'asse delle ascisse). I punti sono stati ottenuti dalle misure di efficienza in reticoli di diverse dimensioni, con i risultati riportati in tabella 4.2. I valori sono stati misurati su una macchina APEmille, che segue il modello di comunicazione diretto memoria-registro. In ordinata è mostrato il tempo dell'esecuzione di una iterazione dell'operatore, in unità tali da valere 1 per $\rho = \infty$, che è il caso in cui tutte le comunicazioni sono locali (il calcolo è eseguito su un solo processore). Il grafico è spiegato dal modello discusso sopra: la trasmissione remota di un operando ha un costo costante di

$$T_r = \frac{1}{w_l} + \frac{1}{w_r} + \frac{L}{m}$$

dove w_l e w_r sono le bande passanti locale e remota (numero di parole che possono essere trasmesse nell'unità di tempo), L è la latenza e m è la dimensione media (in parole reali) della trasmissione. Nel caso dell'operatore \mathcal{D} vale $m = (24 + 24 + 18)/2 = 33$.

Se assumiamo che una frazione $f = 1/\rho$ di operandi sia remoto, il tempo medio per la trasmissione di una parola è di

Dimensione reticolo:					
locale	fisico	ρ	flops	$f = \rho^{-1}$	T (u.a.)
$2 \times 2 \times 2$	$4 \times 4 \times 4$	3.6	126	0.27	1.97
$4 \times 2 \times 2$	$4 \times 4 \times 4$	5.4	151	0.18	1.65
$4 \times 4 \times 4$	$8 \times 8 \times 8$	7.2	168	0.14	1.48
$4 \times 4 \times 2$	$4 \times 4 \times 4$	10.9	190	0.09	1.31
$4 \times 4 \times 4$	$4 \times 4 \times 4$	∞	249	0.00	1.00

Tabella 4.2: Impatto delle comunicazioni remote per comunicazioni sincrone

$$(1 - f) T_l + f T_r = \frac{1}{w_l} + f \left(\frac{1}{w_r} + \frac{L}{m} \right)$$

dal fit in figura si vede che la quantità tra parentesi vale 3.61: in conclusione una comunicazione remota costa quanto $(1 + 3.61)$ comunicazioni locali.

4.6.2 Comunicazioni con prefetch

Abbiamo visto che nel modello di comunicazione diretto memoria \rightarrow registro il tempo impiegato cresce linearmente con la frazione di accessi remoti e diventa più rilevante con il crescere della latenza L (ritardo della linea di trasmissione). L'impatto della latenza cresce per trasferimenti piccoli in media; in pratica il valore di m è fissato dalle dimensioni di U e ϕ .

La rete di comunicazione di apeNEXT è stata studiata in modo da risolvere questi limiti ed in particolare assorbire il ritardo della trasmissione. Durante un tempo L un clock di periodo T batte evidentemente L/T cicli. Assumiamo in prima approssimazione che la latenza L non dipenda dalla tecnologia impiegata: dal momento che $T_{\text{APEmille}} \simeq 3 \cdot T_{\text{apeNEXT}} \simeq 15$ ns si vede apeNEXT sprecherebbe tre volte più cicli ad attendere dati in transito.

Questo problema può essere risolto usando la FIFO per disaccoppiare la trasmissione del dato dalla sua scrittura sul register file. La trasmissione, che include il tempo di lettura dalla memoria, la attivazione del canale di comunicazione giusto, la propagazione del segnale e la ricezione nella coda di arrivo, è interamente

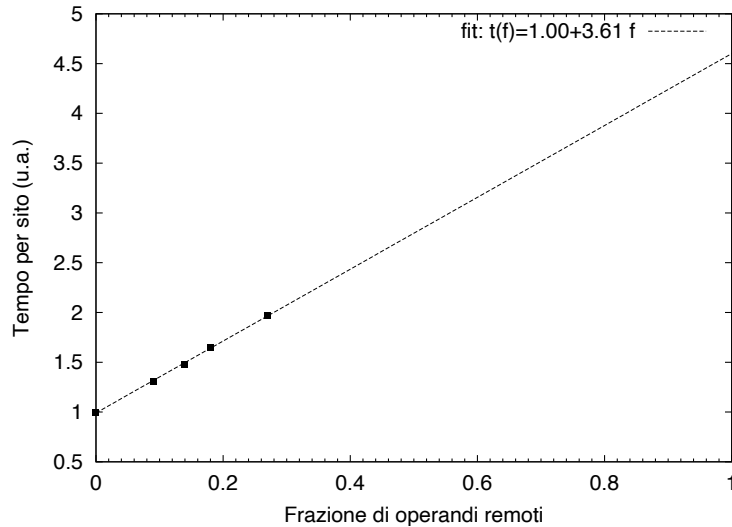


Figura 4.8: *Il tempo di calcolo cresce linearmente con la frazione di operandi remoti.* Il grafico mostra il tempo di calcolo per sito, misurato su un codice “operatore di Dirac”, su macchine APEmille di DESY-Zeuthen. Il margine destro mostra l’extrapolazione per $f = 1$, cioè il tempo medio di un accesso remoto.

gestita dall'hardware, che la porta avanti senza intervento del microcodice. Se si programma questa fase almeno con anticipo sufficiente ad assorbire la latenza della rete (~ 100 ns), quella della memoria (~ 20 ns) ed il calcolo dell'indirizzo (~ 10 cicli = 50 ns), si può essere ragionevolmente certi che l'unità di calcolo continuerà a funzionare in parallelo con la trasmissione.

Supponiamo invece che l'utente richieda il passaggio **memoria** \rightarrow **coda** (1) e poi in rapida successione quello **coda** \rightarrow **registro** (2). In questo caso è estremamente probabile che a causa dell'operazione (2) il nodo dovrà bloccarsi fino a quando non avrà ricevuto i dati trasmessi dalla (1). Questa situazione è nota come *stretch*. È evidente che lo stretch è una situazione non desiderabile ed una causa di perdita di efficienza: se il nodo è bloccato in attesa di dati, il tempo passa senza che l'unità aritmetica esegua calcoli. Durante gli stretch il clock della macchina è bloccato ed il microcodice non avanza.

4.6.3 Compensazione nei siti di bordo

L'uso di questa tecnica, nota come prefetch oppure preload, ha anche un altro vantaggio. Il valore di $1/\rho$ dà la frazione di operandi che sono remoti, *mediato* su tutti i punti dello spazio. Il numero per così dire *istantaneo* di accessi remoti invece dipende fortemente da quale parte del reticolo si sta aggiornando: l'aggiornamento dei siti posti nel centro del reticolo locale non ha bisogno di alcuna comunicazione remota. I siti posti nel centro di una delle sei facce comunicheranno ciascuno con un processore confinante (si veda di nuovo la figura 4.3; stiamo assumendo un reticolo locale non degenerare, cioè con ogni lato più grande di un sito), quelli sullo spigolo con due e quelli sul vertice con tre. È evidente quindi che l'aggiornamento dei siti posti sul bordo del reticolo locale richiede un numero di accessi remoti superiore alla media: se dimensionassimo la rete in modo da bilanciare il carico medio, la larghezza di banda sarebbe insufficiente per questi siti, che sarebbero quindi causa di rallentamento.

L'uso del prefetch risolve anche questo problema: le trasmissioni nelle due direzioni remote saranno attivate senza che una attenda il completamento dell'altra e quindi avverranno in parallelo. Anche le due latenze saranno pagate una volta sola. Avere uno, due o tre processori confinanti con cui comunicare ha approssimativamente lo stesso impatto sul tempo dell'aggiornamento: i siti di bordo hanno più trasferimenti da eseguire, ma anche più banda passante a disposizione, che ne compensa il costo.

Possiamo calcolare in modo molto approssimato il vantaggio che si ha nei due casi in cui le comunicazioni in direzioni diverse siano del tutto sincrone ovvero si possano sovrapporre. Supponiamo di avere un reticolo locale di $4^3 \times 8$ siti di cui consideriamo, come prima, un volume a t costante. Assumiamo che l'aggiornamento di ogni sito

avvenga prelevando una uguale quantità di dati da ciascuno dei siti vicini. Allora la spazzata del volume considerato richiederà $6n^2 = 96$ accessi remoti, pari al numero di link che escono dal cubo in figura 4.3, che nel caso del tutto sincrono sarà proporzionale al tempo impiegato per eseguire gli accessi remoti.

Nel caso in cui gli accessi si possano sovrapporre, invece, si può assumere che l'aggiornamento di ciascuno dei siti posti sulla superficie del cubo impieghi una uguale quantità di tempo; il tempo per aggiornare l'intero volume sarà proporzionale al numero di siti che si affacciano all'esterno del cubo, che è $6n^2 - 12n + 8 = 56$.

Il vantaggio (96 contro 56) della strategia di sovrapporre i trasferimenti in direzioni ortogonali è evidente.

4.6.4 Altre caratteristiche

Su ogni nodo è presente un settimo collegamento che viene usato per comunicare con l'esterno, oppure per richiudere un gruppo di nodi su se stessi. Questa caratteristica è utile per partizionare una macchina grande in altre indipendenti e più piccole, allo scopo ad esempio di dedicarle alla prova di nuovi algoritmi.

La rete di comunicazione inoltre generalizza in due modi il modello di comunicazione omogeneo o SIMD, in cui tutti i nodi trasmettono nella stessa direzione contemporaneamente e ricevono da quella opposta: le comunicazioni possono essere asimmetriche (la direzione di ingresso può essere non essere opposta a quella di uscita) e disomogenee (diverse in ogni nodo della macchina, purchè ognuno abbia al più una comunicazione entrante ed una uscente).

Queste estensioni non sono strettamente richieste nelle simulazioni di QCD ma possono aumentare notevolmente l'efficienza su altri eventuali programmi, come quelli scritti con metodi ipersistolici [28].

4.7 La generazione degli indirizzi

I valori dei campi ed i potenziali di ogni sito sono immagazzinati in indirizzi contigui della memoria ad accesso casuale. Durante l'esecuzione del programma, i dati vanno trasportati dalla memoria ai registri per essere da lì elaborati nell'unità floating point, ma questa operazione ("load") presuppone il calcolo dell'indirizzo a partire

		Banda	Rapporto con		
		(parole/sec)	w_f	w_l	w_r
Floating point	w_f	$1.6 \cdot 10^9$	1.0	4.0	64.0
Memoria	w_l	$400 \cdot 10^6$		1.0	16.0
Rete	w_r	$25 \cdot 10^6$			1.0

Tabella 4.3: Rapporti tra le bande passanti nel nodo di apeNEXT

dalle assegnate coordinate spaziali e temporali del sito e degli indici di spin e di colore.

Per non interferire con i calcoli in virgola mobile che avvengono nello stesso istante, si è prevista una cosiddetta AGU (Address Generation Unit), che realizza in modo indipendente somme e prodotti tra indici. Somme e prodotti per costante sono le operazioni necessarie per convertire una coppia di indici, ad esempio spin e colore, in un unico intero che indirizza gli stessi elementi considerati come un lungo vettore. L'uscita della AGU è collegata al bus di indirizzi della memoria e viene letta su un opportuno strobe, sollevato esplicitamente dal programma quando il calcolo è concluso e l'indirizzo è valido.

4.8 Sincronizzazione e controllo globale

L'architettura di apeNEXT (a differenza di quella delle precenti macchine APE) non prevede che i nodi siano strettamente sincronizzati; inoltre i nodi possono anche eseguire programmi diversi tra loro, perché il modello SIMD non è imposto dall'hardware (questa caratteristica non è necessaria per calcoli di LGT). Ci sono tuttavia due casi in cui i nodi vengono sincronizzati:

- Quando avviene una comunicazione remota
- Quando occorre valutare una condizione per tutta la macchina

Lo scambio di dati tra due nodi è cooperativo, nel senso che richiede che entrambi siano giunti nello stesso punto del programma, quello dove è specificato che la comunicazione deve avvenire.

Supponiamo di avere due nodi A e B che eseguono lo stesso programma e che A sia in anticipo per qualche motivo (figura 4.9 *a*). Dal momento che il comando di ricezione (rettangolo chiaro) blocca il nodo fino a quando i dati non sono ricevuti, un effetto collaterale della comunicazione remota è quello di sincronizzare⁵ A e B .

⁵ A meno della differenza tra la latenza remote ed il tempo che intercorre tra T ed R.

Potenza di calcolo (64 bit):	
complessa	1.6 Gflops
reale	0.8 Mflops
Velocità memoria	25.6 Gbit/s
Dimensione della memoria (dati+programma)	2 Gbit
Banda passante rete	1.6 Gbit/s/dir

Tabella 4.4: I parametri di apeNEXT (per un singolo nodo)

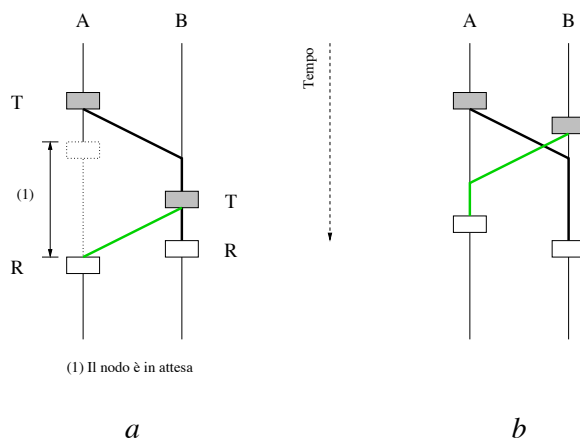


Figura 4.9: Le comunicazioni remote causano la sincronizzazione dei nodi di calcolo.

Nel capitolo 5 sarà discussa la tecnica detta del prefetch che permette di eliminare i cicli di attesa della rete, aumentando l'intervallo di tempo tra T ed R. È interessante notare che in questo caso (figura 4.9 b) la sincronizzazione non avviene.

Il secondo caso di sincronizzazione elencato si presenta quando occorre valutare la verità di una condizione non localmente ad un nodo, ma globalmente, come somma o prodotto dei valori di verità di condizioni locali. Si supponga ad esempio di avere un programma che altera i valori degli elementi di un vettore fino a quando una qualche misura di errore E non sia inferiore ad una soglia ϵ stabilita su *tutti* i nodi, nel quale caso si vuole che l'aggiornamento termini. Lo stesso algoritmo sarà eseguito in parallelo, ma con condizioni iniziali diverse, per cui succederà ad un certo istante che in alcuni nodi la condizione $E < \epsilon$ è soddisfatta, mentre in altri no. Una opportuna istruzione (nota sotto il nome di *if-any*) attiva una rete speciale che propaga questa informazione. La decisione se proseguire con altre iterazioni oppure no può essere presa solo quando tutti i nodi abbiano dichiarato la loro condizione, da cui la necessità della sincronizzazione.

4.9 Il microcodice

Il nodo di calcolo di apeNEXT è composto dalle unità funzionali accennate nelle sezioni precedenti. In alcuni casi le unità possono essere attivate in maniera indipendente e lavorare contemporaneamente, mentre in altri occorre che il risultato dell'elaborazione di una sia trasferito all'ingresso di un'altra e nello stesso istante occorre che la seconda venga attivata. Questo schema di dipendenze è implicito nel codice da eseguire: la *decodifica delle istruzioni* è il processo con cui dal codice si decide quali segnali devono essere presenti agli ingressi di ogni unità funzionale in ogni istante dell'esecuzione del codice.

Il problema della decodifica delle istruzioni è comune ad ogni processore programmabile; nei processori commerciali essa viene eseguita dall'hardware mentre la macchina esegue il programma. L'adottare questo approccio in una macchina dedicata avrebbe due notevoli svantaggi: innanzitutto appesantirebbe la progettazione con lo sviluppo di una parte molto complicata; inoltre è molto difficile realizzare un hardware che trovi la temporizzazione ottimale da dare alle istruzioni.

La soluzione adottata in apeNEXT è stata quella di rendere il processore *micro-programmato*: la temporizzazione dei segnali è calcolata prima che il programma venga eseguito e fa parte della catena di compilazione. La figura 4.10 mostra un segmento di microcodice tratto dal codice che applica l'operatore di Dirac. Ogni riga corrisponde ad un ciclo di 5 ns; il tempo scorre verso il basso. La colonna MPC contiene i comandi per la unità aritmetica ed è non vuota quando si vuole avviare una operazione: al secondo dei cicli mostrati c'è il comando di somma tra due grandezze intere. Quando l'unità aritmetica riceve un comando, legge gli operandi dai registri indicati nelle ultime tre colonne; restituisce il risultato scrivendolo nel registro indicato sotto la colonna P3, dopo la lunghezza di pipe nota (in questo caso 4 cicli). Gli altri campi funzionano in modo simile. La D ad esempio è una operazione di AGU che copia il valore immediato presente sotto DISP in un registro interno. Il codice LAL porta il registro interno sul bus indirizzi della memoria, preparandola così ad un successivo trasferimento di dati (potenzialmente remoto).

