

Lezioni di Calcolo Parallelo e Distribuito

Prof. Almerico Murli ¹

Università degli Studi di Napoli
“Federico II”

11 giugno 2004

¹Già direttore del Centro di Ricerca per il Calcolo Parallelo e i Supercalcolatori del Consiglio Nazionale delle Ricerche (CPS-CNR), ora direttore della Sezione di Napoli dell'Istituto di Calcolo e Reti ad Alte Prestazioni del CNR (ICAR).

Indice

Introduzione	1
1 Calcolo parallelo: l'idea di base	1
1.1 CASE STUDY: l'operazione di somma	15
2 Le architetture parallele	30
2.1 Classificazione delle architetture parallele	30
2.2 Reti di interconnessione per sistemi a memoria distri- buita	41
2.3 Il principio della località dei dati	46
3 Il paradigma dello scambio di messaggi (message pas- sing)	50
3.1 CASE STUDY: l'operazione matrice per vettore . . .	51
3.2 CASE STUDY: somma di p vettori	84
3.3 Lo standard MPI	92
3.4 CASE STUDY: l'operazione del prodotto di due matrici	100
4 I parametri di valutazione del software parallelo	128
4.1 La definizione "classica" di speed-up ed efficienza . . .	129
4.2 Il modello scalato di speed-up ed efficienza	141

"... As soon as Analytical Engine exists, it will necessarily guide the future course of science.

*Whenever any result is sought by its aid,
the question will then arise -
by what course of calculation can these results
be arrived at by machine in the shortest time? ..."*

Babbage, 1864

*"...Non appena la Macchina Analitica esisterà,
essa necessariamente guiderà il futuro della scienza.
Tutte le volte che un risultato sarà ottenuto con il suo aiuto,
una domanda sorgerà spontanea -
Con questa macchina attraverso quale tipo di calcoli
si può ottenere lo stesso risultato
nel minimo tempo possibile ? ..."*

Introduzione

Uno dei primi riferimenti in letteratura all'utilizzo di uno strumento di calcolo in grado di effettuare numerose operazioni "contemporaneamente" è probabilmente quello comparso nella pubblicazione della Biblioteca Universale di Ginevra dell'ottobre 1842 intitolata "Sketch of the Analytical Engine Invented by Charles Babbage" di J.F. Menabrea. Nell'elencare le capacità della macchina analitica, egli scrive [31]:

"[...] In secondo luogo, risparmiare tempo: per convincerci di questo basti pensare che la moltiplicazione di due numeri, costituiti entrambi da venti cifre, richiede almeno tre minuti di calcolo. Parimenti, quando deve essere eseguita una lunga serie di calcoli identici, come quelli richiesti per realizzare una tabella numerica, la macchina può essere impiegata in modo tale da fornire più risultati allo stesso tempo, cosa che ridurrebbe significativamente il numero di processi. [...]"

Non è chiaro se il "parallelismo"¹ fosse effettivamente implementato dalla macchina analitica, ma, in ogni caso, fu evidente già a quei tempi quanto forte poteva essere l'impatto dello strumento di calco-

¹Per ora possiamo solo intuire il significato di "parallelismo" basandoci su considerazioni empiriche e sull'esperienza che ciascuno ha. Nel seguito verrà data una definizione più adeguata e precisa.

lo sulla metodologia di risoluzione di un problema scientifico.

Dai tempi della “*analytical engine*” di Babbage ad oggi è sempre stata forte l’influenza tra l’avanzamento della tecnologia e la necessità e/o possibilità di risolvere problemi nuovi e più complessi derivanti dal mondo “*fisico*” [42]. All’aumentare della capacità di calcolo e dell’efficienza degli algoritmi, aumenta la possibilità di risolvere problemi di maggiore complessità².

Attualmente le infrastrutture per il calcolo e la comunicazione sono uno strumento indispensabile per l’avanzamento della conoscenza in tutte le scienze, dalla fisica alla biologia, all’economia. La disponibilità di calcolatori e reti veloci, tuttavia, non è necessariamente garanzia di risoluzione di problemi complessi poiché l’utilizzo efficace di tali strumenti di calcolo richiede la conoscenza dell’ambiente di calcolo, portando a volte a modifiche sostanziali delle metodologie, degli algoritmi e del relativo software.

La risoluzione efficiente dei problemi del mondo fisico richiede, però, una forte sinergia tra discipline differenti. In questa ottica, la comunità scientifica ha, da tempo, iniziato a parlare di una terza metodologia per l’indagine scientifica che si basa sull’approccio computazionale, affiancando gli approcci classici, cioè quello teorico e quello sperimentale (***Computational Science and Engineering*** (CSE) fig. 1).

La simulazione computazionale consente la risoluzione di problemi prima ritenuti inaccessibili o comunque di risoluzione difficile. Essa, infatti, consente una analisi qualitativa e quantitativa di fenomeni troppo complessi da studiare analiticamente (gallerie del vento, evoluzione del clima, evoluzione delle galassie,...) o di problemi

² “The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less.” L. Trefethen - “MAXIMS ABOUT NUMERICAL MATHEMATICS, SCIENCE, COMPUTERS, AND LIFE ON EARTH”

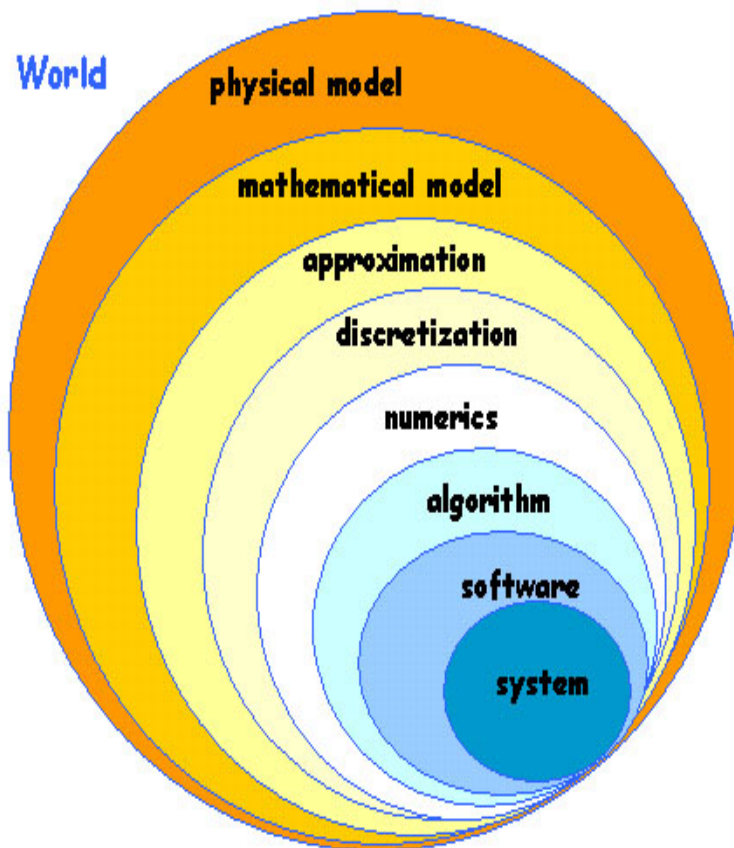


Figura 1: Il modello computazionale del Problem Solving della CSE può essere rappresentato come un insieme di cerchi concentrici. Il mondo reale è situato all'esterno dei cerchi. I livelli successivi rappresentano le diverse fasi del processo di risoluzione computazionale del problema: la costruzione del modello fisico, la formulazione del modello matematico, l'approssimazione e discretizzazione di quest'ultimo, lo sviluppo di algoritmi, l'implementazione di questi ultimi in uno specifico ambiente di calcolo (software) e la realizzazione del sistema che costituisce lo strumento finale a disposizione dell'utente.