Il numero di linee di microcodice comprese tra due istruzioni corrisponde esattamente al numero di cicli di clock che le separa. Quindi la lunghezza di un core di calcolo in linee di microcodice è una stima inferiore del tempo che esso impiegherà nella sua esecuzione. Questa stima può essere fatta anche senza simulare la macchina, ma essa purtroppo non è esatta perché non tiene conto di due potenziali eventi:

Fifo stretch avviene potenzialmente quando si esegue la seconda parte di un trasferimento remoto: il microcodice comanda la copia del contenuto della coda in un insieme di registri dati, ma i dati potrebbero non essere ancora disponibili perché il tempo di transito sul link non è noto a priori con precisione (e potrebbe essere ritardato da trasferimenti precedenti nella stessa direzione).

Memory stretch che accade perché la memoria ha una latenza variabile: se, a causa di molti fattori, i dati non sono disponibili quando sono richiesti, occorre fermare il microcodice. Un memory stretch può avere una durata compresa tra 5 e 100 ns.

Queste due cause di rallentamento possono essere misurate solo osservando il comportamento del programma nell'hardware, o nella sua versione simulata.

ADDR	X	DISP	MCC	MS	STKC	FLW	IOC	AGU	ASEL	P5	BS4	C4	P4	MPC	BS3	C3	P3	P2	P1	P0
0BC5:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	3	0	4a	00	00	00
0BC6:	-	0000000e	-	0	-	-	-	D	-	00	0	0	00	IADD	0	0	00	00	01	14
0BC7:	-	00000000	-	0	-	-	-	-	LSL	00	0	0	00	-	0	0	00	00	00	00
0BC8:	-	00000f23	M2IA	0	-	-	-	D	-	00	0	0	00	-	0	0	00	00	00	00
0BC9:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BCA:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	1	0	14	00	00	00
0BCB:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	ISUB	0	0	00	00	15	14
0BCC:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BCD:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BCE:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BCF:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	1	0	ff	00	00	00
0BD0:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BD1:	-	00000000	-	0	PGT	SLP	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BD2:	-	00000000	-	0	-	-	-	-	LAL	00	0	0	00	-	0	0	00	00	00	00
0BD3:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BD4:	-	0000000e	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BD5:	-	00000000	-	0	-	-	-	D	-	00	0	0	00	-	0	0	00	00	00	00
0BD6:	-	00000000	M2IAC	0	POP	-	-	-	LSL	00	0	0	00	-	0	0	00	00	00	00
0BD7:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00
0BD8:	-	00000000	-	0	-	-	-	-	-	00	0	0	00	-	0	0	00	00	00	00

Figura 4.10: *Esempio di microcodice*. Queste linee fanno parte del microcodice del programma che applica l'operatore di Dirac iterando su tutti i siti del reticolo. La parte mostrata incrementa di uno l'indice n del sito corrente (riquadro tratteggiato) e lo confronta con il numero di siti (grigio chiaro): se il valore dell'indice è maggiore, il programma ha termine. È interessante notare tre fenomeni:

1. contemporaneamente al calcolo ed al confronto, hanno luogo una serie di istruzioni che preparano il caricamento di un nuovo indirizzo di programma, per l'eventuale uscita dal ciclo (grigio scuro);
2. nel riquadro trasparente è evidente l'esistenza di una pipe di latenza $l = 4$ per le operazioni intere. Il registro 14 contiene l'indice n e ad esso viene sommato il contenuto del registro 1. Il risultato della somma sovrascrive lo stesso registro 14, dopo quattro cicli.
3. Il confronto tra i registri 14 e 15 non può iniziare se non quando la scrittura nel registro 14 è avvenuta (dipendenza).

4.10 Architettura del nodo di apeNEXT

Questa sezione esaminerà in maniera dettagliata come le richieste elencate nei paragrafi precedenti sono state affrontate nel nodo di calcolo di apeNEXT. Il tipo di progetto è detto *custom* perché, a differenza di altre analoghe macchine dedicate, lo sviluppo dell'hardware del nodo parte da zero piuttosto che su componenti di libreria commerciali o processori già pronti.

La figura 4.11 dà una visione generale delle unità che compongono ciascuno dei nodi di calcolo di apeNEXT. I blocchi disegnati verranno descritti in qualche dettaglio nel seguito.

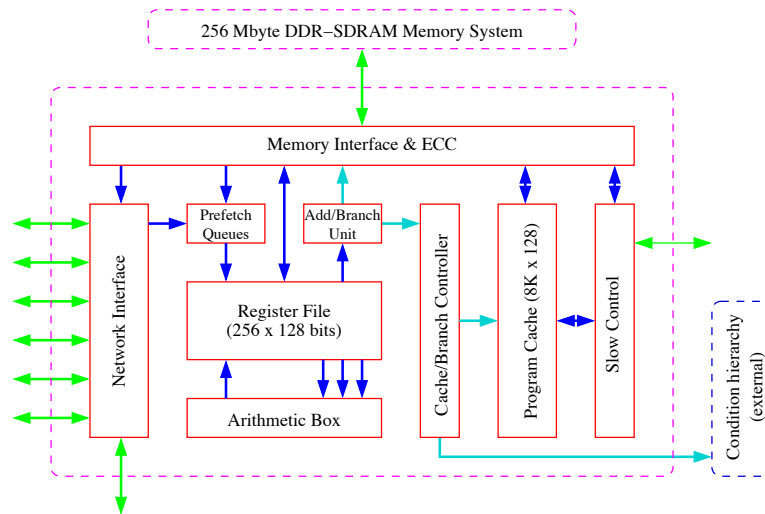


Figura 4.11: Schema semplificato del nodo di calcolo di apeNEXT.

4.10.1 Il register file

La figura 4.12 mostra un diagramma generale dell'architettura e mette in evidenza il ruolo del register file, senza mostrare i dettagli di memoria e rete, che saranno descritti in seguito.

Il register file è una memoria che può fornire e registrare dati con latenza pari ad un ciclo e comunicare da e verso l'esterno contemporaneamente attraverso sei porte.

La capienza del register file è di 256 parole da 128 bit ciascuna. Ogni parola può essere utilizzata per contenere un numero complesso floating point, oppure coppie di grandezze reali, oppure ancora coppie di interi. In ogni caso la precisione del dato è di 64 bit.

Tre porte di uscita sono collegate all'unità aritmetica in virgola mobile (FBB) e forniscono gli operandi a , b e c delle operazioni normali. Le stesse porte possono

portare operandi interi nell'unità aritmetica intera (IBB), oppure nel blocco *lookup table* (LBB). Questi blocchi saranno descritti nel seguito.

La quarta porta è una porta di ingresso e permette la scrittura del risultato dei calcoli svolti nella unità aritmetica. Nella figura la porta esce dal multiplexer *outMux*, che in ogni ciclo seleziona il blocco aritmetico che contiene il risultato valido. La quinta porta (in alto a sinistra) è bidirezionale e collega il register file alla memoria. Anche essa è larga 128 bit; a parte la latenza della memoria, ciascuna di queste porte può trasmettere al ritmo di una parola per ciclo.

La sesta ed ultima porta serve a comunicare i dati alla AGU, al il cui ruolo si è già accennato. Questa porta è collegata solo ai primi 64 registri; questa restrizione è stata adottata per ridurre la quantità di spazio sul chip occupato dal register file (tabella 4.5). Essa non è una grave limitazione perché i codici di calcolo utilizzano tipicamente un numero di indici dell'ordine della decina.

Anche il numero di registri è stato fissato considerandone l'uso di un tipico programma. Se il numero di registri disponibili non fosse sufficiente, alcuni valori temporanei non potrebbero essere memorizzati nel register file e ma dovrebbero essere conservati e ricaricati dalla memoria. Questo causerebbe un aumento delle richieste di banda passante verso la memoria e le prestazioni dei programmi diminuirebbero di conseguenza.

Il codice “operatore di Dirac” scritto dall'autore utilizza i registri dal 82 al 228 per i calcoli. Sebbene l'utilizzo dei registri potrebbe essere ridotto adottando un *register renaming*, si è scelto di adottare una dimensione di 256 registri perché sufficiente e dotata di un ragionevole margine di sicurezza. Questa scelta è sopportata inoltre dall'esperienza di APEmille, che ha dimostrato che 256 coppie di registri sono sufficienti in ogni kernel calcolo, inclusi quelli che sono stati impiegati in simulazioni di fluidodinamica.

Le sei porte del register file sono indirizzate da altrettanti bus di indirizzi indipendenti. I bus sono visibili nella figura 4.10 sotto le colonne P0 – P5. Le porte di *ingresso* nel register file sono accompagnate da altre colonne il cui nome inizia per BS e C. La prima è composta da due bit che permettono di inibire la scrittura su uno o entrambi i due banchi che compongono il register file, mascherando così la memorizzazione della parte reale od immaginaria a richiesta. Il bit sotto la colonna C vale uno se la scrittura non deve essere condizionata dal valore della condizione sullo stack.

4.10.2 Il collegamento tra memoria e rete

Questa sezione descrive le interfacce di memoria e di rete, che nella figura 4.12 erano indicate con un ellisse. La relazione generale tra questi due dispositivi è indicata in figura 4.13. Nel diagramma sono visibili diversi percorsi:

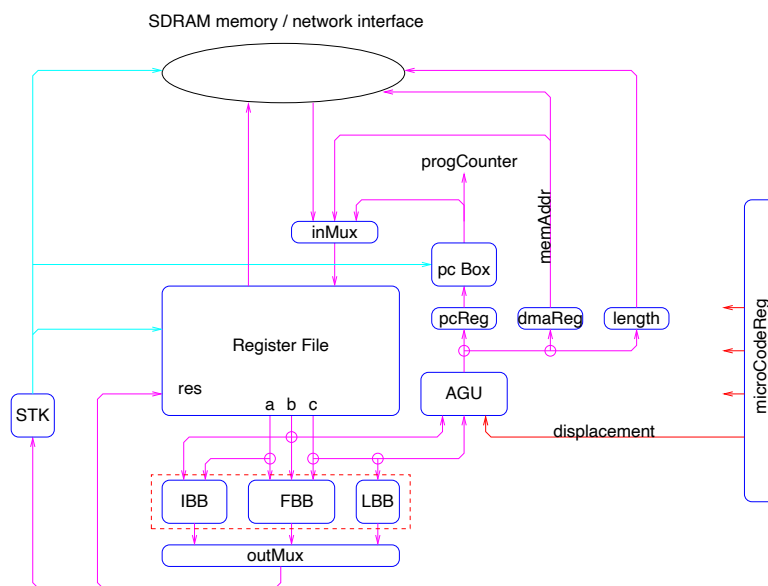


Figura 4.12: Diagramma a blocchi dell'architettura del nodo di apeNEXT. Il ruolo del register file è posto in evidenza.

Unità	APEmille (porte)	incremento	apeNEXT (porte)	apeNEXT (mm ²)
Register file	200 K	2	400 K	5.0
Floating point	100 K	2.5	250 K	3.7
Rete	30 K	2	60 K	1.0
Cache istruzioni	—	—	4 K × 128 bit	34
Totale	330 K	2 + cache	700 K + cache	48

Tabella 4.5: Stima del numero di porte ed aree richieste su chip per ciascuna componente del nodo di calcolo di apeNEXT.

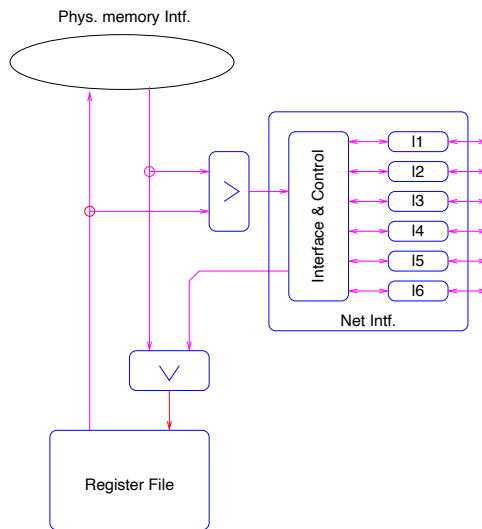


Figura 4.13: Diagramma a blocchi di alto livello delle interfacce di rete e di memoria.

- Un percorso diretto tra il register file e la memoria
- I dati che provengono dalla memoria possono essere inviati verso l'interfaccia di rete per essere trasmessi ad un altro nodo
- I dati in arrivo dalla rete sono inviati al register file
- Il register file può inviare dati verso la rete (in questo modo vengono realizzate comunicazioni remote registro \rightarrow registro)
- Per ogni trasferimento remoto, la rete seleziona uno dei sei link esaminando una parte dell'indirizzo chiamata *bit remoti*.

La strategia dei bit remoti è una estensione del modello adottato in APEmille. Lo spazio di indirizzamento della memoria fisica è di 27 bit, ma il bus degli indirizzi è largo 64 bit. I bit compresi tra il 32 ed il 47 vengono considerati dall'hardware che li usa per decidere quali code attivare.

Nel caso di una comunicazione a primo vicino nella direzione X^+ (figura 4.14), ad esempio, il programma accedrebbe all'indirizzo remoto dato dalla coppia 00.00.00.10 00.00.00.01, che prepara la rete a trasmettere nella direzione X^+ e ricevere da quella X^- . È possibile eseguire trasferimenti in cui più di un bit remoto sia acceso, come sarà descritto in una sezione successiva.

4.10.3 La memoria fisica

Ogni nodo di calcolo sarà dotato di una quantità di memoria compresa tra 4 e 16 Gbit. La memoria sarà composta da chip commerciali DDR SDRAM (Double Data

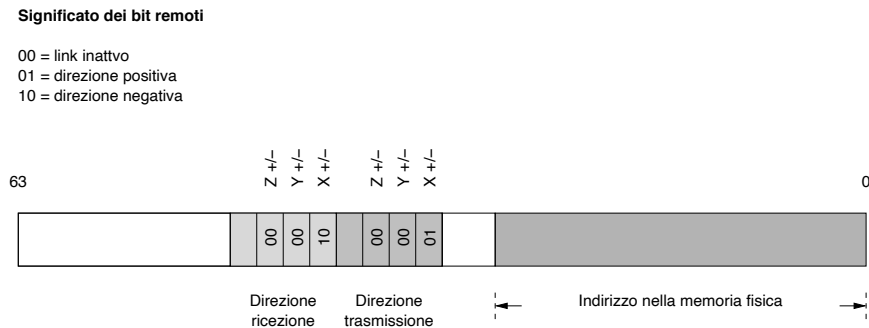


Figura 4.14: Codifica di bit remoti ed indirizzo in un'unica parola di 64 bit.

Rate Synchronous Dynamic RAM), che possono trasferire i dati su un bus di 16 bit al rate di due parole per ciclo di clock, che è di 100 MHz. La quantità di memoria desiderata sarà raggiunta unendo otto banchi da 16 bit, più uno per la correzione di errore; i banchi lavorano contemporaneamente per fornire parole larghe 128 bit. La banda passante totale della memoria è quindi capace di fornire parole di 128 bit al ritmo di 200 MHz.

La gestione della memoria e delle temporizzazioni degli accessi è affidata a quattro memory controller appositamente sviluppati che lavorano in parallelo. Gli accessi in memoria possono essere parzialmente sovrapposti (pipelined) in modo da ridurre l'impatto delle latenze. I casi comuni in cui la latenza della memoria si manifesta con un memory stretch sono: accessi ad indirizzi di parità dispari (1 ciclo); accessi ad un numero dispari di parole (1 ciclo); accessi alla stessa pagina dello stesso banco (5–20 cicli).

Agli otto banchi di memoria ne viene affiancato un nono, che contiene le informazioni ridondanti di controllo di parità (ECC, Error Correcting Code). L'ECC o EDAC permette di correggere errori di un singolo bit su ogni parola e di riconoscere la presenza di errori doppi. Il codice di correzione aumenta l'affidabilità della macchina e consente di generare eccezioni nei casi in cui il programma dovesse accedere per errore ad un indirizzo di memoria non inizializzato.

4.10.4 L'unità floating point

L'unità di calcolo di apeNEXT (FILU, per Floating Integer Logic Unit) svolge i compiti seguenti:

- Esegue le operazioni in virgola mobile ed intere (meno frequentemente). I calcoli floating point sono eseguiti nella rappresentazione IEEE a 64 bit (v. appendice).
- Restituisce la prima approssimazione di alcune funzioni speciali. Questa stima viene assunta come partenza per algoritmi iterativi (LUT).

- Le condizioni logiche “zero” e “positivo” sono associate al risultato di ogni operazione aritmetica. In questo modo le condizioni tipo $a < b$ sono valutate all’interno dell’unità FILU considerando il segno di $a - b$.

L’architettura di FILU è mostrata in figura 4.15. FILU usa dati memorizzati nel register file, composto da 256 coppie di valori. Le parti reali ed immaginaria di un operando complesso sono memorizzate nelle due metà di ogni parola; le stesse parti possono anche contenere le due componenti di un operando vettoriale. Nel diagramma si riconoscono quattro blocchi uguali: ciascuno di essi realizza una operazione normale su numeri reali a 64 bit. Il multiplexer sulla destra seleziona il tipo di operazione desiderata: nel caso di operandi reali si prende l’uscita di due blocchi soltanto, che eseguono operazioni reali sulle due metà del register file senza mescolarle. Nel caso sia richiesto il risultato di una operazione complessa, intervengono anche gli altri due blocchi. La tabella 4.6 riassume il numero di operazioni che possono essere eseguite da FILU nelle diverse modalità.

4.10.5 La rete di trasmissione

Con una potenza di calcolo di 1.6 Gflops per nodo, i parametri $\rho \sim 8$ e $R \sim 4$ impongono l’adozione di una tecnologia di trasmissione capace di sostenere velocità dell’ordine dei 3 Gbit/s. La tecnologia che è stata adottata è quella LVDS. Un link seriale LVDS permette di raggiungere velocità di trasferimento di dati dell’ordine di 600 Mbit/s, per mezzo di linee differenziali formate da una coppia di conduttori.

Ogni nodo di apeNEXT possiede sei collegamenti a nodi vicini, ciascuno dei quali è realizzato con otto coppie LVDS; ogni link ha (nella attuale versione) la stessa frequenza del nodo di 200 MHz. La banda passante per ogni direzione è quindi $8 \times 200 \cdot 10^6 = 1.6 \cdot 10^9$ bit/s, che corrispondono ad una parola complessa ogni 16 cicli di clock. La frequenza di arrivo delle parole trasmesse è in realtà una ogni 17 cicli a causa dell’aggiunta di una parola di controllo ogni 16. Ogni nodo è dotato di un ulteriore link bidirezionale, il settimo, il cui uso sarà discusso tra breve.

La rete supporta questi tipi di trasferimenti rigidi:

Tipo	Istruzioni per sec.	Latenza ($\times 5$ ns)	Capienza del RF
Complessa	1600 Mflops	10	256
Reale	400 Mflops	7	512
Reale vettoriale	800 Mflops	7	256
Intera	200 Mips	4	512
Intera vettoriale	400 Mips	4	256

Tabella 4.6: Potenze di picco del processore matematico.

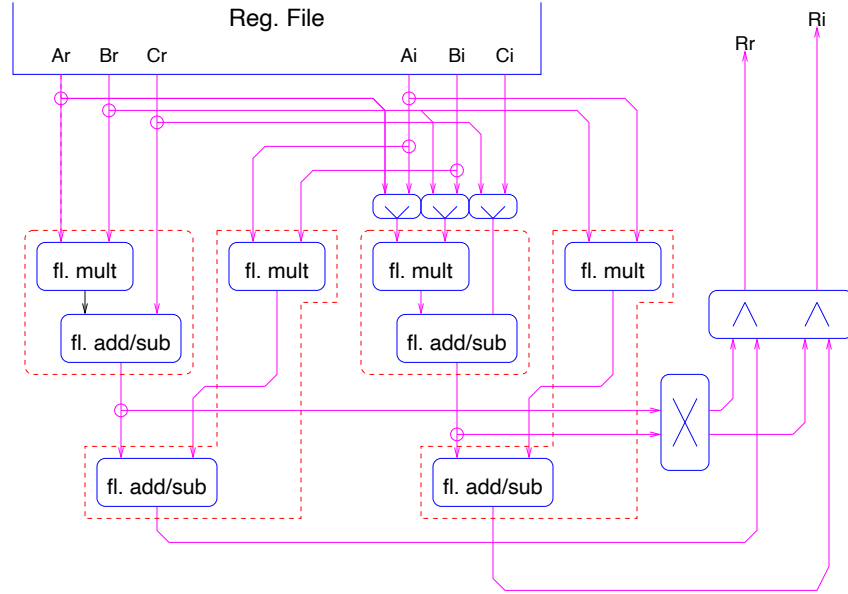


Figura 4.15: *Diagramma a blocchi delle sottounità floating point all'interno del processore matematico. L'operazione normale complessa è realizzata da quattro operazioni normali reali, nei riquadri.*

- trasmissioni a primo vicino, nelle direzioni positiva o negativa di uno qualsiasi dei tre assi della macchina
- trasmissioni a squadra, in cui si eseguono due salti di un passo in direzioni ortogonali
- trasmissioni a sedia, in cui il dato esegue tre salti di lunghezza unitaria nelle tre direzioni.

In altre parole, se indichiamo le coordinate di ciascun nodo nella rete con (x, y, z) , i collegamenti permessi sono quelli che trasportano

$$(x, y, z) \rightarrow (x + \Delta x, y + \Delta y, z + \Delta z)$$

con $|\Delta i| \leq 1$ e $\sum_i |\Delta i| \leq 3$.

Le interfacce di rete possono essere programmate per eseguire trasferimenti non omogenei, ovvero in cui il salto Δi è diverso per ciascun nodo, purché lo schema delle trasmissioni sia tale che ogni nodo abbia al più una comunicazione entrante ed un uguale numero di comunicazioni uscenti.

Questa possibilità verrà sfruttata per permettere alla macchina di comunicare con l'esterno, ad esempio per registrare le configurazioni calcolate su dischi condivisi. La figura 4.16 mostra come questo sia possibile. La freccia obliqua indica il *settimo link*, che è una interfaccia di comunicazione simile alle 6 presenti per ogni direzione, solo che è collegata con un PC su cui gira un sistema operativo che permette lo scambio di dati tra la macchina ed il mondo esterno.

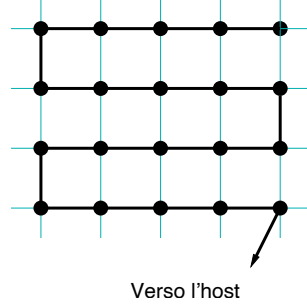


Figura 4.16: Il contenuto della memoria di tutta la macchina è trasmesso all'esterno passando attraverso il settimo link di uno o più nodi.

Il pattern di comunicazione “a serpente” mostrato in figura può essere ottenuto in maniera elegante grazie alla caratteristica di *remap* dei link. Per mezzo di un remap si può fare in modo che ciascun nodo “veda” come suo vicino nella direzione (per esempio) X^+ quello che in realtà potrebbe essere nella direzione (per esempio) Y^+ . Tutti i nodi mostrati in figura eseguirebbero quindi lo stesso programma e comunicerebbero con un vicino nella stessa direzione, mentre effettivamente i dati percorrerebbero tutta la macchina.

Il settimo link e la caratteristica di remapping consentono di suddividere una macchina in due o più macchine più piccole ed indipendenti dette *partizioni*, come mostrato in figura 4.17. Il partizionamento è trasparente per l'utente, che può eseguire lo stesso programma senza modifiche e con gli stessi risultati su una macchina piccola allo stesso modo che su una partizione di pari dimensioni di una grande.

Il remapping può essere realizzato senza arrestare il programma in esecuzione (è sufficiente che non ci siano trasferimenti remoti attivi). Con delle strategie simili a quella mostrata in figura 4.18, la topologia naturale della macchina, il 3-toro (o una sua parte), può essere rimappata in un anello. Questa topologia risulta particolarmente adatta a problemi come la FFT e quelli ad n corpi.

La rete è concettualmente sincrona e permette di eseguire programmi che seguono il modello omogeneo SIMD; al livello fisico i link non sono sincronizzati rigidamente. Accade quanto segue:

- Su ciascun nodo è avviato un programma che prevede una sequenza definita di operazioni remote, di dimensioni note.
- Ad ogni comunicazione è assegnato un set di valori $(x + \Delta x, y + \Delta y, z +$

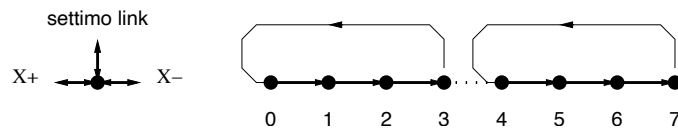


Figura 4.17: Una macchina di otto nodi può essere partizionata in due macchine indipendenti di quattro nodi. I nodi 3 e 7 trasmettono e ricevono sul settimo link anziché nella direzione X^+ ; esso è chiuso sul link X^- delle macchine 0 e 4.

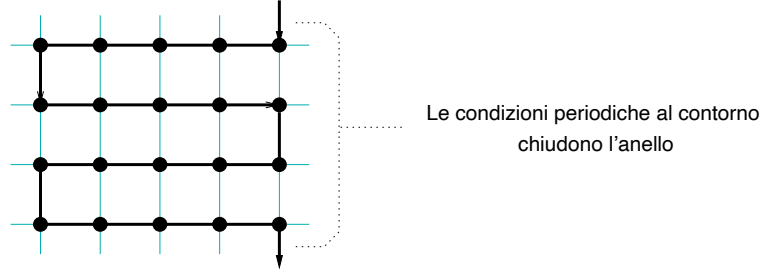


Figura 4.18: Il remapping permette di trasformare in maniera naturale la topologia a griglia in una ad anello.

$\Delta z, S, N$), dove i Δi individuano la direzione del trasferimento, S è la dimensione del trasferimento ed N è un intero diverso per ogni messaggio.

- L'interfaccia di rete di ciascun nodo accetta i dati in arrivo e prova a smistarli verso il nodo di destinazione. Tutti i nodi inviano pacchetti nello stesso ordine, ma il tempo esatto in cui questi pacchetti partono può essere lievemente diverso tra i nodi.
- Lo smistamento dei pacchetti è comandato dalle regole seguenti:
 1. Ciascuna interfaccia di rete rifiuta un pacchetto che proviene da un altro nodo con indice N , a meno che non sia essa stessa stata istruita ad iniziare la trasmissione del pacchetto N .
 2. Ciascuna interfaccia trasmette i dati in arrivo in ordine di N crescente.

In questo modo tutti i messaggi raggiungono la destinazione nell'ordine corretto. La figura 4.19 mostra le unità che sono attivate durante un trasferimento ad L. (Ricordo che questi passi avvengono simultaneamente in tutti i nodi.) Supponiamo di voler eseguire un trasferimento nel piano xy : avvengono, nell'ordine, i seguenti passi.

1. Il nodo trasmittente attiva la parte di decodifica dell'indirizzo ed istruisce il memory controller a trasmettere (STX) un certo numero di dati nella direzione $(+1, +1, 0)$.
2. Il dato passa attraverso il link X^+ e raggiunge il nodo adiacente, che non è il destinatario del trasferimento. L'STX del nodo intermedio è stato già attivato, quindi il messaggio viene instradato ulteriormente verso la direzione Y^+ .
3. Il dato passa attraverso il link fisico e raggiunge il nodo di destinazione, che lo copia (ETX) nei registri.

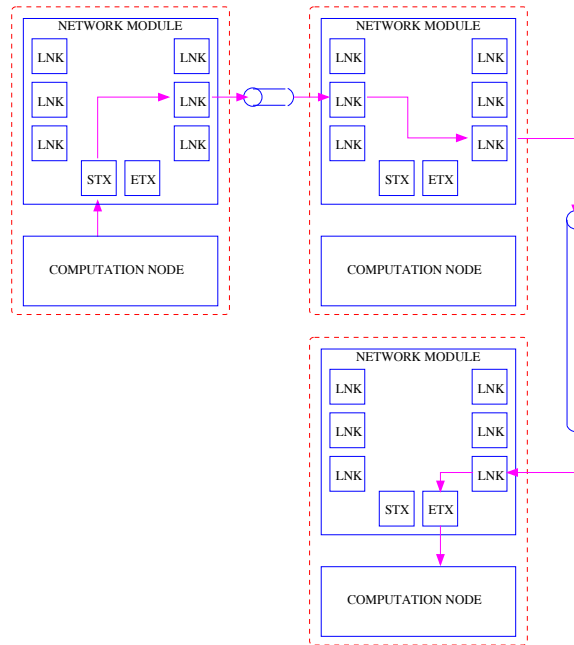


Figura 4.19: Una trasmissione ad L è un trasferimento remoto a distanza 2, come descritto nel testo.

Un punto importante preso in considerazione è l'affidabilità della trasmissione: la richiesta che una macchina di grandi dimensioni possa lavorare senza errori per periodi dell'ordine del giorno implica che la frazione di bit trasferiti con errore deve essere inferiore ad 1 ogni 10^{17} . Dal momento che questo obiettivo non è realistico (su APEmille si è misurato una incidenza di errori $\lesssim 10^{-15}$), ai dati è aggiunto un codice di controllo CRC (figura 4.20).

La correzione degli errori avviene secondo lo schema

1. Il pacchetto di dati in partenza viene diviso in segmenti più piccoli
2. A ciascun segmento viene aggiunto un codice calcolato con un algoritmo standard (cyclic redundancy check)
3. Il nodo ricevente esegue per suo conto il CRC sui dati e lo confronta con quello ricevuto
4. Se i due valori discordano, la trasmissione viene ripetuta.

4.10.6 Lo stack delle condizioni

Le condizioni logiche (maggiore, minore, uguale) possono essere valutate su grandezze floating point ed intere e portate nel blocco che nella figura 4.12 indicato con STK. Il blocco tiene traccia delle condizioni che vengono progressivamente calcolate

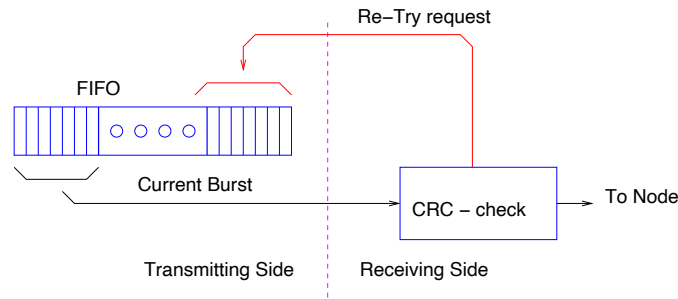


Figura 4.20: Il controllo di errore sul link.

e permette di combinarle per mezzo dei consueti operatori booleani *and*, *or* ecc. Lo stack deve il suo nome alla gestione *last in first out* delle condizioni. Il valore di verità posto in cima allo stack condiziona automaticamente le scritture nel register file e nella memoria.

In questo modo viene realizzato il costrutto che nei programmi ad alto livello è espresso dall'istruzione già menzionata *where()*. Il valore del top dello stack fornisce anche la condizione in base alla quale eventualmente controllare il flusso del programma (*if - then - else*). In apeNEXT il flusso del programma può essere potenzialmente diverso su ciascun processore, sebbene questa caratteristica non sia necessaria in programmi di fisica che seguono il modello di calcolo parallelo.

4.10.7 Le condizioni globali

La necessità di trasmettere una condizione locale di ogni nodo per poterne valutare la verità sull'intera macchina è stata discussa in una sezione precedente.

Il meccanismo cosiddetto di *if-any* ed *if-all* è realizzato per mezzo di quattro segnali che collegano ciascun nodo ad una gerarchia di livelli. Due delle quattro linee, un segnale di *verità out*, accompagnato dal rispettivo strobe, escono dal nodo. Due segnali sono entranti: *verità in* ed il relativo strobe (figura 4.21).

Il commutatore di segnale di livello superiore è sempre in attesa di ricevere lo strobe da uno qualsiasi dei nodi. Quando tutti gli strobe sono alti, i valori di v-out provenienti dai tutti nodi vengono messi in *and* tra di loro ed il risultato è posto sulla linea v-in; contemporaneamente viene alzato il valore dello strobe in ingresso. In questo modo i nodi possono sincronizzarsi e se desiderato il valore di v-in può essere messo sullo stack.

La rete descritta prevede altri segnali: un *kill* in ingresso ed uno in uscita ed un *trigger*. Il kill in uscita è sollevato in caso di eccezione⁶ e si propaga immediatamente lungo la gerarchia in modo da arrestare l'intera macchina. Il trigger ha un

⁶ Una eccezione è una condizione inattesa non gestita in altro modo, come una divisione per zero.

comportamento analogo, ma non arresta la macchina e sarà usato (ad esempio) per sincronizzare la partenza dei nodi quando uno solo sarà interfacciato con l'esterno e dovrà dare l'avvio agli altri.

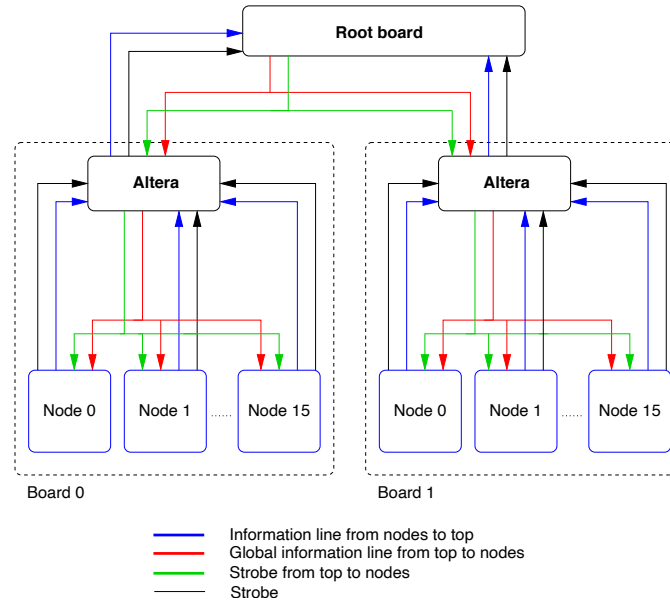


Figura 4.21: Gerarchia per la trasmissione di condizioni locali ed eccezioni.