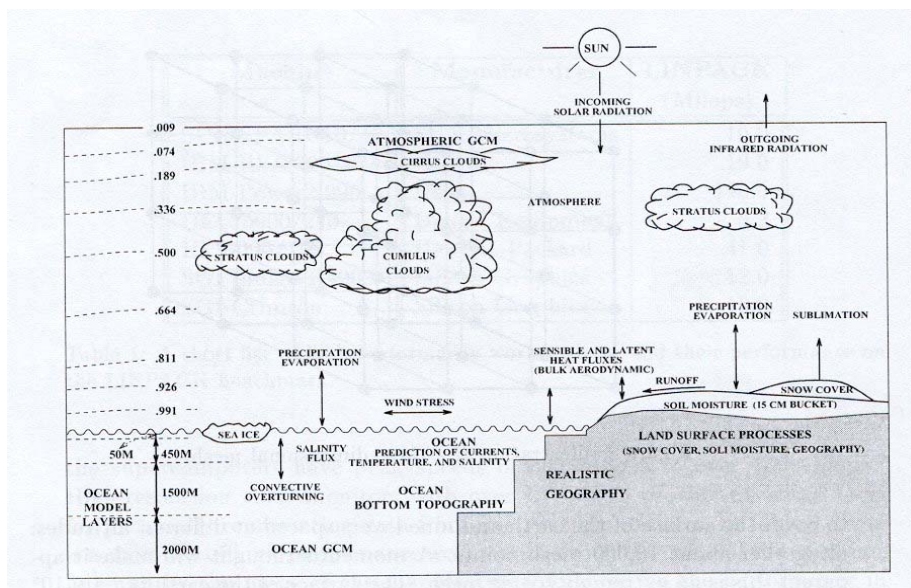


Figura 2: Schema del processo fisico alla base del modello GCM

troppo pericolosi per essere replicati in laboratorio (sicurezza delle armi nucleari, efficacia dei farmaci,...).

♣ Esempio 1 [21]

Il surriscaldamento del globo è oggetto di una attenzione sempre maggiore da parte della comunità scientifica. Lo studio di questo problema serve a comprendere come la variazione di concentrazione di diossido di carbonio nell'atmosfera contribuisca al surriscaldamento del pianeta per effetto serra. Uno studio di questo tipo richiede la modellizzazione del clima in un certo periodo di tempo. Sono stati effettuati studi al *National Center for Atmospheric Research*. Un modello climatico noto come General Circulation Model (GCM) viene usato per studiare il surriscaldamento che si genererebbe raddoppiando la concentrazione del diossido di carbonio in un periodo di 20 anni. Gli effetti che il modello GCM tenta di descrivere sono mostrati in Fig. 2.

L'atmosfera è un fluido, quindi il modello è descritto da equazioni differenziali alle derivate parziali. Le soluzioni numeriche di queste equazioni sono ottenute mediante algoritmi in cui le derivate rispetto alle coordinate spaziali e al tempo vengono approssimate da formule alle differenze nello spazio e nel tempo. Viene quindi generato un reticolo (griglia) a tre dimensioni nello spazio.

Per determinare la soluzione del problema vengono fissate alcune condizioni iniziali che assegnano alle variabili dei valori nei punti della griglia.

Se si usa un reticolo tridimensionale con circa 2000 punti, che coprono la superficie terrestre e nove piani posti a diverse altitudini, in totale vengono utilizzati circa 18000 punti per approssimare la soluzione. Questa discretizzazione risulta però abbastanza larga. La superficie terrestre è infatti di circa $5.103 \times 10^5 \text{ km}^2$, quindi ogni punto del reticolo rappresenta 28.350 km^2 della superficie. Ad esempio, tenuto conto che l'Italia ha una superficie di circa 301.401 km^2 , solo 10 punti del reticolo rappresentano il nostro Paese. È ovvio quindi che occorre una maggiore accuratezza: se raddoppiamo il numero dei punti in ognuna delle tre direzioni aumentiamo il numero totale di un fattore $2^3 = 8$ anche se abbiamo solo 80 punti che rappresentano la superficie dell'Italia.

△

In tale contesto, il risultato del processo di risoluzione di un problema è un software funzionante in uno specifico ambiente di calcolo. L'intero processo, noto come *Computational Problem Solving Methodology* consiste nella costruzione del modello fisico, del modello matematico, nell'approssimazione e discretizzazione di tale modello, nell'applicazione di un metodo numerico, nel progetto di un algoritmo ed infine, nell'implementazione di un software per uno specifico sistema di calcolo.

Ed è proprio in questo contesto che si inserisce il *Calcolo Parallelo*, caratterizzato da quell'insieme di principi e metodologie, che intervengono in tutte le aree del calcolo scientifico in un ambiente ad alte prestazioni e che rendono possibile la risoluzione computazionale di un problema; pertanto, nella parte più strettamente numerica, il Calcolo Parallelo può essere considerato come una naturale evoluzione dell'Analisi Numerica.

♣ Esempio 2 [5]

Si vuole stimare il tempo di risposta necessario per effettuare una ricerca sulla rete Internet tramite "motore di ricerca".

È evidente che il tempo di risposta dipende soprattutto dalla velocità del collegamento che si ha a disposizione e dal motore di ricerca. Per quanto riguarda il collega-

mento, se questo viene effettuato ad esempio da casa, il tempo necessario ad inviare la richiesta è uguale ad 1 secondo (modem a 56 Kbs). Se il collegamento viene effettuato da un centro universitario come ad esempio l'Università degli Studi di Napoli Federico II, al momento in cui si scrive il tempo per inviare la richiesta è circa $\frac{1}{10^6}$ di secondo (1G-GE)³. Una volta inviata la richiesta nasce il problema di fornire una risposta nel più breve tempo possibile.

La ricerca di informazioni tramite un motore di ricerca è realizzata mediante parole chiave: l'utente fornisce al motore di ricerca le parole più significative e questo restituisce un elenco di tutte le pagine web contenenti tali parole. La ricerca di documenti da parte dell'utente è basata sul significato concettuale di ciò di cui ha bisogno. Il problema è che singole parole non evidenziano il contenuto concettuale di un documento. Ci sono infatti molti modi per esprimere un concetto (*synonymy*), quindi i termini letterali forniti in una richiesta (query) possono non essere quelli presenti nel documento. Inoltre, molte parole possono avere significati diversi in contesti diversi o se usati da persone diverse (*polysemy*).