4.10.8 La cache istruzioni

La tabella 4.3 raccoglie i valori discussi finora di larghezze di banda. Dai valori di w_f (potenza floating point) e w_l (velocità della memoria) si nota che la macchina è bilanciata per codici con $R < 4$. Sebbene il calcolo di R per l'operatore di Dirac abbia dato $R \simeq 7$, il margine non è largo come sembra, perché non sono state considerate le latenze della memoria, che possono portare ad un aumento di R del 30% circa. Questa cifra si ottiene dividendo un valore tipico della latenza di ~ 3 cicli per la dimensione media l'accesso, che per una matrice di gauge ed uno spinore è in media $(9 + 12)/2$ cicli. Di conseguenza gli accessi in memoria dati saturano l'intera banda disponibile.

Occorre a questo punto affrontare il problema di dove memorizzare il codice del programma. Se questo fosse mantenuto in memoria, la lettura precedente alla sua esecuzione richiederebbe una parte della banda verso la memoria. La banda è però già satura a causa degli accessi alla memoria dati e di conseguenza le prestazioni decadrebbero. (La lettura di un segmento di microcodice, per inciso, richiede un tempo pari a quello della sua esecuzione; il solo caricamento del microcodice già da solo quindi è sufficiente a saturare l'intera banda passante verso la memoria.)

La soluzione adottata è suggerita dal fatto che i programmi reali passano una

frazione estremamente grande del loro tempo in loop critici lunghi poche linee di microcodice. Vedremo, per esempio, che la sezione importante del codice che applica l'operatore di Dirac, per esempio, è composta da due loop annidati lunghi circa 400 linee, compresi gli overhead.

Esecuzione in modalità fifo

Alla memoria di ogni nodo è affiancata una cosiddetta *cache istruzioni* (figura 4.22). Il microcodice che compone il programma può essere eseguito dalla cache istruzioni senza cicli di latenza e senza interferire con gli accessi alla memoria dati. Motivi tecnici limitano la dimensione della cache a contenere un massimo di circa 4000 parole di microcodice. Questa dimensione è largamente sufficiente a contenere i kernel di calcolo più importanti, ma non è sufficiente a contenere un intero programma, che include anche parti relativamente poco pesanti ma di dimensioni notevoli, quali le inizializzazioni.

La politica di gestione della cache istruzioni è quella di *instruction look-ahead*. Questo meccanismo legge dalla memoria la parte di programma successiva al punto che il nodo sta eseguendo e la copia nella cache. Quando la cache ha raggiunto il limite della sua capacità, le porzioni di programma già eseguite vengono cancellate e sostituite da nuove. Questa modalità di esecuzione è detta *fifo mode* (il termine sfortunatamente coincide con quello usato per la coda di comunicazione). Il meccanismo di look-ahead è complesso ed in prima approssimazione viene gestito mediante tre puntatori: uno di lettura dalla memoria, uno di scrittura in cache ed uno di esecuzione (figura 4.23). I salti nel flusso del programma vengono realizzati in modalità fifo spostando il puntatore di *lettura* ad un nuovo indirizzo.

Dal momento che il look-ahead legge dalla memoria, i codici eseguiti in modalità fifo possono essere più lenti anche di un fattore 2 rispetto alla loro efficienza misurata sul microcodice. Il look-ahead che ricopia il codice nella cache istruzioni ha la priorità sugli accessi a dati: quando il look-ahead è richiesto contemporaneamente ad un accesso alla memoria dati, il programma viene arrestato e si dice che il nodo è in *sleep*.

Esecuzione in modalità cache

Il decadimento di prestazioni descritto è accettabile per le parti di programma come le inizializzazioni, ma non lo è sui kernel di calcolo. Per essi è prevista una diversa strategia di gestione detta *cache mode*: il kernel viene copiato una volta sola e per intero dalla memoria nella cache; poi l'esecuzione avviene interamente in cache fino a quando ci sono iterazioni da eseguire (figura 4.24). Il programmatore segnala con opportune istruzioni di controllo le parti di programma che vanno caricate in cache. Una volta copiato nella cache, la memoria è usata solo per accessi a dati, ed

il codice può girare in cache per un tempo arbitrariamente lungo. I salti nel flusso del programma sono realizzati spostando il puntatore di *esecuzione* all'indirizzo richiesto.

Se c è la frazione delle istruzioni di un codice che può essere eseguita in modalità cache, per l'intero programma ci si attende una efficienza di

$$\epsilon = \frac{1}{c + 2 \cdot (1 - c)}$$

Questa formula si ottiene considerando, in base a quanto detto nella sezione precedente, che ogni istruzione eseguita fuori dalla cache costi il doppio.

A scopo di illustrazione, calcoliamo l'impatto che ha il caricamento della cache assegnando dei valori realistici. Supponiamo che in un programma si usi un metodo iterativo per invertire l'operatore di Dirac, a campi U fissati, su un reticolo locale di $4^3 \times 8 = 512$ siti. Nei capitoli finali di questa tesi si mostra che una applicazione dell'operatore richiede circa 520 cicli per sito e che il codice che applica l'operatore è lungo circa 400 cicli.

Se assumiamo (in maniera alquanto pessimistica) che il codice venga ricaricato ad ogni iterazione, pagheremmo uno sleep di circa 400 cicli su un tempo di esecuzione di $512 \cdot 570 \simeq 3 \cdot 10^5$ cicli, che corrisponde ad una penalità del 2 per mille.

Questo costo trascurabile può essere ulteriormente ridotto praticamente a zero, perché il codice può essere conservato tra una iterazione e l'altra.

La compressione del microcodice

La dimensione della cache istruzioni è di 4096 parole; il kernel dell'operatore di Dirac è lungo ~ 500 parole, quindi la cache è largamente sufficiente a contenere uno o più core di calcolo.

Tuttavia l'analisi del microcodice di questi core rivela una notevole quantità di informazioni ridondanti. Questo ha suggerito l'adozione di uno schema di *compressione del microcodice*: il microcodice viene conservato sia in memoria che in cache non nella sua versione mostrata in figura 4.10, ma codificato in maniera da non ripetere i campi vuoti che compaiono su righe consecutive.

La compressione soddisfa due richieste:

1. Riduce la richiesta di banda passante tra memoria e cache. In questo modo un codice può girare in modalità fifo con una penalità di prestazioni inferiore al fattore 2 assunto sopra.
2. Permette di memorizzare maggiori quantità di microcodice in cache.

La compressione riduce la dimensione in memoria del codice dell'operatore di Dirac approssimativamente di un fattore 2.

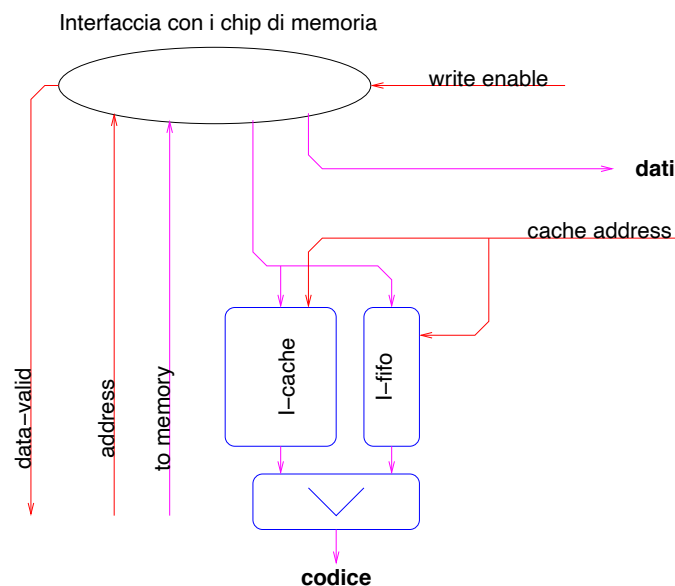


Figura 4.22: Schema a blocchi della cache istruzioni ed del relativo meccanismo di precaricamento.

La relazione tra cache, memoria e compressione è complicata ed ha richiesto una quantità di lavoro sia nello sviluppo che nella verifica del buon funzionamento. I programmi scritti dall'autore sono stati uno strumento importante perché dotati di una complessità realistica. Attualmente gli utenti che desiderano scrivere codici di calcolo in assembler sono isolati dalle complessità della gestione della cache da un opportuno gruppo di macroistruzioni, espanse da un programma di nome MPP. Questo preprocessore (scritto dall'autore insieme ad H. Simma) semplifica un certo numero di notazioni, quali ad esempio quella per la coniugazione di una combinazione arbitraria di operandi nelle operazioni normali, oppure il caricamento/cancellazione in cache dei core di calcolo.

4.10.9 Lo slow control

Abbiamo visto che ogni nodo può comunicare con l'esterno quando venga stabilito un opportuno percorso attraverso la macchina (figura 4.16). Il path che si stabilisce in questo modo è veloce ma potenzialmente di difficile impiego nelle condizioni di errore, perché richiede che tutti i nodi siano funzionanti e che la rete sia configurata nella forma “avvolta” descritta. Nel caso di arresti eccezionali (per esempio: divisione per zero) è desiderabile poter ispezionare il contenuto delle memorie del nodo che ha prodotto l'eccezione per comprenderne la causa del problema, ma questo potrebbe non essere possibile perché qualche nodo è rimasto in uno stato non consistente.

Per questo motivo ogni nodo possiede un secondo percorso per comunicare con

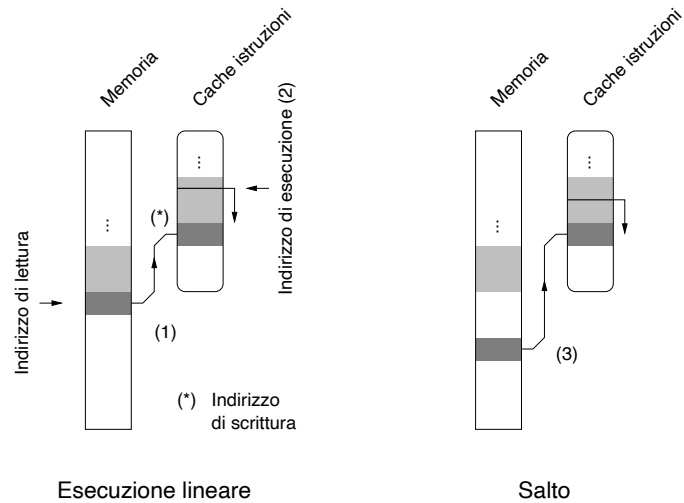


Figura 4.23: *Esecuzione con look-ahead*. La cache istruzioni viene riempita progressivamente (1) copiandovi il programma dalla memoria. Il microcodice viene letto dalla cache una parola alla volta ad indirizzi consecutivi (2) ed eseguito. Se il programma non prevede accessi in memoria, le due operazioni sono contemporanee. Nel caso che il flusso del programma preveda un salto, solo l'indirizzo di lettura viene modificato opportunamente (3), non l'indirizzo di scrittura né quello di esecuzione (*fifo jump*).

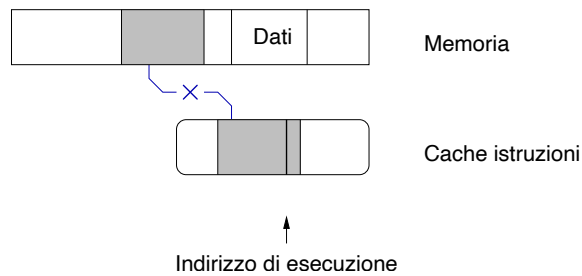


Figura 4.24: *Esecuzione in modalità cache*. La porzione critica di programma (*core*) viene copiata prima della sua esecuzione e per intero nella cache istruzioni. La sua esecuzione avviene senza richiedere ulteriori trasferimenti di codice dalla memoria. I salti del flusso del programma (p. es. per iterare sui siti) vengono ottenuti in questa modalità alterando la posizione del puntatore di lettura (*cache jump*).

l'esterno, l'*interfaccia I2C*. Questa interfaccia permette di comunicare con ogni nodo singolarmente, arrestarlo e riavviarlo ed inoltre ispezionarne le memorie ed i registri, indipendentemente dal suo stato.

L'accesso tramite I2C è molto lento rispetto a quello tramite link LVDS, ma questo non costituisce una limitazione visto che l'interfaccia non viene usata per trasferimenti massicci o a bassa latenza.

4.11 Conclusioni

Dagli esempi presentati dovrebbe essere evidente che la efficiente esecuzione di un particolare codice dipende essenzialmente da tre fattori:

1. Quali sono le unità hardware e quanto sono veloci.
2. La scrittura del codice in una forma che permetta alle unità indipendenti di funzionare contemporaneamente per quanto possibile.
3. La temporizzazione ottimale delle unità, cioè la generazione di un microcodice che traduce fedelmente il programma rispettandone le dipendenze.

Il primo punto lo fissano essenzialmente i progettisti e le tecnologie usate: le scelte fatte per apeNEXT sono state descritte in questo capitolo.

L'utente si deve fare carico del secondo punto: chi scrive programmi deve cercare di scrivere le formule contenute nei suoi programmi in una forma più simile all'equazione (4.4) che alla (4.3).

Il terzo punto, cioè la generazione del microcodice ottimale quando sia dato lo schema delle dipendenze, viene gestito in maniera automatica da un software chiamato *shaker*. Lo shaker esamina il programma e costruisce un grafo simile a quello di figura 4.7; il grafo viene quindi visitato ed ad ogni nodo viene assegnato un ciclo nel quale l'istruzione può essere avviata. Lo scheduling viene effettuato in modo da poter eseguire il maggiore numero possibile di istruzioni in uno stesso ciclo di clock⁷. Questo viene fatto non solo per i calcoli aritmetici (che sono il caso più facile) ma anche per ogni altro tipo di dipendenza, quale ad esempio la seguente sequenza tipica: calcolo di un indice discreto \rightarrow calcolo dell'indirizzo corrispondente \rightarrow accesso alla memoria \rightarrow copia del dato indicizzato nei registri, come vedremo nel capitolo successivo.

⁷ Questo problema è noto come *instruction scheduling* ed appartiene alla classe di problemi NP completi: ciò vuole dire essenzialmente che la soluzione ottimale può essere trovata solo per esaurimento, esaminando cioè tutte le possibili permutazioni di istruzioni.

Capitolo 5

Codice e misure

I capitoli precedenti hanno indicato che la maggior parte del tempo di calcolo per simulazioni di QCD, sia completa che quenched, nei prossimi anni sarà speso nella esecuzione di codici (kernel di calcolo) il cui scopo essenzialmente è applicare l'operatore di Dirac sul reticolo. Questi codici saranno scritti in modi diversi, ma prevedibilmente avranno caratteristiche simili al modello le cui prestazioni sono state analizzate in dettaglio nel lavoro di questa tesi. Questo capitolo descrive le misure che sono state eseguite, le commenta e ne trae le conseguenze.

5.1 Obbiettivi e metodo

L'architettura della macchina all'inizio è stata definita da considerazioni in parte qualitative come la considerazione dei parametri attesi R e ρ ed il confronto con analoghi progetti precedenti. Una volta fissate le caratteristiche principali è iniziato lo sviluppo dell'hardware del nodo di calcolo, tuttora in corso.

Riporto qui la forma scritta prima dell'operatore di Dirac sul reticolo (4.1), scrivendo per esteso gli indici di spin α e β , di colore i e j e di sito n ed n' . Le lettere greche sono indici che assumono i valori $0 \dots 3$ e quelle latine assumono i valori $1 \dots 3$:

$$D_{n,n'} = \delta_{n,n'} - K \sum_{\mu} \left\{ (\delta^{\alpha\beta} - \gamma_{\mu}^{\alpha\beta}) U_{n\mu}^{ij} \delta_{n+\hat{\mu},n'} + (\delta^{\alpha\beta} + \gamma_{\mu}^{\alpha\beta}) U_{n'\mu}^{ji*} \delta_{n-\hat{\mu},n'} \right\}.$$

Lo scopo del lavoro era misurare in maniera esatta l'efficienza del nodo di calcolo di apeNEXT su un codice che applica questo operatore, in anticipo rispetto alla costruzione della macchina. La scrittura e lo studio di questo codice ha prodotto questi tre risultati:

1. Ha fornito un banco di prova con cui assicurarsi che le modifiche apportate all'hardware durante lo sviluppo ne conservassero l'efficienza.

2. Ha costituito la prima esperienza di sviluppo di un codice realistico per la nuova architettura.
3. Ha dato delle indicazioni quantitative sull'efficienza della macchina su un tipico kernel di calcolo.

Il primo punto risponde all'esigenza di avere una forma di test la cui buona riuscita dipendesse dalla corretta funzionalità di molte parti della macchina. La descrizione dell'hardware durante lo sviluppo è stata soggetta a molte modifiche ed a volte questi cambiamenti hanno conseguenze non previste. Per questo il fatto che un codice relativamente complicato venga eseguito dal modello simulato e dia i risultati attesi è una assicurazione di notevole valore sul buon funzionamento dell'insieme.

Il secondo punto sarà particolarmente importante quando la macchina sarà disponibile ad un pubblico esperto ma che non ne conosce i dettagli. I linguaggi di alto livello che saranno usati dovranno permettere di praticare le strade che il presente studio ha mostrato essere più fruttuose.

Le valutazioni che sono emerse dal terzo punto sono presentate in questi capitoli.

5.2 Quantità misurate

La domanda che in ultima analisi si pone l'autore di un codice è *quanto tempo impiegherà la macchina ad eseguire questo programma?* La risposta alla domanda dipende dai seguenti fattori:

- Quanto durano le fasi di inizializzazione
- Quante volte viene eseguito ciascuno dei *core* di calcolo
- Quante operazioni aritmetiche esso contiene
- Quante parole di microcodice è lungo ciascuno di questi core
- Quanti cicli di fifo stretch vengono causati dalle comunicazioni remote
- Quanti cicli di memory stretch sono dovuti alla latenza delle memorie

Per le ragioni discusse noi assumeremo di avere un solo core di calcolo, dal quale estrarremo tutti i parametri elencati. Le fasi di inizializzazione verranno trascurate, perché nella pratica anche esse hanno un peso del tutto trascurabile. Gli altri valori saranno studiati in dettaglio e questa sezione descrive i parametri che sono stati misurati per quantificarli.

Lo sviluppo del programma è avvenuto in parallelo a quello del modello del nodo. Lo stato di questo modello ha determinato le misure che potevano essere prese su ogni versione del codice.

5.2.1 L'efficienza del microcodice

La misura della lunghezza di una porzione di microcodice è essenziale, perché essa è proporzionale al tempo impiegato dalla macchina per eseguirlo, in condizioni di memoria e rete “ideali”. Il microcodice dà una rappresentazione in qualche modo intuitiva della sequenza di stati in cui passa la macchina durante l'esecuzione; in assenza di cause di stretch, esso viene eseguito al ritmo di una riga per ciclo di clock (5 ns).

Durante lo sviluppo della prima versione del codice lo stato dell'hardware non era tale da permetterne l'esecuzione. Nondimeno è stato possibile ottenere il microcodice corrispondente e da esso ricavare le due grandezze che compaiono nella quantità c definita come

$$c = \frac{\text{numero di operazioni normali}}{\text{lunghezza del microcodice}} = \frac{N}{l}$$

La grandezza c è sempre minore di uno perché l'unità floating point è in grado di iniziare al più una istruzione normale per ciclo; la stima di c include gli effetti delle dipendenze (ed è equivalente alla definizione di riempimento della pipe data nel capitolo precedente), ma trascura i tempi morti dovuti all'attesa di dati dalla rete e dalla memoria.

È possibile ottenere entrambe le grandezze contando il numero di righe rilevanti nella porzione di microcodice che itera \mathcal{D} su un sito. Il codice contiene due loop uno interno all'altro, sull'indice spaziale (quello esterno) e su quello temporale (quello interno). La parte di codice compresa nel loop interno viene eseguita $L^3 \cdot T$ volte, mentre la parte compresa solo nel loop esterno viene eseguita T volte. La formula data sopra va quindi precisata; è possibile definire due misure di riempimento della pipe: quella del loop interno

$$c_i = \frac{\text{operazioni}}{\text{dimensione del loop interno}} = \frac{N}{l_i}$$

e quella efficace

$$c_{eff} = \frac{\text{operazioni}}{\text{dimensione efficace dei due loop}} = \frac{N}{l_e},$$
$$l_e = l_i + \frac{l_o - l_i}{T}$$

dove l_i indica il numero di parole di microcodice che compongono il loop interno ed l_o quello del loop esterno.

5.2.2 L'efficienza della rete

Quando il modello è stato in grado di eseguire il codice si è simulata una macchina composta da un nodo i cui link la richiudevano su se stessa. In questo modo si

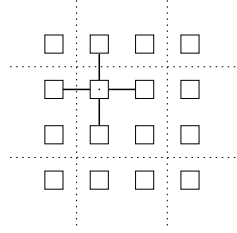


Figura 5.1: *Reticolo locale* $2^3 \times 4$. L'aggiornamento di ogni sito richiede tre comunicazioni remote in direzioni ortogonali. Le condizioni periodiche al contorno fanno sì che ogni nodo acceda a repliche di se stesso. Gli assi z e t non sono mostrati; l'asse t è locale al nodo.

è assunto un reticolo interamente locale $2^3 \times 4$, ma il cui aggiornamento richiede l'accesso a tre primi vicini per ogni sito (figura 5.1). Questa scelta permette di stimare l'impatto delle comunicazioni remote senza necessità di simulare più di un nodo.

Il modello della rete è realistico perché include ritardi e possibili interferenze sulle trasmissioni; di conseguenza è possibile definire il parametro

$$f = 1 - \frac{\text{cicli di attesa per fifo}}{\text{totale cicli}} = 1 - \frac{F}{C}$$

La quantità al numeratore è il numero di cicli in cui il nodo è rimasto fermo a causa dell'attesa di dati non ancora arrivati nella coda di ricezione (fifo stretch); al denominatore compare il numero di cicli di 5 ns richiesti per eseguire le iterazioni su tutti i siti. Entrambi questi valori possono essere ottenuti osservando i risultati delle simulazioni. Nel caso di un reticolo interamente locale e senza accessi remoti si misura $f = 1$ perché il passaggio dalla fifo locale è pressoché immediato. Il numero di accessi rispettivamente locali e potenzialmente remoti contenuti in una iterazione del codice è raccolto nella tabella 5.1.

	Dim. \times 128 bit	Locali	Poss. remoti	Scritture
ϕ	12	3	6	1
U	9	5	3	—

	Locali	Remoti	Totale
Accessi	8	9	17
Parole	81	99	180

Tabella 5.1: Numero e dimensione degli accessi alla memoria per sito.

5.2.3 L'efficienza della memoria

La memoria del nodo è stata inserita nel modello in maniera realistica solo in un secondo momento; per questo durante lo studio iniziale non è stata disponibile una stima accurata dell'impatto dei memory stretch. Quando il modello è stato perfezionato, questi stretch sono stati misurati con un parametro simile al precedente

$$m = 1 - \frac{\text{cicli di attesa per memoria}}{\text{totale cicli}} = 1 - \frac{M}{C}$$

Il valore al numeratore è il numero di cicli durante i quali l'unità aritmetica è stata bloccata a causa di una latenza lunga (memory stretch) ed è un dato di uscita della simulazione. Questi ritardi hanno una causa tecnologica: il tempo impiegato per fornire un dato dipende in maniera complicata dall'indirizzo richiesto, da quelli precedenti e dai tempi in cui sono avvenute le rispettive richieste. La quantità al denominatore è la stessa usata nella definizione precedente.

5.2.4 L'efficienza netta

Nei casi in cui si è potuta fare la stima dell'efficienza nel modo più realistico possibile, misurando allo stesso tempo i parametri c , r ed m , verrà dato anche il valore della efficienza netta

$$\eta = \frac{\text{numero di operazioni normali}}{\text{totale cicli}} = \frac{N_{tot}}{C}$$

È interessante notare che il numero di cicli C che compare nelle espressioni precedenti può essere misurato dall'esito delle simulazioni oppure può essere ottenuto come somma del numero efficace di parole contenute nel microcodice, dei memory stretch e dei fifo stretch. Questa formula è valida perché le due cause di stretch non avvengono mai contemporaneamente.¹

Come conseguenza, un modo indipendente di calcolare η quando siano note le efficienze parziali c , r ed f è

$$\eta_{eff} = c_{eff} \cdot f \cdot m$$

Il valore di η_{eff} è stato confrontato con η per verificare la consistenza delle misure. In tutti i casi si è verificato $|\eta - \eta_{eff}| \leq 0.01$. L'1% è una ragionevole stima dell'errore sulle misure effettuate.

È utile a questo punto definire per analogia:

$$\eta_i = c_i \cdot f \cdot m$$

Questa formula dà una misura dell'efficienza della *sola* parte interna di codice, quella che contiene i calcoli in virgola mobile. La parte rimanente è utilizzata per calcolare indici, decidere se l'aggiornamento del reticolo è terminato e caricare puntatori a siti vicini; questa parte prende il nome di *overhead*. Il peso dell'overhead può in generale essere ridotto con un opportuno lavoro di ottimizzazione. Il confronto tra

¹ Dal punto di vista dell'efficienza, ogni ciclo di clock per la macchina corrisponde ad uno ed uno solo di questi quattro stati: 1. memory stretch, 2. fifo stretch, 3. esecuzione, ma senza la produzione di risultati in virgola mobile, 4. fine del calcolo di una operazione normale. Massimizzare l'efficienza vuol dire evidentemente ridurre il numero di cicli di tipo 1., 2. e 3.

η_i ed η dà la misura di quanto guadagno (al massimo) si potrebbe ottenere da questo lavoro.

5.2.5 Il tempo del calcolo

Il parametro finale al quale siamo interessati è il tempo impiegato per calcolare l'operatore sul reticolo assegnato. Questo tempo è dato dal parametro C introdotto in precedenza, moltiplicato per la durata del ciclo di clock, che è 5 ns. Il tempo così calcolato non include le fasi di inizializzazione.

La stima ottenuta in questo modo è esatta, nel senso che la simulazione rispecchia ciclo per ciclo quelli che saranno i tempi di calcolo della macchina reale.

5.2.6 Il campo di riferimento

In ciascun caso ci si è assicurati del fatto che i programmi producessero i risultati attesi. Il reticolo è stato riempito con campi di riferimento presi per semplicità come segue.

$$(U_{test})_{\mu,n}^{ij} = \begin{pmatrix} a & b & b \\ b & a & b \\ b & b & a \end{pmatrix} \quad (\phi_{test})_n^{s,i} = \delta_{0i}\delta_{0s}$$

con $a = 15, b = 2 + 2i$. La matrice U non è ortogonale, ma ciò non era rilevante perché lo scopo era testare la correttezza dell'aritmetica complessa ed il tempo di calcolo ed entrambi sono indipendenti dai valori degli operandi. Il risultato della applicazione di D a questo campo è noto (non lo riporto); detto risultato ed è stato usato per verificare la corretta esecuzione dei codici.

5.3 Gli strumenti a disposizione

La verifica del codice scritto ha richiesto l'uso di un certo numero di programmi di appoggio, che menziono brevemente:

Il simulatore funzionale. Nei primi mesi dello sviluppo il codice è stato provato su questo programma, che simula in qualche misura il comportamento del nodo di calcolo. La simulazione è *funzionale* nel senso che riproduce solo i risultati dei calcoli, non la successione di stati attraverso cui passano le varie unità della macchina per ottenerli. Il simulatore funzionale è stato abbandonato in favore di quello hardware quando il modello dettagliato è stato perfezionato abbastanza da sostituirsi a quello funzionale.

Lo shaker. L'importanza di questo strumento è stata accennata in precedenza. Esso traduce la lista delle operazioni (incluse quelle aritmetiche), fornita dall'utente in un linguaggio molto semplice, nel microcodice equivalente che la macchina eseguirà al ritmo di una parola per ciclo di clock (in mancanza di cause di stretch).

Il simulatore dell'hardware. I progettisti esprimono il funzionamento richiesto della macchina in un linguaggio formale noto come VHDL². La descrizione può essere trasformata in ogni momento nell'insieme di circuiti che la realizza, oppure se ne può simulare il comportamento. La simulazione VHDL riproduce fedelmente il funzionamento del nodo e di ogni segnale interno.

5.4 Il programma

Il codice scritto realizza il prodotto dell'operatore lineare dato in (4.1) per un vettore ϕ_n^s posto in memoria; le matrici $U_n^{ss'}$ sono anche esse prelevate da indirizzi (contigui) della memoria. Il prodotto è memorizzato in un vettore risultato $\phi_n'^s$ analogo a quello di ingresso.

Il codice compie $T \times L^3$ iterazioni su altrettanti siti di un quadrispazio di dimensioni arbitrarie. Ad ogni sito è associata una coppia di indici (t, p) ; i siti sono calcolati iterando su $t = 0 \dots (T - 1)$ a fissate coordinate spaziali p , per poi passare a $p \rightarrow p + 1$ e di nuovo iterare su t . L'applicazione dell'operatore è completa dopo il calcolo del sito $t = (T - 1)$, $p = (L^3 - 1)$ e l'algoritmo ha termine.

Ad ogni sito sono associate 12 grandezze complesse di indice $s = 0 \dots 11$, dovute ai tre colori per le quattro componenti di spin. Le dimensioni del reticolo $n = 0 \dots (TL^3 - 1)$ sono arbitrarie; durante le prove le abbiamo mantenute a 4×2^3 in modo che i tempi delle simulazioni rimanessero entro limiti ragionevoli (il simulatore a disposizione impiega circa 1 minuto per sito).

D'ora in avanti sarà adottata la convenzione di indicare con l'indice zero la prima componente di ogni vettore. Questa convenzione è conveniente ed è quella usata consistentemente nel programma; essa come è noto è comoda perché l'indirizzo di un elemento in memoria, $\text{ind}(\phi_n)$, è calcolato come $\text{ind}(\phi) + n \cdot \text{dim}(\phi)$. La disposizione dei campi in memoria al variare di tutti gli indici è mostrata in dettaglio in figura 5.2. La coordinata spaziale e quella temporale del sito che si sta aggiornando e dei vicini in ogni direzione sono conservate come interi in alcuni registri, che sono indicati in tabella 5.2. Nella seconda colonna è scritto il nome con cui (per chiarezza) quel numero di registro viene indicato nel programma.

Il codice è stato scritto in più versioni, a seconda degli strumenti a disposizione e della variante che si voleva realizzare. Esso comprende anche un certo numero di parti necessarie ma non direttamente collegate al calcolo, quali l'inizializzazione dei vettori ϕ ed U e delle tabelle degli indici ai siti primi vicini. Queste parti sono descritte brevemente in appendice. Tutte le versioni del codice possono essere ottenute all'indirizzo <http://pcape2.pi.infn.it/cgi-bin/cvsweb.cgi/DiracNext>.

² Il linguaggio è standardizzato dall'istituto IEEE e la sigla sta per *Very high speed integrated circuit Hardware Description Language*.

Indice	Nome nel codice	Registro (dec.)
Spaziale (p)	tr_s	20
Temporale (t)	tr_t	22
$p + \hat{x}$	tr_xp1	24
$p - \hat{x}$	tr_xm1	27
$p + \hat{y}$	tr_xp2	25
$p - \hat{y}$	tr_xm2	28
$p + \hat{z}$	tr_xp3	26
$p - \hat{z}$	tr_xm3	29
$t + 1$	tr_tmp	30
$t - 1$	tr_tmp	30

Tabella 5.2: Uso dei registri-indice.

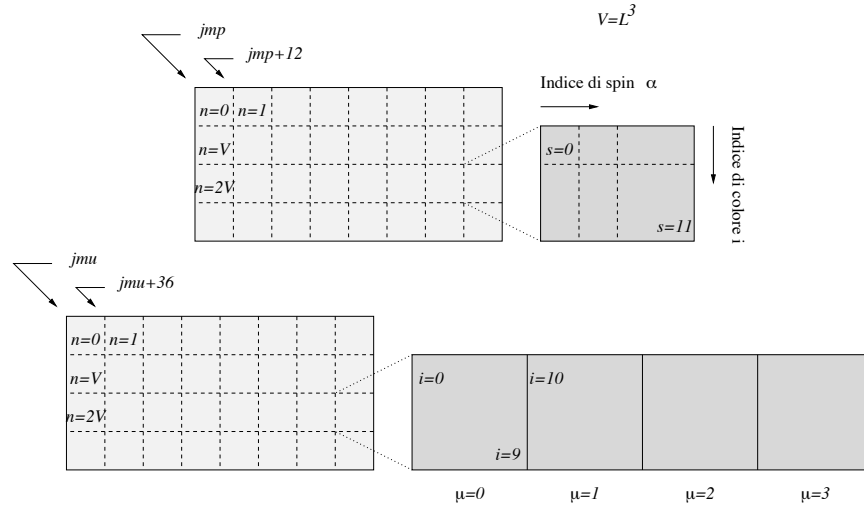


Figura 5.2: Disposizione dei campi $\phi_{p,t}^{\alpha i}$ (in alto) ed $U_{\mu,n}^{ij}$ (in basso) nella memoria.