È stato sviluppato un metodo per l'indicizzazione e la ricerca di documenti. Esso è noto come LSI (Latent Semantic Indexing) [2, 5] e si basa sull'utilizzo della Decomposizione in valori singolari (Singular Value Decomposition)⁴ [27]. Gli elementi della matrice sono l'occorrenza di ogni parola in un documento

$$A = [a_{ij}]$$

con a_{ij} frequenza con cui il termine i occorre nel documento j . Poiché ogni parola normalmente non appare in tutti i documenti, la matrice A generalmente è sparsa. Ogni coefficiente a_{ij} della matrice A è ottenuto moltiplicando il peso $L(i, j)$ locale del termine i , nel documento j , e il peso $G(i)$ globale dei termini i . Ovvero ogni elemento

³ Il simbolo G indicare 1 Ggigabit al secondo, mentre GE indica Ggigabit Ethernet. A regime si avrà una banda di 2,5 gigabit.

Nel sito www.garr.net è possibile accedere a grafici che mostrano in tempo reale la velocità di collegamento alla rete dall'Università e dai principali nodi italiani.

⁴ Diamo un breve richiamo della SVD. Data una matrice A di dimensione $m \times n$ con $m \geq n$ e $\rho(A) = r$ (ρ denota il rango della matrice), la SVD di A , denotata da $SVD(A)$, è definita da:

$$A = USV^T$$

dove $U^T U = V^T V = I_n$ e $S = \text{diag}(\sigma_1, \dots, \sigma_n)$, $\sigma_i > 0$ per $1 \leq i \leq r$, $\sigma_j = 0$ per $j \geq r + 1$. Le prime r colonne delle matrici ortogonali U e V definiscono gli autovettori ortonormali associati agli r autovalori non nulli di AA^T e $A^T A$, rispettivamente. Le colonne di U e V rappresentano i vettori singolari sinistri e destri rispettivamente, e i valori singolari di A sono definiti dagli elementi diagonali di S , che sono le radici quadrate degli autovalori di AA^T . Si dimostra [27] che se della matrice S si considerano solo i k valori singolari più grandi e le rispettive colonne delle matrici U e V , la matrice risultante A_k è la matrice di rango k che meglio approssima nel senso dei minimi quadrati la matrice A .

L'idea è che questa matrice, contenendo solo le prime k componenti di A linearmente indipendenti, evidenzia le associazioni nella matrice ed elimina i documenti con cui non ci sono associazioni.

Label	Titles
B1	A Course on <u>Integral Equations</u>
B2	Attractors to Semigroups and Evolution <u>Equations</u>
B3	Automatic Differentiation of <u>Algorithms</u> : <u>Theory</u> <u>Implementation</u> and <u>Application</u>
B4	Geometrical Aspects of <u>Partial Differential Equations</u>
B5	Ideals, varieties, and <u>Algorithms</u> - An <u>Introduction</u> to Computational Algebraic Geometry and Commutative Algebra
B6	<u>Introduction</u> to Hamiltonian Dynamical <u>Systems</u> and the N-body <u>Problem</u>
B7	Knapsack <u>Problems</u> : <u>Algorithms</u> and Computer <u>Implementations</u>
B8	<u>Methods</u> of Solving Singular <u>System</u> of <u>Ordinary Differential Equations</u>
B9	<u>Nonlinear System</u>
B10	<u>Ordinary Differential Equations</u>
B11	<u>Oscillation Theory</u> for Neutral <u>Differential Equations</u> with <u>Delay</u>
B12	<u>Oscillation Theory</u> of <u>Delay Differential Equations</u>
B13	Pseudodifferential Operators and <u>Nonlinear Partial Differential Equations</u>
B14	Sine <u>Methods</u> for Quadrature and <u>Differential Equations</u>
B15	Stability of Stochastic <u>Differential Equations</u> with Respect- to Semi-Martingales
B16	The Boundary <u>Integral</u> Approach to Static and Dynamic Contact <u>Problems</u>
B17	The Double Mellin-Barnes Type <u>Integral</u> and Their <u>Applications</u> to Convolution <u>Theory</u>

Tabella 1: *Titoli di libri estratti dalla SIAM Review. Le parole sottolineate compaiono in più di un titolo.*

a_{ij} è del tipo:

$$a_{ij} = L(i, j) \times G(i)$$

Si consideri ad esempio un piccolo database di titoli di libri. Nella tabella 1 sono riportati 17 titoli di libri estratti dalla SIAM Review, Volume 54, Number 4. Le parole sottolineate sono parole chiave usate come referenti al titolo del libro. Il metodo di analisi utilizzato richiede che le parole chiave appaiano in più di un titolo di libro.

A partire dalla tabella viene costruita una matrice termini-documenti di dimensione 16×17 mostrata in Fig. 3.

Mediante la SVD, la matrice A viene fattorizzata nel prodotto di tre matrici. La SVD fornisce il modello della struttura semantica latente a partire dalle matrici U , V e S . Queste matrici rappresentano una separazione delle relazioni originali in vettori

Terms	Documents															
<i>algorithms</i>	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
<i>application</i>	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
<i>delay</i>	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
<i>differential</i>	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0
<i>equations</i>	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	0
<i>implementation</i>	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
<i>integral</i>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
<i>introduction</i>	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
<i>methods</i>	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
<i>nonlinear</i>	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
<i>ordinary</i>	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
<i>oscillation</i>	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
<i>partial</i>	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
<i>problem</i>	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0
<i>system</i>	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0
<i>theory</i>	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	1

Figura 3: Matrice termini-documenti corrispondente ai titoli di libri riportati nella tabella 1. Ogni elemento della matrice rappresenta la frequenza con cui il termine i occorre nel documento j .

linearmente indipendenti. L'uso di k componenti è equivalente ad approssimare la matrice A con A_k .

Calcoliamo la SVD troncata con $k = 2$ per ottenere l'approssimazione di rango 2, A_2 .

Utilizzando per la coordinata x la prima colonna di U_2 moltiplicata per il primo valore singolare σ_1 e per la coordinata y la seconda colonna di U_2 moltiplicata per il secondo valore singolare σ_2 , i termini possono essere rappresentati su di un piano cartesiano. Analogamente la prima colonna di V_2 moltiplicata per il primo valore singolare σ_1 fornisce la coordinata x e la seconda colonna di V_2 moltiplicata per il secondo valore singolare σ_2 fornisce la coordinata y dei documenti. In Fig. 4 sono rappresentati i termini e i documenti della matrice termini-documenti. Si osservi che i documenti e i termini che riguardano l'argomento “*differential equations*” sono posizionati tutti intorno all'asse x e i termini e i documenti riguardanti l'argomento “*algorithms and applications*” sono posizionati tutti intorno all'asse y . Questo suggerisce che l'insieme $\{B2, B4, B8, B9, B10, B13, B14, B15\}$ contiene titoli simili nel significato.