Il calcolo, come si è visto, è ricondotto ad operazioni normali tra registri e gli operandi sono negati o coniugati a seconda della necessità. Nel codice è adottata la seguente rappresentazione delle matrici γ (la metrica dello spazio è euclidea a causa della formulazione nel tempo immaginario):

$$\gamma_x = \begin{pmatrix} 0 & 0 & -i \\ 0 & i & 0 \\ i & 0 & 0 \end{pmatrix} \quad \gamma_y = \begin{pmatrix} & & -1 \\ & +1 & \\ -1 & +1 & \end{pmatrix}$$

$$\gamma_z = \begin{pmatrix} & -i & \\ i & & i \\ & -i & \end{pmatrix} \quad \gamma_t = \begin{pmatrix} & & -1 \\ -1 & & -1 \\ & -1 & \end{pmatrix}$$

Inoltre nel programma il numero di operazioni da eseguire viene ridotto fattorizzando le matrici $(1 \pm \gamma)$ nella forma AB , come mostro solo per γ_x :

$$\begin{aligned} (\gamma_x - 1) &= \begin{pmatrix} -1 & 0 & 0 & -i \\ 0 & -1 & -i & 0 \\ 0 & i & -1 & 0 \\ i & 0 & 0 & -1 \end{pmatrix} = \\ &= \begin{pmatrix} -1 & 0 \\ 0 & -1 \\ 0 & i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & i \\ 0 & 1 & i & 0 \end{pmatrix} = A \cdot B \end{aligned}$$

Questa forma è utile perché le matrici di gauge U commutano con B (come pure con γ ed A): le prime agiscono sugli indici di colore, le seconde su quelli di spin e quindi vale

$$(\gamma - 1)U\phi = AB U\phi = AU B\phi$$

L'ultima forma è quella che è più conveniente da calcolare numericamente, perché prima si forma il prodotto $\psi = B\phi$, che ha due componenti $(\psi_a, \psi_b)^t$, poi le si moltiplica per U . Il vantaggio consiste nel fatto che è sufficiente eseguire due prodotti riga per colonna $U\psi_a$ ed $U\psi_b$ anziché quattro $U\phi_0 \dots U\phi_3$. I prodotti per A e B sono semplici perché le matrici sono costanti e non occorre eseguire le moltiplicazioni con gli elementi nulli.

Nelle sezioni successive presenterò le misure che sono state realizzate. I valori ottenuti hanno guidato l'analisi del codice corrispondente per scoprire quale fosse la limitazione maggiore. L'obiettivo di rimuovere i vincoli scoperti ha guidato lo sviluppo delle varianti successive. Le pagine seguenti descriveranno una serie di passi che hanno portato a codici di efficienza crescente.

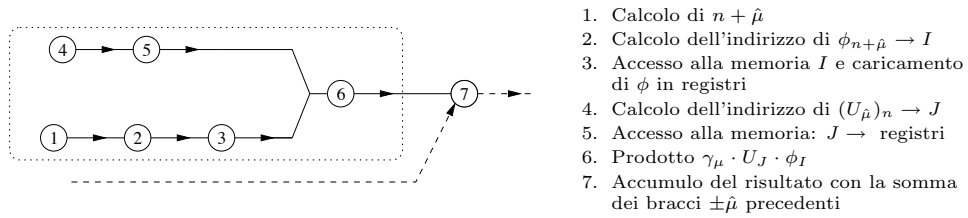


Figura 5.3: Schema delle dipendenze per un braccio di \mathcal{D} .

5.5 La prima versione: comunicazioni sincrone

La prima versione realizza il calcolo dell'operatore in maniera diretta: lo schema delle dipendenze che ne risulta è dato (in maniera semplificata) in figura 5.3. Tutti gli accessi in memoria sono fatti nel modo più generale, cioè passando per una delle fifo. Tutte le copie di memoria in registri sono espresse nel microcodice come coppie MTQ-QTR (memory to queue, queue to register).

Il programma è stato trasformato nel microcodice corrispondente ed il risultato è stato analizzato, con i risultati mostrati nella tabella 5.3. Al momento dello sviluppo il modello dell'hardware era dotato di un memory controller approssimato e non permetteva di leggere i valori degli stretch: non è stato possibile misurare le efficienze f ed m .

I valori misurati vanno migliorati in due direzioni:

- Il microcodice potrebbe essere reso più compatto
- Un uso più efficace della rete

Nelle sezioni successive verrà illustrato il lavoro fatto per risolvere entrambi i problemi.

operazioni (per sito)	registri usati	dimensione microcodice			
N		c_i	c_{eff}	l_i	l_o
342	184	0.48	0.46	716	816

Tabella 5.3: *Risultati della versione naif.* Questi sono i risultati delle misure sul programma che realizza il calcolo dell'operatore di Dirac, nella sua versione senza alcuna ottimizzazione.

5.6 Utilizzo della AGU

Il valore di c_{eff} misurato è inferiore a 0.5. Questo vuol dire che l'unità aritmetica era attiva solo nella metà delle parole di microcodice disponibili. Il microcodice

corrispondente al programma descritto al punto precedente è stato analizzato per scoprire quali dipendenze fossero causa della sua lunghezza.

La causa dell'inefficienza è stata rintracciata nel metodo usato per calcolare l'indirizzo di memoria dal quale prelevare i campi. In base a quanto mostrato nella figura 5.2, questo calcolo è (prendo come esempio il ϕ centrale):

$$\text{Indirizzo}(\phi_{t,p}) = \text{Base} + 12 \times (L^3 \cdot t + p) \quad (5.1)$$

Una volta calcolato, questo indirizzo viene posto sul bus indirizzi della memoria e vengono prelevati 12 valori consecutivi che vengono depositati nella coda (nel caso di un sito potenzialmente remoto) oppure direttamente in registri. La figura 5.4 mostra la parte di microcodice corrispondente al calcolo; tutti i valori che vi compaiono, inclusi i numeri di registro, sono in base esadecimale.

Uno dei passaggi intermedi della somma (5.1) prevede la moltiplicazione dei registri 32 (che contiene il valore 12) e 20 (l'indice di sito p). La moltiplicazione viene comandata dal codice **RXR**. Il risultato viene scritto nel registro 32 mediante il codice **A2RF** (1).

Il risultato di una operazione **RXR** è disponibile due cicli dopo l'invio del codice stesso; ciò equivale a dire che in condizioni di scheduling ottimale il codice **A2RF** potrebbe comparire nel microcodice due righe sotto il codice **RXR**. La figura evidenzia però che il campo di microcodice corrispondente alla prima colonna è occupato da un altro comando, **Q2RF** (2); quindi lo scheduling di **A2RF** è stato ritardato (3) fino al primo ciclo seguente disponibile.

Il codice **Q2RF** è in realtà una copia da coda a registro e fa parte dell'accesso precedente alla memoria. Questo vuol dire che un accesso in memoria ed il calcolo dell'indirizzo del successivo si contendono la stessa risorsa (il campo **AGUC** del microcodice) e non possono avvenire in contemporanea. Questa situazione è gravemente penalizzante perché impone la sequenza temporale stretta: calcolo indirizzo \rightarrow inizio accesso \rightarrow scrittura nei registri \rightarrow calcolo indirizzo successivo. Il calcolo dell'indirizzo in assenza di conflitti impiega circa 20 cicli di microcodice; l'inizio dell'accesso ne impiega circa 10; il caricamento dei registri 9 o 12. Se nessuna di queste operazioni si sovrappone alle altre, il numero di cicli di microcodice necessari per una iterazione sarà approssimativamente almeno $20 \times 17 + 10 \times 17 + 180 = 690$, che è consistente con quanto misurato nella sezione precedente.

La parte di codice che calcola l'indirizzo è stata riscritta in modo da non richiedere l'uso di registri temporanei. Questo è possibile perché la AGU ha un registro interno detto *feedback* che conserva l'ultimo valore calcolato. Indicando con A l'accumulatore, il calcolo (5.1) è stato scritto come

$$\begin{aligned}
A &\leftarrow 12 \cdot p \\
A &\leftarrow 12 \cdot t \cdot L^3 + A \\
\text{Indirizzo} &\leftarrow \text{Base} + A
\end{aligned}$$

La modifica ha avuto l'effetto desiderato ed ha parallelizzato il calcolo dell'indirizzo con i load remoti e locali; la tabella 5.4 mostra il risultato ottenuto.

operazioni (per sito)	registri usati	dimensione microcodice			
N		c_i	c_{eff}	l_i	l_o
342	184	0.65	0.62	529	629

Tabella 5.4: *Risultati della seconda versione.* Questa versione del programma ha prestazioni migliori della precedente perché i calcoli degli indirizzi sono stati resi paralleli agli accessi ai dati.

5.7 Riduzione dei fifo stretch

La modifica descritta nella sezione precedente ha aumentato il riempimento della pipe aritmetica. L'esecuzione però è affetta dal problema delle comunicazioni sincrone: le trasmissioni memoria \rightarrow coda e le ricezioni coda \rightarrow registri avvengono sempre in coppia.

Questo comporta due svantaggi notevoli.

- Non si attende un tempo sufficiente a permettere ai dati remoti di transitare sul link. Nel momento in cui avviene la richiesta di lettura della coda, la macchina viene fermata fino a quando i dati non arrivano (figura 5.5).
- Non partono mai due trasmissioni consecutive, quindi non viene usata la caratteristica della rete di eseguire contemporaneamente le trasmissioni in direzioni diverse.

Stimiamo qui l'ordine di grandezza dei valori che non si sono potuti misurare in maniera diretta. L'arresto per fifo stretch in una comunicazione remota sincrona ha una durata dell'ordine di 200 cicli; da questo si può dedurre un fifo stretch per l'intero reticolo di circa $F = 4 \times 200 \times 32 \sim 25000$ cicli. Il fattore 4 è il numero medio di accessi remoti per sito e si può ottenere (molto approssimativamente) ad esempio dividendo 17 accessi remoti per il valore di $\rho(n = 2) \simeq 4$ calcolato nell'equazione (4.2).

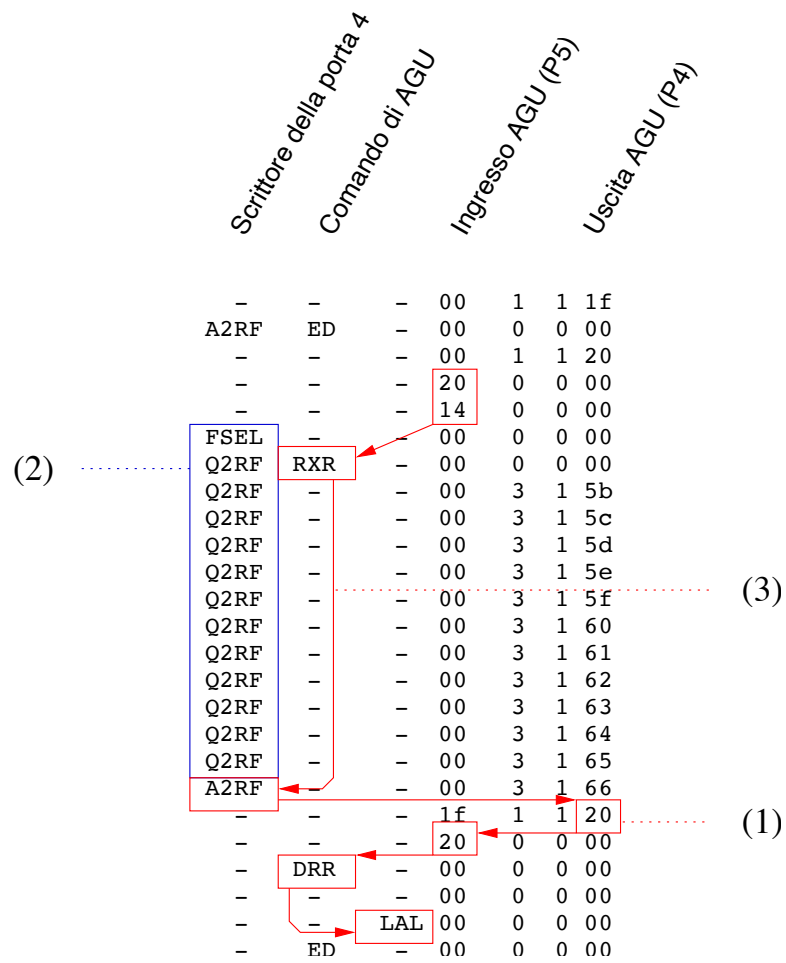


Figura 5.4: La causa dell'inefficienza nella prima versione descritta.

La durata di questo stretch è maggiore del numero di cicli di microcodice utili $529 \times 32 \simeq 17000 = C - F$. Da queste cifre si può stimare molto approssimativamente una efficienza di rete

$$f \sim 1 - \frac{F}{C} \simeq 0.40$$

Il problema è stato risolto; la soluzione elaborata consiste nel caricare ad ogni iterazione non i dati remoti che occorrono per calcolare il sito corrente, ma bensì quelli che serviranno al *prossimo*; i dati sono lasciati nella coda in attesa di essere recuperati alla successiva iterazione. Il meccanismo è illustrato nella figura 5.6 ed è chiamato *prefetch*.

La tecnica del prefetch è stata realizzata in un codice e le prestazioni sono state misurate nel simulatore. Il codice è stato caricato nella cache istruzioni. La tabella 5.5 mostra in dettaglio i risultati raccolti. È evidente che il prefetch ha avuto l'esito sperato ed ha eliminato quasi interamente i fifo stretch: $f \simeq 1$.

operazioni (per sito)	registri usati	dimensione microcodice			
N		c_i	c_{eff}	l_i	l_o
342	186	0.79	0.77	433	482

operazioni (totale)	totale cicli	stretch	
N_{tot}	C	fifo F	memoria M
10944	18607	430	3812

	η_i	η_{eff}	f	m	η
Prefetch	0.62	0.60	0.98	0.80	0.59

Tabella 5.5: *Risultati delle misure sul programma che realizza il calcolo dell'operatore di Dirac con prefetch.* L'uso delle code come area di appoggio, descritto nel testo, permette di nascondere la latenza delle comunicazioni remote.

5.8 Il prefetch completo

Il codice della versione con prefetch prevede due percorsi per i dati: quelli potenzialmente remoti subiscono il precaricamento attraverso la coda; quelli sicuramente locali vegono trasferiti direttamente dalla memoria ai registri. Questa scelta è apparentemente conveniente perché il passaggio diretto da memoria locale a registro è di alcuni cicli più rapido che il passaggio attraverso la coda locale.

Questo effetto è stato valutato con una quarta versione, che trasferisce tutti i dati, anche quelli locali, attraverso la coda tramite prefetch. I risultati ottenuti

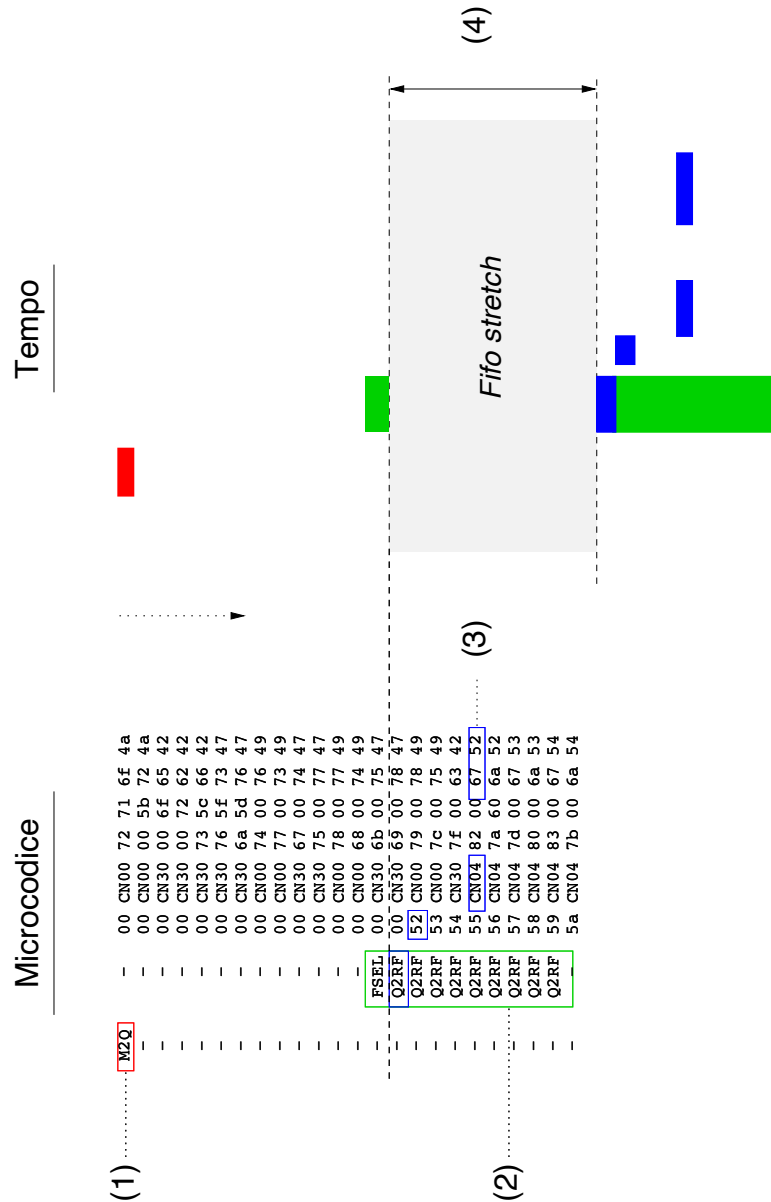


Figura 5.5: Le coppie MTQ-QTR impediscono la sovrapposizione tra calcoli e comunicazioni. La ricezione dei dati (2) è stata temporizzata 14 cicli dopo la trasmissione (1). Questa distanza è sufficiente a fare transitare i dati, nel caso in cui provengono dallo stesso nodo (coda locale). Il primo dato arrivato è scritto nel registro 52 e viene utilizzato appena possibile nei calcoli aritmetici (3). Quando il dato proviene da una coda remota, invece, 14 cicli non sono sufficienti a farlo giungere a destinazione. Per questo, l'istruzione Q2RF blocca il nodo con un fifo stretch (4) fino a quando i dati non sono arrivati. Il fifo stretch è stato ridotto con la tecnica del prefetch descritta nel testo.