Per il recupero di informazioni, l'utente deve formulare una richiesta. Tale richiesta può essere rappresentata da un vettore nello spazio di dimensione k che deve essere confrontato con i documenti. Una richiesta è un insieme di parole e può essere rappresentata da:

$$\bar{q} = q^T U_k S_k^{-1} \quad (1)$$

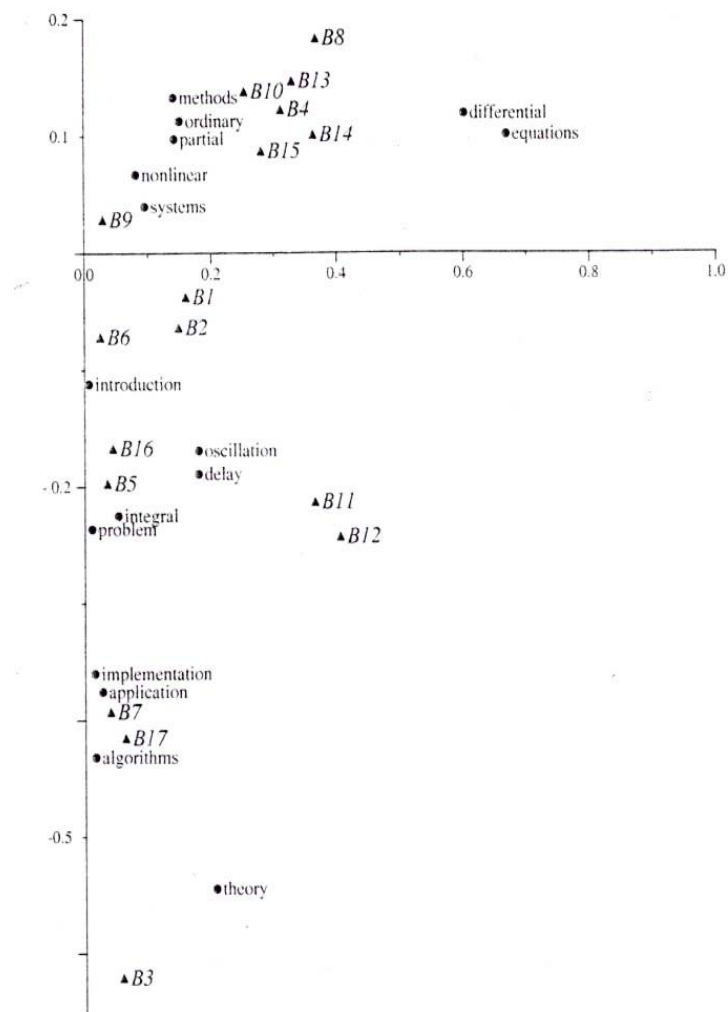


Figura 4: Rappresentazione nel piano cartesiano dei termini e dei documenti.

$$(0.0511 \ -0.3337) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} 0.0159 & -0.4317 \\ 0.0266 & -0.3756 \\ 0.1785 & -0.1692 \\ 0.6014 & 0.1187 \\ 0.6691 & 0.1209 \\ 0.0148 & -0.3603 \\ 0.0520 & -0.2248 \\ 0.0066 & -0.1120 \\ 0.1503 & 0.1127 \\ 0.0813 & 0.0672 \\ 0.1503 & 0.1127 \\ 0.1785 & -0.1692 \\ 0.1415 & 0.0974 \\ 0.0105 & -0.2363 \\ 0.0952 & 0.0399 \\ 0.2051 & -0.5448 \end{pmatrix} \begin{pmatrix} 4.5314 & 0 \\ 0 & 2.7582 \end{pmatrix}^{-1}$$

Figura 5: *Coordinate della richiesta "application theory". Il primo vettore a secondo membro rappresenta la nostra richiesta. Tenendo presente l'ordine in cui sono inserite le parole chiave nel database, nel vettore richiesta vi è 1 in corrispondenza di una parola richiesta e 0 in corrispondenza di una parola non richiesta. Il secondo termine a secondo membro è una matrice di dimensione 16×2 e rappresenta la matrice U_2 ottenuta mediante SVD troncato con $k = 2$. Il terzo termine a secondo membro è una matrice di dimensione 2×2 e rappresenta la matrice Σ ottenuta mediante SVD.*

dove \bar{q} è il vettore delle parole della richiesta dell'utente moltiplicato per gli appropriati pesi del termine: $q^T U_k$ rappresenta la somma di vettori di dimensione k e la moltiplicazione per S_k^{-1} pesa in modo diverso la dimensione.

Supponiamo di essere interessati ai documenti che riguardano l'argomento "*applications and theory*".

Dal punto di vista matematico, le coordinate cartesiane della richiesta (vedi fig. 5) sono determinate dall'equazione (1) e la richiesta è rappresentata dal punto marcato QUERY mostrato in Fig. 6.

Il vettore richiesta viene confrontato con i vettori dei documenti e i documenti vengono classificati in base alla loro similitudine alla richiesta. Una misura della similitudine è data dall'angolo delimitato dal vettore richiesta e dal vettore documento. I documenti il cui coseno supera un certo valore limite vengono restituiti all'utente.

La risposta alla nostra richiesta sarà ad esempio costituita dai documenti il cui vettore forma con il vettore richiesta un angolo il cui coseno è maggiore di 0.9. Tali documenti sono situati nella Fig. 6 tra l'asse negativo delle y e il vettore più grande. Analizziamo la complessità di tempo di questo algoritmo.

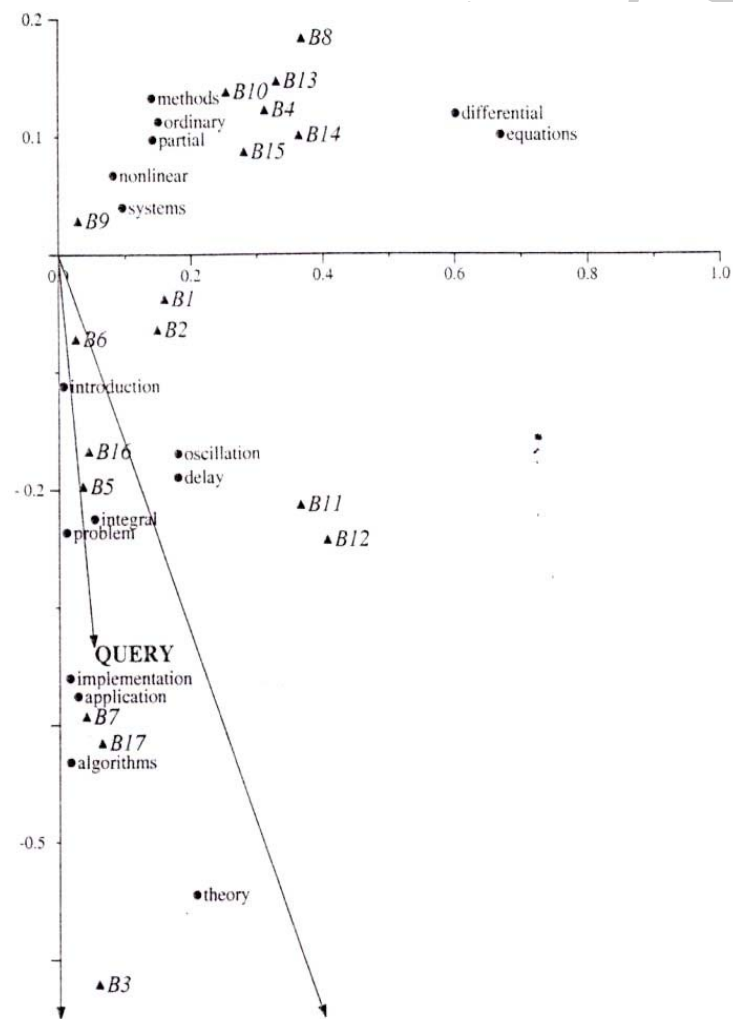


Figura 6: Rappresentazione nel piano cartesiano della richiesta denotata in figura con QUERY.

La complessità di tempo è data da $T(n) = kN^2$ con k numero di dimensioni considerate. Supponiamo che il motore di ricerca contenga 20 milioni di documenti (tale numero raddoppia ogni 5 mesi) e che risulti $k = 1000$. Questo vuol dire che $T(N) = kN^2 = 10^3 \cdot 2 \cdot 10^7 = 2 \cdot 10^{10}$, sono richiesti cioè 20 miliardi di operazioni per la risoluzione del problema.

△

L'esempio del motore di ricerca in Internet descrive una delle applicazioni di calcolo ad alte prestazioni più diffuse e più vicine alle nostre attività quotidiane, cioè la ricerca di una informazione sulla rete del web. Quanto si è disposti ad aspettare il risultato di una ricerca sulla rete internet?

Prendendo spunto da questa “*naturale*” esigenza vediamo quali caratteristiche dovrebbe avere un calcolatore perché sia in grado di elaborare una nostra richiesta sulla rete internet in 1 sec (secondo). Sia μ il periodo di clock, il tempo cioè che intercorre tra l'emissione di due impulsi nell'orologio interno del calcolatore. Il tempo di esecuzione della richiesta, τ , è proporzionale alla complessità di tempo dell'algoritmo utilizzato, T , e al periodo di clock, μ :

$$\tau \cong T \cdot \mu$$

Abbiamo visto che la complessità computazionale dell'algoritmo LSI, nel caso di un motore di ricerca che deve elaborare 20 milioni di documenti, assumendo il numero di dimensioni $k = 1000$, è:

$$T = 2 \cdot 10^{10}$$

Quindi, nel caso in esame si ha:

$$1 \text{ sec} = 20 \text{ Gflop} \cdot \mu$$

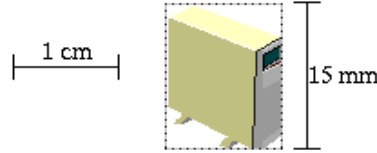


Figura 7: Affinché un motore di ricerca possa rispondere ad una richiesta in circa un secondo le dimensioni del computer devono essere di circa 15mm.

cioè

$$\mu = \frac{1}{2 \cdot 10^{10}} \text{ sec} = 0.5 \times 10^{-10} \text{ sec} = 0.05 \text{ nsec}$$

Si indichi con d la distanza tra la CPU e la memoria. Affinché l'attesa sia di 1 secondo deve risultare

$$d = c \cdot \tau \leq 3 \cdot 10^{11} \text{ mm/sec} \cdot \frac{1}{2 \cdot 10^{10}} \text{ sec} \approx 15 \text{ mm}$$

avendo indicato con c la velocità della luce.

In definitiva per avere la risposta in 1 secondo bisogna avere a disposizione un computer piccolo più o meno quanto quello rappresentato in Fig. 7, con un periodo di clock $\mu = 0.05 \text{ nsec}$.

Da quanto fino ad ora illustrato, anche se con grande semplificazione, risulta evidente che con i calcolatori mono processore non è possibile ottenere “*alte prestazioni*”. L'introduzione del parallelismo, ovvero del calcolo parallelo, cioè la decomposizione del problema sottoproblemi risolti contemporaneamente su più calcolatori, ha reso possibile il superamento di limiti tecnologici.

Oggetto di discussione dei capitoli seguenti saranno appunto alcuni aspetti metodologici alla base del Calcolo Parallelo. In particolare, verranno discusse alcune problematiche preliminari:

- Può, e deve, essere parallelizzata una data applicazione?
- Se sì, come questo deve essere fatto?
- Qual è il calcolatore più adatto?
- Quali sono gli algoritmi opportuni?
- Quale tecnologia software usare? (linguaggio, compilatore, programma di comunicazione)

In altre parole ciò significa:

- I Identificare le componenti del calcolo che possono “*girare*” in parallelo.
- II Adottare una strategia opportuna per decomporre il problema (algoritmo, programma) in componenti parallele.
- III Scegliere il modello di programmazione (per esempio MPI).
- IV Scegliere uno stile che dipenda dalle applicazioni e dal modello di programmazione scelto.

Come vedremo, verrà data particolare attenzione al concetto di ***decomposizione*** ovvero alla suddivisione di un problema in più sottoproblemi, alla suddivisione dei dati e dei calcoli in più processi (o *tasks*); la decomposizione rappresenta l’idea di base del Calcolo Parallelo⁵. In particolare, nel **primo capitolo** verranno introdotte le metodologie di base del Calcolo Parallelo.

Nel **secondo capitolo** viene introdotto lo schema funzionale di alcuni tipi di architetture parallele e illustrate le caratteristiche fondamentali dei calcolatori paralleli.

Nel **terzo capitolo** mediante alcuni esempi viene spiegato uno dei paradigmi più utilizzati per la comunicazione tra i processi: il message passing.

⁵ Altri concetti importanti alla base del Calcolo Parallelo, come vedremo nel seguito, sono la *concorrenza* e la *scalabilità*.

Nel **quarto capitolo** vengono definiti i parametri di valutazione per il software in ambiente parallelo.

Nel **quinto capitolo** vengono discussi i principi alla base della progettazione di software parallelo per il calcolo matriciale.

Infine, nel **sesto capitolo** viene presentata una panoramica delle principali architetture parallele che dagli anni '80 fino ad oggi sono state presenti sul mercato.

*"A grand challenge is a fundamental problem
in science and engineering, with broad applications,
whose solution would be enabled by the application of
High Performance Computing technology,
which could become available in the near future."*

**NASA, High Performance Computing and
Communication Program, 1993**

*"Una grande sfida è un problema fondamentale
delle scienze e dell'ingegneria, che ha vaste applicazioni e
la cui soluzione sarà possibile attraverso l'uso
di tecnologie di Calcolo ad Alte Prestazioni,
che saranno rese disponibili nell'immediato futuro."*

Capitolo 1

Calcolo parallelo: l'idea di base

Una misura delle prestazioni del software è fornita dal tempo necessario alla sua esecuzione in uno specifico ambiente di calcolo (efficienza del software).