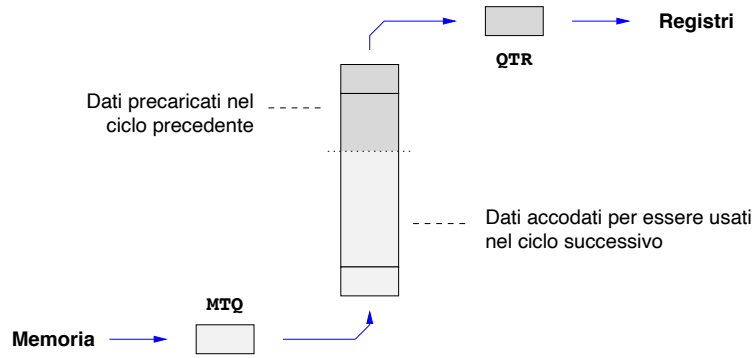


Figura 5.6: *Il prefetch*. La figura mostra una “istantanea” del contenuto di una coda durante il calcolo dell’operatore su un sito. I dati che serviranno al sito successivo vengono scritti in fondo alla FIFO. Contemporaneamente è possibile estrarre dalla cima della coda i dati che servono a calcolare la iterazione corrente. La lettura e la scrittura possono avvenire in contemporanea: la coda le disaccoppia.

sono riassunti nella tabella 5.6: l’efficienza netta è ulteriormente migliorata dal 59 al 66%.

Il risultato si spiega perché il prefetch ha un vantaggio ulteriore a quelli descritti. Il calcolo l’accesso in memoria ed il trasferimento nei registri vengono disaccoppiati nel tempo e possono anche avvenire nell’ordine inverso³. In questo modo lo shaker ha una ampia libertà nello scegliere la temporizzazione ottimale senza dover rispettare l’ordine causale attivazione della memoria \rightarrow scrittura in registri. La rimozione di questa dipendenza permette la generazione di un microcodice più compatto; dalle tabelle si vede che l’incremento dell’efficienza netta η (+7%) è dovuto per la maggior parte all’aumento della densità del microcodice c_{eff} (+5%).

In questa versione del codice il numero di fifo stretch è comprensibilmente aumentato, il numero di stretch di memoria è diminuito ed il prodotto $f m$ è rimasto circa costante. Questo dipende dal fatto che la fifo ed il memory controller lavorano indipendentemente. Quando tutti gli accessi passano attraverso la coda, alcuni cicli che prima erano classificati come memory stretch saranno ora contati come fifo stretch, ma la somma del numero dei due sarà approssimativamente invariata.

La tabella 5.7 mostra i risultati dello stesso codice, quando tutti gli accessi sono locali. La formula (4.2) per $\rho(n)$ mostra che il numero di accessi remoti diminuisce con l’aumentare della dimensione del reticolo. I valori ottenuti nel caso senza accessi remoti, quindi, sono interessanti perché rappresentano il limite asintotico per reticoli molto grandi: $n \rightarrow \infty \Rightarrow \rho \rightarrow \infty$.

³ Si ricordi che nei registri vanno i dati che erano già stati spediti nel ciclo precedente.

operazioni (per sito)	registri usati			dimensione microcodice	
N		c_i	c_{eff}	l_i	l_o
342	186	0.85	0.82	400	459

operazioni (totale)	totale cicli	stretch	
N_{tot}	C	fifo F	memoria M
10944	16631	1495	1636

	η_i	η_{eff}	f	m	η
Prefetch completo	0.70	0.67	0.91	0.90	0.66

Tabella 5.6: Il prefetch di tutti i dati aumenta ulteriormente l'efficienza.

5.8.1 Il register windowing

Il *register windowing* è stato introdotto nel CP-PACS per permettere a calcoli e caricamenti dalla memoria di avvenire contemporaneamente. Lo schema di funzionamento tipico di un kernel di calcolo, incluso quello che si sta studiando in questo lavoro, segue approssimativamente lo schema semplificato dato in figura 5.7.

Lo scopo del windowing è di anticipare il più possibile i calcoli eliminando la dipendenza dal load eseguito nello stesso ciclo (figura 5.8). Il risultato è ottenuto usando due set di registri che vengono scambiati alla fine di ogni ciclo.

Il prefetch è in qualche modo alternativo al windowing e ne dà lo stesso risultato, come è mostrato in figura 5.9.

5.8.2 L'importanza del prefetch

Dalla analisi sono emersi i vantaggi che comporta la tecnica del prefetch, che si è rivelata quindi uno strumento molto potente:

1. Nasconde la latenza dei trasferimenti remoti.
2. Permette ai link di funzionare contemporaneamente.
3. Permette di caricare molto rapidamente i valori nei registri ed iniziare i calcoli quasi all'inizio dell'iterazione.
4. Permette l'inversione dell'ordine tra l'avvio di una operazione di memoria e la scrittura nel register file; questo a sua volta permette di eseguire uno scheduling più compatto.

5.9 Riduzione dei memory stretch

Nell'ultima versione discussa la fifo e la memoria sono responsabili di una perdita di efficienza del 10% ciascuna circa. Ci si aspetta che su reticoli di dimensioni più

operazioni (per sito)	registri usati	dimensione microcodice			
N		c_i	c_{eff}	l_i	l_o
342	186	0.85	0.82	400	459

operazioni (totale)	totale cicli	stretch	
N_{tot}	C	fifo F	memoria M
10944	15193	0	1653

	η_i	η_{eff}	f	m	η
Prefetch completo (reticolo locale)	0.76	0.73	1.00	0.89	0.72

Tabella 5.7: *Prefetch completo, nel caso di un reticolo interamente locale.* Questi valori sono il limite asintotico delle efficienze per reticoli molto grandi.

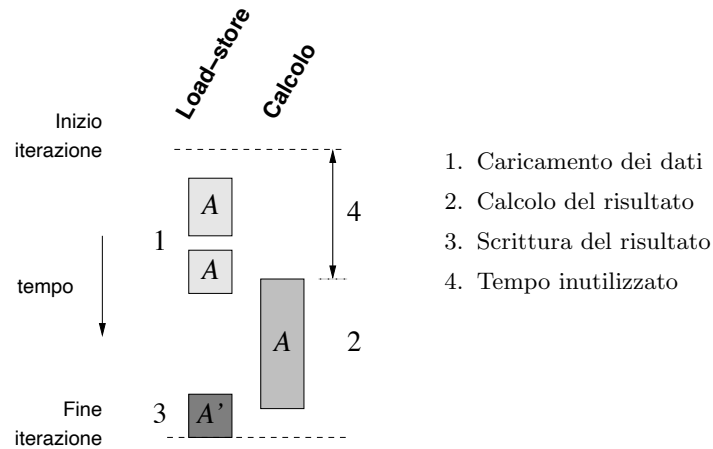


Figura 5.7: *Schema delle dipendenze senza windowing.* Nella versione più semplice del programma, l'unità aritmetica è inutilizzata nella parte iniziale di ogni ciclo (4). Il motivo è che le matrici U e ϕ di cui dovrà eseguire il prodotto non sono ancora state caricate dalla memoria (1). Il calcolo può iniziare solo quando siano passate entrambe le latenze ed almeno uno dei vettori sia stato letto per intero.

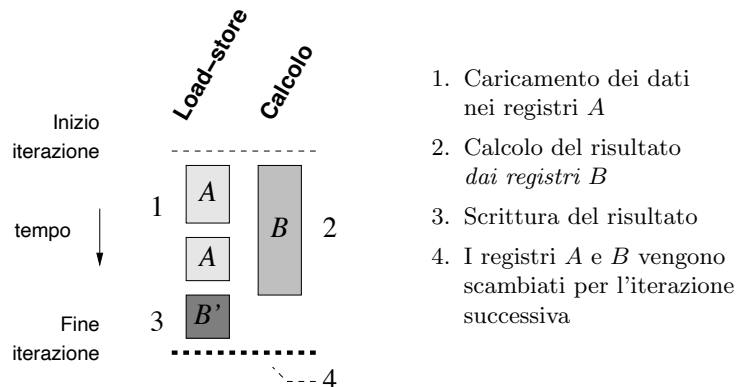


Figura 5.8: *Schema delle dipendenze con windowing*. Il register windowing permette di eliminare il tempo morto visibile in figura 5.7 perché l'unità aritmetica lavora su dati già caricati al ciclo precedente. Il caricamento dei valori che saranno usati al sito successivo avviene su *un altro* set di registri, che prende il posto del primo alla fine del ciclo. I set di registri sono utilizzabili uno alla volta, da cui il nome.

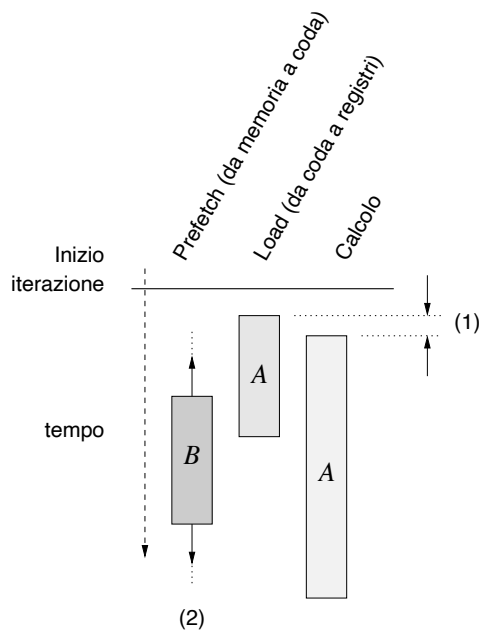


Figura 5.9: *Schema delle dipendenze con prefetch*. Il prefetch, allo stesso modo del windowing, permette di iniziare quasi subito ad elaborare i dati caricati (1). L'accesso in memoria per il prefetch del prossimo sito non deve rispettare particolari vincoli e i calcoli non dipendono da esso (2).

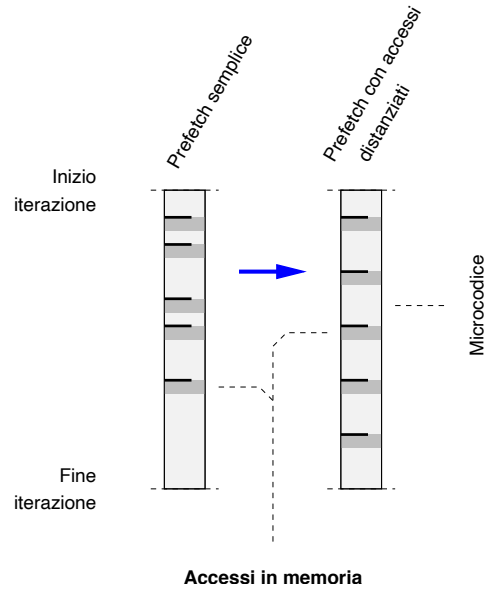


Figura 5.10: *Riduzione dei memory stretch*. Si è provato a ridurre la possibilità di memory stretch allontanando tra di loro gli accessi alla memoria: i preload sono stati distribuiti uniformemente nel microcodice.

grandi il peso dei fifo stretch diminuisca sensibilmente, mentre non vale lo stesso per i memory stretch. Si è provato ridurre l'impatto di questi ultimi distribuendo gli accessi in memoria nel microcodice di modo che fossero ugualmente distanziati (figura 5.10). Il programma è stato fatto girare e si sono ottenuti i risultati riportati in tabella 5.8.

L'efficienza m della memoria è effettivamente aumentata, ma allo stesso tempo l'efficienza f della rete è diminuita. Il motivo è che in media i prefetch si sono avvicinati alla fine della iterazione e quindi all'inizio della successiva. L'efficienza netta in questo caso è rimasta invariata, ma la modifica ha un certo valore perché, come si è detto, i fifo stretch in reticoli di dimensioni maggiori saranno notevolmente ridotti.

operazioni (per sito)	registri usati	dimensione microcodice			
N		c_i	c_{eff}	l_i	l_o
342	186	0.85	0.82	400	459

operazioni (totale)	totale cicli	stretch	
N_{tot}	C	fifo F	memoria M
10944	16687	1765	1395

	η_i	η_{eff}	f	m	η
Prefetch completo (accessi distanziati)	0.70	0.67	0.89	0.92	0.66

Tabella 5.8: L'impatto dei memory stretch è stato ridotto distanziando gli accessi in memoria.

5.10 Considerazioni finali

5.10.1 Il rapporto con i codici di produzione

Lo scopo della scrittura di questo codice era determinare la potenza di calcolo effettiva che sarà ragionevolmente disponibile su apeNEXT. A questo scopo il codice è stato sottoposto ad uno studio piuttosto intenso ed ottimizzazioni “a mano” che normalmente non vengono adottate nello sviluppo di codici di produzione.

I codici vengono comunemente scritti in un linguaggio di programmazione di alto livello chiamato TAO, adottato inizialmente nel primo modello di APE. Questo linguaggio ha una sintassi semplice che può essere estesa all'interno dello stesso programma (grammatica dinamica); tra le estensioni già disponibili ci sono ad esempio le operazioni tra matrici $SU(3)$ e spinori, che vengono espresse con una sintassi naturale ed eventualmente modificabile dall'utente.

Non è realistico aspettare che un codice di funzionalità analoga a quello realizzato, ma scritto in TAO, raggiunga lo stesso livello di prestazioni misurate in questo lavoro. Tuttavia, l'esperienza raccolta nei progetti APE100 ed APEmille mostra che è ragionevole attendersi livelli di performance intorno ai valori comunque rispettabili del 40 – 50%.

5.10.2 La gestione della fifo nei linguaggi di alto livello

In base a quanto affermato, è importante che chi programmerà in TAO o in altri linguaggi di alto livello abbia a disposizione una sintassi per imporre esplicitamente il prefetch sulle quantità desiderate. Una caratteristica simile non ha analoghi nei linguaggi di programmazione esistenti.

La figura 5.11 mostra due esempi di come gli utenti potranno comandare esplicitamente l'uso della coda ed il prefetch. Gli esempi espongono rispettivamente il formato previsto per il linguaggio TAO e quello del C.⁴

5.10.3 Ulteriori modifiche

L'analisi che è stata descritta ha messo in evidenza un certo numero di valori interessanti e di direzioni per sviluppi futuri. Innanzitutto, dal confronto di η con η_i si nota che riscrivendo le intestazioni dei loop in modo diverso, si potrebbe guadagnare fino ad un ulteriore 3% di efficienza in ogni versione.

L'uso di reticoli locali di dimensioni maggiori porterà un incremento di prestazioni che ragionevolmente sarà dell'ordine del 5%. In ogni caso converrà utilizzare la tecnica descritta per distanziare nel tempo gli accessi alla memoria.

⁴ La sintassi indicata è supportata dal compilatore C per apeNEXT scritto dall'autore.

L'impatto dei fifo stretch, già trascurabile, può essere ulteriormente ridotto usando una ulteriore tecnica, che consente di ridurre il numero di dati remoti da trasferire. La tecnica consiste nell'eseguire il prodotto $U \cdot B \cdot \phi$ nel sito remoto e trasferire il risultato anziché i suoi fattori. Non è prevedibile però un notevole miglioramento nelle prestazioni: da una parte il fifo stretch è già praticamente nullo su reticoli di dimensioni locali $L > 3$; una riduzione del numero di dati da trasferire non porterà probabilmente un beneficio evidente. D'altro canto questa tecnica non altera il numero di operazioni aritmetiche che è necessario calcolare, che sono il fattore che attualmente incide di più sulla lunghezza del microcodice.