In generale, una rappresentazione semplificata¹ di tale tempo è:

$$\tau \cong k \cdot T(n) \cdot \mu$$

dove:

- $T(n)$ è la funzione complessità di tempo, definita dal numero di operazioni richieste dall'algoritmo;
- μ è il tempo di esecuzione di 1 operazione e dipende dal *periodo di clock*;
- k è un fattore di proporzionalità.

¹Tale rappresentazione non tiene conto di alcuni fattori che dipendono dall'architettura utilizzata.

Le fasi di elaborazione in un calcolatore sono scandite dal segnale di clock emesso per sincronizzare le varie unità. L'intervallo temporale che intercorre tra due impulsi successivi è detto periodo (o ciclo) di clock.

Un'importante caratteristica di un calcolatore è la *peak performance*, P_{max} , che specifica il numero massimo di operazioni floating point che possono essere teoricamente eseguite nell'unità di tempo (generalmente il secondo).

La peak performance si può calcolare come il rapporto tra il massimo numero N_c di operazioni che possono essere eseguite in un ciclo di clock e il tempo di ciclo T_c :

$$P_{max} = \frac{N_c}{T_c}$$

Essa è misurata in flop/sec (floating-point operations per second) o in Mflop/sec (10^6 flop/sec), Gflop/sec (10^9 flop/sec) e in Tflop/sec (10^{12} flop/sec).

Ad esempio il tempo di ciclo della workstation IBM Power2 modello 591 è $T_c = 13 \text{ nsec}$. Possono essere eseguite ad ogni ciclo di clock 4 operazioni floating point (una moltiplicazione e un'addizione). Quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{4 \text{ operazioni}}{13 \cdot 10^{-9} \text{ secondi}} = 308 \text{ Mflop/sec}$$

Il Pentium III esegue una operazione floating point e $T_c = 650 \text{ nsec}$, quindi

$$P_{max} = \frac{N_c}{T_c} = \frac{1 \text{ operazione}}{65 \cdot 10^{-10} \text{ secondi}} = 650 \text{ Mflop/sec}$$

Il processore Athlon esegue due operazioni floating point e $T_c = 600 \text{ nsec}$, quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{2 \text{ operazioni}}{6 \cdot 10^{-11} \text{ secondi}} = 1200 \text{ Mflop/sec}$$

Il processore Power 3 esegue 4 operazioni floating point e $T_c = 375 \text{ nsec}$, quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{4 \text{ operazioni}}{375 \cdot 10^{-9} \text{ secondi}} = 1500 \text{ Mflop/sec}$$

La frequenza del processore, in termini di Hertz, indica quanti cicli vengono eseguiti in un secondo.

Se rapportiamo tale quantità al numero di cicli p necessari per eseguire una operazione otteniamo il numero di operazioni eseguite al secondo, cioè:

$$\frac{[MHz]}{p} = [Mflop/sec]$$

Tenuto conto di tale relazione è possibile calcolare la *peak performance* nel modo seguente:

$$P_{max} = N_c \times Hz$$

È quindi chiaro che la *peak performance* dipende strettamente dal fattore N_c ovvero dal numero massimo di operazioni floating point che possono essere eseguite in un ciclo di clock. In altre parole la frequenza del processore non basta a determinare le sue prestazioni massime.

I Mflop/sec possono essere utilizzati per confrontare le prestazioni di macchine diverse. Un *benchmark* è un programma scritto appositamente a tale scopo. Esistono varie collezioni di benchmarks. LINPACK, un package per l'algebra lineare, è spesso utilizzata per confrontare differenti architetture valutando il tempo di esecuzione dell'algoritmo di eliminazione di Gauss su di una matrice di ordine 100.

Il Linpack Benchmark utilizza le routine LINPACK/sgefa e LINPACK/sgesl per la soluzione di sistemi di equazioni lineari con $n = 100$. La prestazione P è determinata dal rapporto tra il numero di operazioni eseguite ($W = 2n^3/3 + 2n^2$) e il tempo effettuato per il calcolo T :

$$P = \frac{W}{T} [flop/sec]$$

Confrontando tale valore con la *peak performance* della macchina si ha un'indicazione della capacità con cui due routines di LINPACK sfruttano le risorse della macchina (<http://www.netlib.org/benchmark/>).

L'efficienza del software dipende quindi dalla complessità di tempo dell'algoritmo e dalle caratteristiche dell'ambiente di calcolo. Un'efficienza maggiore si può ottenere se si riduce la complessità di tempo dell'algoritmo oppure se diminuisce il tempo di esecuzione di una operazione floating point. È possibile dimostrare che per alcune classi di problemi esistono algoritmi con complessità di tempo

minima; tali algoritmi sono, per questo, detti “*ottimali*”. Ad esempio, per il prodotto di due matrici di dimensione $n \times n$, l'algoritmo di Strassen è ottimale² [47], oppure per la valutazione di un polinomio l'algoritmo di Horner è ottimale [43].

Ridurre μ è invece una sfida tecnologica (Nanotecnologia).

²Il numero di operazioni scalari richieste per il prodotto di due matrici di dimensione $n \times n$ è $M(n) = 2n^3 - n^2$. Nel 1969 Strassen ha descritto un algoritmo per la risoluzione del prodotto matrice-matrice che richiede un numero di operazioni di base $S(n) = 7 \cdot 7^{lg n} - 6 \cdot 4^{lg n}$ dove lg è il logaritmo in base 2.

Date due matrici A e B , l'algoritmo di Strassen suddivide le due matrici in 2×2 blocchi:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

e calcola:

$$\begin{aligned} P_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) & C_{11} &= P_1 + P_4 - P_5 - P_7 \\ P_2 &= (A_{21} + A_{22}) \cdot B_{11} & C_{12} &= P_3 + P_5 \\ P_3 &= A_{11} \cdot (B_{12} - B_{22}) & C_{21} &= P_2 + P_4 \\ P_4 &= A_{22} \cdot (B_{21} - B_{11}) & C_{22} &= P_1 + P_3 - P_2 + P_6 \\ P_5 &= (A_{11} + A_{12}) \cdot B_{22} \\ P_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{aligned}$$

Se viene applicato un solo livello dell'algoritmo di Strassen ad una matrice $n \times n$ i cui elementi sono blocchi di dimensione $n/2 \times n/2$ e viene poi utilizzato l'algoritmo standard per le sette moltiplicazioni di blocchi di matrice che occorrono nel calcolo dei P_i , il numero totale di operazioni è:

$$S(n) = 7 \cdot \left[2 \cdot \left(\frac{n}{2} \right)^3 - \left(\frac{n}{2} \right)^2 \right] + 18 \cdot \left(\frac{n}{2} \right)^2 = \frac{7}{4}n^3 + \frac{11}{4}n^2$$

Il rapporto tra $S(n)$ e $M(n)$ è il seguente:

$$\frac{S(n)}{M(n)} = \frac{7n^3 + 11n^2}{8n^3 - 4n^2}$$

che all'umentare di n tende a $\frac{7}{8}$. Questo implica che per matrici sufficientemente grandi l'applicazione dell'algoritmo di Strassen produce un miglioramento del 12,5% rispetto al prodotto matrice-matrice standard.

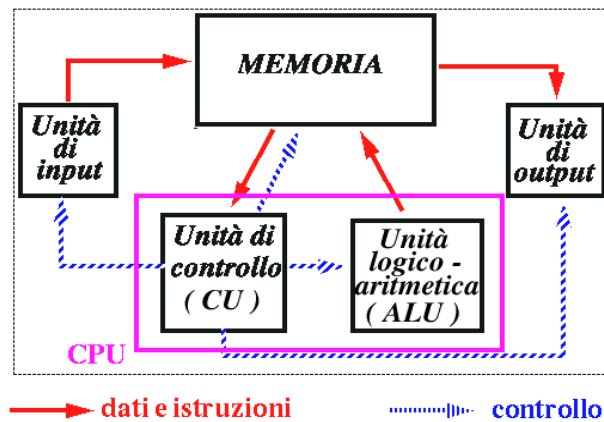


Figura 1.1: Schema funzionale della macchina di Von Neumann

Per quanto riguarda i PC che tipicamente vengono utilizzati in casa, si è passati nel giro di pochi anni dai processori Intel con frequenza di clock di 100 MHz (486) agli attuali Pentium IV, appartenenti alla stessa casa produttrice, con una frequenza di 1.13 GHz. Tenendo conto che frequenza e periodo sono l'uno il reciproco dell'altro, risulta, nel caso del processore Intel, $\mu = 10^{-2}$, e, nel caso del processore Pentium IV, $\mu = 10^{-9}$.

Come ridurre allora ulteriormente il tempo richiesto dalla risoluzione di un problema? L'idea nuova è di organizzare i calcoli in modo diverso.

L'architettura di un calcolatore sequenziale è basata sullo schema funzionale della macchina di Von Neumann descritto in Fig. 1.1. In tale schema una singola unità di elaborazione è preposta all'esecuzione di un insieme di istruzioni. In particolare, è presente una singola unità preposta all'esecuzione delle operazioni floating point (Aritmetic Logic Unit).

♣ Esempio 3

Si consideri la somma di due numeri floating point. Ricordiamo che, fissato il sistema aritmetico floating point, un numero è rappresentato dalla sua mantissa e dall'esponente. La somma di due numeri viene eseguita confrontando inizialmente gli esponenti dei due numeri, viene quindi eseguito uno shift della mantissa del numero con esponente più piccolo e le due mantisse vengono sommate. Infine, il risultato viene normalizzato. Siano $a = 0.956$ e $b = -1.23$, e fissiamo il sistema aritmetico $F = \{10, 3, -2, 20\}$. In tale sistema i due numeri sono così rappresentati:

$$fl(a) = 0.956 \times 10^0$$

$$fl(b) = -0.123 \times 10^1$$

Nei sistemi tradizionali l'operazione di addizione floating point è composta da 4 fasi qui di seguito descritte.

I FASE	<i>confronto esponenti</i>	$esp(+0.956 \times 10^0) < esp(-0.123 \times 10^1)$
II FASE	<i>shift mantissa</i>	$+0.956 \times 10^0 = 0.095 \times 10^1$
III FASE	<i>somma mantisse</i>	$+0.095 - 0.123 = -0.028$
IV FASE	<i>normalizzazione</i>	$-0.028 \times 10^1 = -0.280 \times 10^0$

Mentre viene eseguita una delle quattro fasi i segmenti adibiti all'esecuzione delle altre operazioni rimangono inattivi. Ad esempio, mentre viene eseguito il confronto tra gli esponenti, i segmenti adibiti allo shift della mantissa, alla somma delle mantisse e alla normalizzazione non eseguono alcuna operazione.

△

In uno dei primi articoli introduttivi al Calcolo Parallelo “*Architetture per i supercalcolatori*” di G. Fox e P. Messina del 1987 [23] l'organizzazione dei calcolatori paralleli viene paragonata all'organizzazione di una squadra di operai che sovrintende alla costruzione di una casa. Nella macchina di Von Neumann è come se un operaio

affrontasse l'intero lavoro da solo: egli espleta ognuno dei compiti un passo per volta, eseguendo le parti diverse di ogni lavoro secondo un certo ordine. Questo modo di costruire una casa è ovviamente lento: molti compiti possono essere assolti più velocemente se vengono ripartiti tra muratori diversi simultaneamente all'opera (come di fatto avviene).

In maniera analoga, l'unità funzionale adibita all'addizione f.p., l'ALU, può essere divisa in segmenti, ciascuno dei quali preposto all'esecuzione di una singola fase dell'operazione.

Questa idea è implementata già da diversi anni nei processori con ALU *pipelined* sia sul ciclo delle istruzioni sia su quello delle operazioni [48].

♣ Esempio 4

Si consideri l'esecuzione della somma di N coppie di numeri

$$(a_1, b_1), (a_2, b_2), \dots, (a_N, b_N)$$

su di una unità tradizionale e su di una unità pipelined. La somma delle N coppie di numeri sui due tipi di unità è rappresentata in Fig. 1.2. Sull'unità tradizionale ad ogni passo viene eseguita una fase del processo di somma su di una sola coppia di numeri. Sull'unità pipelined ad ogni passo vengono eseguite contemporaneamente tutte le fasi su coppie distinte di numeri.

Si indichi con t_u il tempo di esecuzione di un segmento dell'operazione. Su di una unità tradizionale solo un segmento alla volta è attivo mentre gli altri rimangono inattivi. Quindi il tempo di esecuzione è :

$$T_{trad} = 4N \cdot t_u$$

Invece, su di una unità pipelined a regime sono attivi tutti i segmenti contemporaneamente: vengono impiegati inizialmente $4t_u$ per andare a regime, successivamente, in $N - 1$ passi, vengono effettuate tutte le somme, quindi:

$$T_{pl} = [4 + (N - 1)] \cdot t_u = 4t_u + (N - 1)t_u = 3t_u + Nt_u = (3 + N)t_u$$