Ancora, si potrebbe raddoppiare il ciclo in modo da spazzare il reticolo esaminando i siti a coppie $(x, y, z, 2k)$ e $(x, y, z, 2k + 1)$, con $k \in \mathbf{N}$. I siti di indice pari vengono calcolati come al solito; quelli di indice dispari accedono invece ad un numero minore di dati perché lo spinore centrale è già stato caricato in quanto adiacente nella direzione $+\hat{t}$. Analogamente, lo spinore nella direzione $-\hat{t}$ del sito dispari è disponibile perché era lo spinore centrale del sito pari. Lo stesso vale per le matrici di gauge. Non è possibile stimare l'effetto di questa strategia sulle prestazioni perché da una parte probabilmente le intestazioni dei loop diventerebbero più complesse; dall'altra lo scheduling avrebbe più libertà nel lavorare su un codice più lungo. Il codice attuale impiega 186 registri; questo valore può essere ridotto riutilizzandone alcuni in maniera opportuna. Il raddoppio del codice con la tecnica descritta aumenterebbe il numero di registri usati a 186×2 , maggiore dei 256 disponibili; occorrerebbe quindi introdurre una strategia di riutilizzo o renaming dei registri.

<pre> matrix real block.[10] queue block qvar real src[10],dest[10] integer i do i = 0,9 src[i] = i enddo extract qvar from src[0+right] do i = 0,9 src[i] = i + 20 enddo replace qvar into dest[0]</pre>	<pre> double src[10]; double dest[10]; int main() { int i; for (i = 0; i < 10; i++) { src[i] = i; } QSEND(src + right, 10); for (i = 0; i < 10; i++) { src[i] = i + 20; } QRCV(dest, 10); }</pre>
---	---

Figura 5.11: Esempi di gestione della coda nei linguaggi di alto livello TAO e C. In entrambi i casi la modifica dell'array `src` avviene durante il transito dei `i` dati.

Conclusioni

Lo scopo del presente lavoro di tesi era misurare l'efficienza del supercalcolatore dedicato apeNEXT, attualmente in progetto presso l'INFN di Pisa, su un kernel di calcolo particolarmente significativo per le simulazioni di QCD su reticolo: l'operatore di Dirac.

È stato scritto un codice che realizza questo calcolo ed il codice è stato eseguito su un modello dell'hardware che ne riproduce il comportamento con l'accuratezza di un ciclo di clock. Dalle simulazioni si è ottenuto che il calcolo dell'operatore di Dirac su un reticolo di dimensioni pari a $2^3 \times 4$ siti avviene con una efficienza netta pari a

$$\eta = 0.66$$

Questa efficienza include gli effetti di ogni dettaglio dell'hardware della macchina, incluse la rete di trasmissione e la memoria dinamica. La misura che è stata ottenuta corrisponde ad una potenza di calcolo *sostenuta* pari a 1.06 Gflops per nodo, ovvero la macchina impiegherà $2.607 \mu s$ per calcolare l'operatore di Dirac (nella forma studiata) su un sito.

Le efficienze fornite hanno ulteriori margini di miglioramento, che sono stati anche essi quantificati nel quinto capitolo.

Il valore dell'efficienza indicato è confrontabile con quello riportato per le macchine analoghe CP-PACS (63%) e QCDSF ($\sim 30\%$), mentre questo valore per la QCDOC non è ancora noto.

Il lavoro ha aiutato lo sviluppo hardware del nodo di calcolo, sia per le valutazioni quantitative che ne sono emerse, sia perché ha permesso di individuare aree critiche e relative soluzioni. In particolare è emerso il ruolo fondamentale della tecnica del prefetch, che è stata studiata a fondo e che sarà resa disponibile agli utenti.

Le valutazioni che sono emerse dal lavoro saranno utili quando la macchina sarà messa al servizio della più larga frazione dei fisici che compongono la collaborazione.

Appendice A

A.1 Il modello MIMD

Un calcolatore parallelo è classificato come MIMD quando ciascuno dei processori di cui è composto può eseguire una sequenza di istruzioni potenzialmente diversa da quella che gli altri stanno eseguendo nello stesso istante.

È evidente che il modello MIMD è più flessibile di quello SIMD (che contiene come caso particolare); esso però presenta due difficoltà fondamentali. Innanzitutto è difficile garantire la cooperazione di un grosso numero di unità complesse sotto tutte le condizioni possibili. Ogni tipo di eccezione e di disallineamento deve essere gestito in maniera efficiente e pulita, perché tale è il regime in cui la macchina opera.

La seconda difficoltà è di carattere software; da un lato chi scrive i codici di calcolo non si può imbarcare nel compito estremamente complesso e pronò ad errori di preparare un diverso programma per ciascun processore; dall'altra parte, nonostante l'ampia ricerca e gli enormi interessi, non sono noti metodi sufficientemente generali per la parallelizzazione automatica del codice.

Ciononostante il modello MIMD è risultato quello di maggiore successo nei supercalcolatori commerciali moderni. Tra i calcolatori SIMD più noti si ricorda la Connection Machine, ora fuori produzione; la schiera dei MIMD conta rappresentanti più numerosi, quali le ultime generazioni di Cray e i multiprocessori con memoria condivisa prodotti dalla Silicon Graphics. Anche le architetture a multiprocessore simmetrico (SMP), note in tempi recenti perché l'economia di scala ne ha reso il costo così basso da essere accessibili anche a privati, possono essere considerate architetture MIMD (per quanto nel caso più comune i problemi non vengano automaticamente parallelizzati).

A.2 Misurare in flops

L'unità di misura più usata per esprimere le prestazioni di calcolo numerico di una macchina è il flops. Il nome deriva dalla contrazione dei termini "floating point

operations per second”. Nella pratica si incontrano valori di flops dell’ordine di $10^6 - 10^9$; per questo le unità derivate di uso più comune sono il Mflops e Gflops.

La definizione dell’unità di misura è apparentemente chiara dal nome: *numero di operazioni numeriche in virgola mobile che possono essere calcolate nell’unità di tempo*. La definizione è naturale, ma contiene un certo numero di ambiguità che è opportuno elencare qui [29].

- non è dichiarata la precisione con cui vengono effettuate le operazioni
- non è specificato il tipo di operazione aritmetica in questione
- non è specificato il tempo che occorre perché sia disponibile il risultato corrispondente ad un dato input
- non sono indicate le dipendenze tra le unità funzionali

Per questo motivo, l’indicazione del numero di flops di una macchina è una stima soltanto approssimativa della complessità dei problemi numerici che l’utente potrà sperare di affrontare.

In pratica si distinguono due tipi di misura. I flops *di picco* ovvero *sostenuti* sono il numero massimo di operazioni in virgola mobile che una macchina può calcolare nelle migliori condizioni: nessuna dipendenza tra dati, e con la precisione che permette il calcolo più veloce. I flops *effettivi* sono dati dal numero di volte che la macchina calcola un algoritmo numerico di interesse per secondo, moltiplicato per il numero di operazioni in virgola mobile indicate nell’algoritmo. La seconda misura, assai più significativa, è fortemente dipendente dal particolare algoritmo numerico, ed anche dalla particolare implementazione che se ne considera.

A.3 Rappresentazioni approssimate

Chi elabora un programma si aspetta che un calcolatore esegua una sequenza assegnata operazioni aritmetiche tra grandezze reali. Il calcolatore (meccanico o elettronico), però, è una macchina finita in cui le operazioni sono scomposte ed eseguite in una serie di passaggi discreti che impiegano un tempo finito. Per questo motivo occorre che le grandezze reali sulle quali opera siano (almeno internamente) rappresentate in maniera necessariamente approssimata. Si dice (in questo contesto) *rappresentazione approssimata* $R = R(a)$ una funzione che associa ad ogni numero reale a appartenente ad un opportuno sottoinsieme della retta reale, un elemento R appartenente ad un insieme finito.

Nel caso di un calcolatore digitale, la scelta della rappresentazione è in parte arbitraria, ed è affidata al progettista; essa influenzerà alcuni parametri come la

semplicità di costruzione e l'efficienza. Una volta scelta una rappresentazione, si derivano infatti gli algoritmi approssimati per rappresentare le operazioni aritmetiche nella stessa. Si fissa ad esempio una P tale che

$$R(a \cdot b) = P(R(a), R(b))$$

Si dice così che P realizza il prodotto di due grandezze reali. P è una funzione di due argomenti appartenenti ad uno spazio finito, e di conseguenza può essere realizzata da un algoritmo. (Un algoritmo è una sequenza *finita* di operazioni specificate indipendentemente dai dati di ingresso.)

Nella pratica esistono alcune rappresentazioni standard alle quali è conveniente aderire, perché portano dei vantaggi che sono stati studiati; per esempio, la rappresentazione delle operazioni aritmetiche in questi standard è realizzata con algoritmi noti.

Una rappresentazione efficace, naturale e standardizzata è quella esponenziale, di cui diamo qui di seguito un breve cenno senza pretesa di rigore.

Se a è un numero reale, si associano ad esso due interi positivi o negativi, chiamati rispettivamente mantissa m ed esponente s , in modo tale che

$$0.m \cdot 10^s - a = \text{minimo}$$

Si fissa quindi il numero di cifre decimali di cui m ed s possono essere composti; chiamiamo questi due interi M ed S . Con questo, la quantità di informazione contenuta in m ed s sarà finita, e di conseguenza diremo che la rappresentazione esponenziale di a , $R_e(a)$, è il vettore di $M + S$ cifre decimali

$$R_e(a) = [m_1 m_2 \dots m_M s_1 s_2 \dots s_S]$$

avendo indicato con l'indice i in basso la i -esima cifra decimale. Quando la coppia (m, s) non è unica, esistono delle prescrizioni per la sua scelta; in caso contrario la rappresentazione sarà detta “denormalizzata”.

La rappresentazione più usata nei calcolatori digitali è quella standardizzata¹ dall'IEEE (Institute for Electrical and Electronic Engineers, inc.). La notazione esponenziale IEEE differisce da quella descritta sopra in quanto non è usata la base decimale ma quella binaria. Lo standard non specifica i valori per M ed S , che di conseguenza possono essere scelti in base alla precisione richiesta nel calcolo. Il valore di $M + S$ viene comunemente chiamato “lunghezza di parola”, e misurato comprensibilmente in bit. Motivi pratici fissano la lunghezza di parola quasi sempre a 32 o 64 bit, in due casi che prendono rispettivamente i nomi convenzionali di “singola precisione” e “doppia precisione”.

¹norme ANSI/IEEE std 754-1985 ed IEC 559

A.4 Le inizializzazioni nel codice dell'operatore di Dirac

Il capitolo 5 presenta una analisi dettagliata del core di calcolo di una versione realistica dell'operatore di Dirac per apeNEXT. Il core da solo non può essere eseguito né nel simulatore né nella macchina reale, e deve essere supportato da un certo numero di inizializzazioni che vengono qui elencate nell'ordine.

- Si caricano in registri le costanti intere e floating point usate più spesso (p. es. 0, 1, i).
- I campi U e ϕ di un sito sono letti e ricopiati in tutti gli altri siti (e direzioni, nel caso di U).
- Si preparano sei vettori con L^3 elementi ciascuno, dove L è la dimensione spaziale del reticolo. Questi vettori servono per ricavare l'indice di sito $p + \hat{\mu}$, ad assegnati p e μ (figura A.1). Analogamente si procede per i puntatori a $t + 1$ e $t - 1$.
- Nel caso di trasferimenti con prefetch, le code vengono caricate con i dati del primo sito prima di entrare nel core
- Vengono eseguite le inizializzazioni necessarie a portare il core nella cache istruzioni.
- Alla uscita del core, vengono svuotate le fifo del sito (inesistente) precaricato durante l'ultima iterazione.

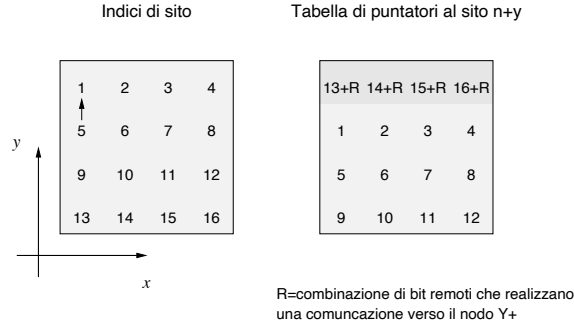


Figura A.1: L'inizializzazione dei puntatori ai siti vicini.

Bibliografia

- [1] H. J. Rothe. *Lattice Gauge Theories: An introduction*. World Scientific, 1997.
- [2] K. Wilson. Confinement of Quarks. *Phys. Rev. D*, **10** (1974), 2445.
- [3] S. Güsken et al. Lattice QCD with two dynamical Wilson fermions on APE100 parallel systems. *Parallel Computing: high performance computing in lattice QCD*, **25** (1999), 1227–1242.
- [4] Particle Data Group. Particle Physics Booklet.
- [5] F. Jegerlehner et al. Requirements For High Performance Computing For Lattice QCD: Report of the ECFA Working Panel. ECFA/99/200.
- [6] M. Creutz. *Quarks, Gluons and Lattices*. Cambridge University Press, 1985.
- [7] S. R. Sharpe. Progress in Lattice Gauge Theory. Preprint on hep-lat/9811006.
- [8] D. Bini e M. Capovani. La scienza del calcolo fra spazio e tempo. *Le Scienze Quaderni: Matematica Computazionale*, **84** (Giugno 1995), 68–74.
- [9] P. Rossi, C. T. H. Davies e G. P. Lepage. A comparison of a variety of matrix inversion algorithms for Wilson fermions on the lattice. *Nuclear Physics B*, **297** (1988), 287–314.
- [10] R. Gupta. General physics motivations for numerical simulations of quantum field theory. *Parallel Computing: high performance computing in lattice QCD*, **25** (1999), 1199–1215.
- [11] N. H. Christ. QCD Machines, Present and Future. In *Computing for High Luminosity and High Intensity Facilities*, volume 209 di *AIP Conference Proceedings*, pagine 541–548. American Institute of Physics, New York, 1990.
- [12] R. Tripiccone. Private communication.
- [13] M. Creutz. Microcanonical Monte Carlo Simulation. *Phys. Rev. Letters*, **50** (1983), 1411–1414.

- [14] V. Kumar, A. Grama, A. Gupta e G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings Publishing Company, Redwood City, California, 1994.
- [15] Th. Lippert, K. Schilling, F. Toschi, S. Trentmann e R. Tripiccone. Transpose Algorithm for FFT on APE/Quadrics. In *proceedings of HPCN98*. Springer, Amsterdam, 1998.
- [16] Th. Lippert, K. Schilling, F. Toschi, S. Trentmann e R. Tripiccone. FFT for the APE parallel computer. *International Journal of Modern Physics C*, **8** (1997), 1317.
- [17] N. H. Christ. Computers for Lattice QCD. *Nuclear Physics B (Proc. Suppl.)*, **83–84** (2000), 111–115. Presentato alla conferenza Lattice '99.
- [18] Y. Iwasaki. Computers for Lattice Field Theories. Preprint on hep-lat/9401030.
- [19] Y. Iwasaki. The CP-PACS Project and Lattice QCD Results. *Progress of Theoretical Physics Supplement*, **138** (2000), 1–10. Preprint hep-lat/0002024.
- [20] A. Kawai et al. GRAPE-5: A Special-Purpose Computer for N-body Simulation. Preprint astro-ph/9909116.
- [21] A. Ukawa (for the CP-PACS collaboration). Lattice QCD results from the CP-PACS computer. *Parallel Computing: high performance computing in lattice QCD*, **25** (1999), 1257–1280.
- [22] R. D. Mawhinney. The 1 Teraflops QCDSP computer. *Parallel Computing: high performance computing in lattice QCD*, **25** (1999), 1281–1296.
- [23] D. Chen et al. QCDOC: A 10-teraflops scale computer for lattice QCD. Preprint on hep-lat/0011004.
- [24] S. Aoki et al. Performance of lattice QCD programs on CP-PACS. *Parallel Computing: high performance computing in lattice QCD*, **25** (1999), 1243–1255.
- [25] R. Alfieri et al. ApeNEXT: A Multi-TFlops LQCD computing project. Preprint hep-lat/0102011.
- [26] S. Aoki et al. *Int. J. Mod. Phys. C*, **2** (1991), 829.
- [27] H. Simma. Private communication.
- [28] Th. Lippert, A. Seyfried, A. Bode e K. Schilling. Hyper-Systolic Parallel Computing. Preprint hep-lat/9507021, WUB 95-13, HLRZ 32/95.
- [29] D. A. Patterson e J. Hennessy. *Computer Architectures: A Quantitative Approach*. Kaufmann, San Francisco, seconda edizione, 1996.