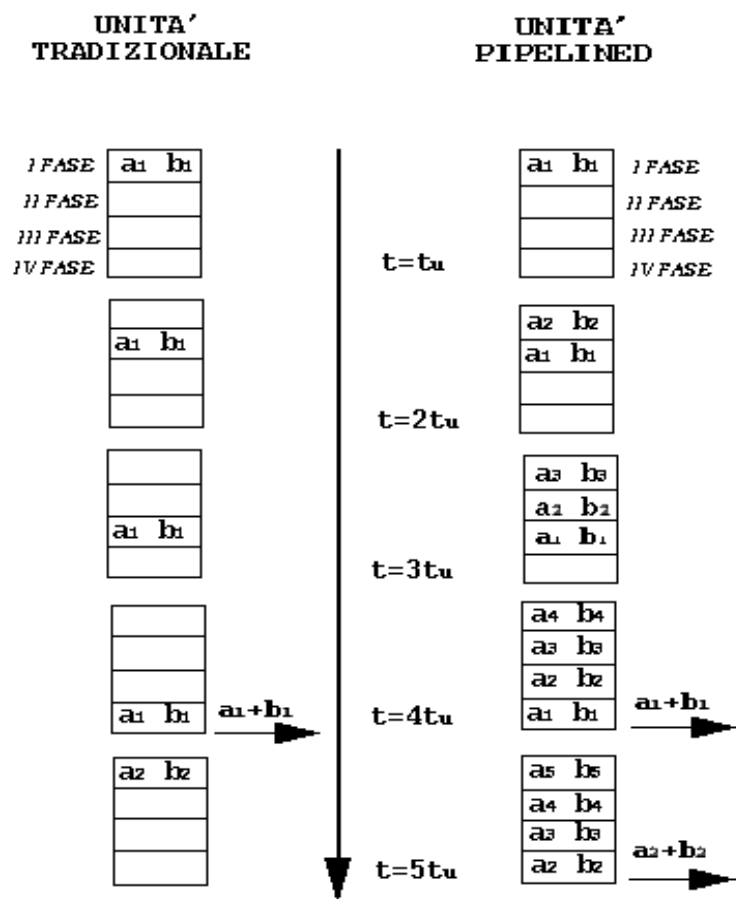


Figura 1.2: Somma di 5 coppie di numeri su di una unità tradizionale e su di una unità pipelined

Essendo:

$$\frac{T_{pl}}{T_{trad}} = \frac{(3 + N) t_u}{4N t_u} = \frac{3}{4N} + \frac{1}{4}$$

al crescere di N si ottiene che:

$$\frac{T_{pl}}{T_{trad}} = \lim_{N \rightarrow \infty} \frac{3}{4N} + \frac{1}{4} = \frac{1}{4}$$

ovvero, il fattore di riduzione del tempo T_{pl} , rispetto al tempo impiegato senza l'uso della pipeline T_{trad} , tende a $\frac{1}{4}$. Tale fattore è pari all'inverso del numero di stadi in cui viene suddivisa la pipeline; infatti nel precedente esempio si avevano 4 stadi. In generale, se s indica il numero di stadi in cui è suddivisa la pipeline, il fattore:

$$\frac{T_{pl}}{T_{trad}} = \frac{1}{s}$$

△

Ritornando all'analogia con la costruzione di una casa, un'unità pipelined è simile a una catena di montaggio in cui più operai eseguono contemporaneamente fasi successive dello stesso lavoro, come mostrato in Fig. 1.3.

Questo tipo di parallelismo si definisce “*parallelismo temporale*” in quanto interviene sulla sequenza temporale in cui uno stesso insieme di dati viene elaborato.

♣ Esempio 5

L'esecuzione di una istruzione consiste di una sequenza di azioni detta “*ciclo delle istruzioni*” (instruction cycle). Tipicamente, un ciclo di istruzioni comprende le seguenti cinque fasi:

1. Instruction Fetch (IF): l'istruzione viene trasferita nei registri;
2. Decoding (D): l'istruzione viene decodificata e interpretata; vengono estratti gli indirizzi degli operandi;
3. Operand Fetch (OF): gli operandi vengono trasferiti nei registri;
4. Execution (EX): gli operandi vengono processati dall'unità aritmetica;
5. Save (SV): il risultato viene trasferito dai registri alla memoria.

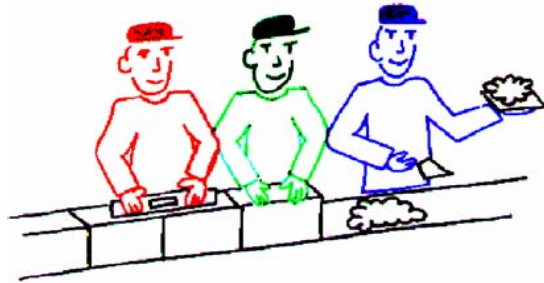


Figura 1.3: Più operai eseguono contemporaneamente fasi successive dello stesso lavoro (parallelismo temporale)

Una schematizzazione del ciclo delle istruzioni è mostrata in Fig. 1.4. Quando i dati o le istruzioni vengono prelevati dalla memoria durante le fasi di IF e OF tutte le unità funzionali del processore rimangono inutilizzate. Anche alla fine, quando il risultato viene trasferito in memoria, le unità funzionali rimangono inattive.

Per migliorare l'utilizzo delle risorse è possibile eseguire le varie fasi del ciclo in pipeline. Esse possono essere eseguite indipendentemente nello stesso tempo, allora l'esecuzione di istruzioni consecutive può essere sovrapposta. Tale procedimento è simile alla catena di montaggio nelle industrie. Nella Fig. 1.5 viene mostrato come vengono eseguite concorrentemente le diverse fasi.

L'esecuzione concatenata delle istruzioni non riduce il numero di cicli di clock necessario per l'esecuzione di una istruzione: il tempo di esecuzione di una istruzione rimane lo stesso. Tuttavia, il numero di cicli di clock necessario per completare un set di istruzioni successive viene ridotto di un fattore corrispondente al numero di stadi (lunghezza) della pipeline.

△

♣ Esempio 6

Esaminiamo la tipologia di un processore RISC (**R**educed **I**nstruction **S**et **C**omputer). L'esecuzione sequenziale delle istruzioni come mostrato in Fig. 1.4 richiede cinque cicli di clock per completare le istruzioni. Nel caso di una pipeline a cinque stadi, come mostrato in Fig. 1.5, l'intervallo minimo per completare istruzioni consecutive è un

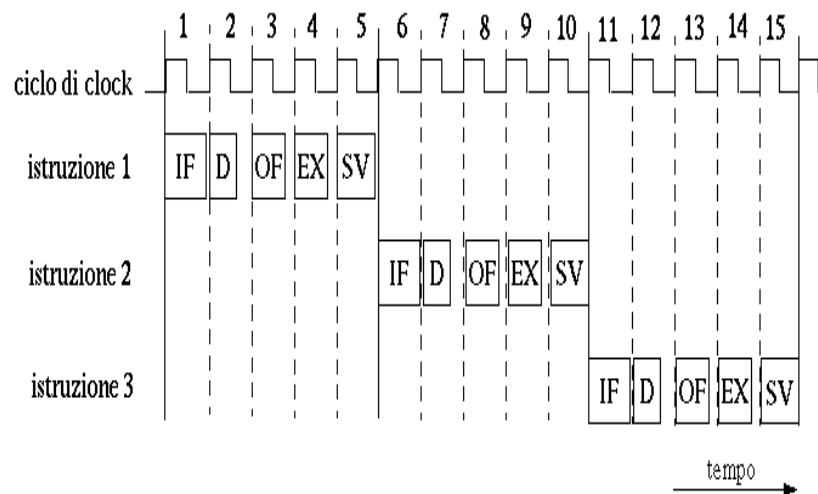


Figura 1.4: Sequenza delle fasi per l'esecuzione di una istruzione.

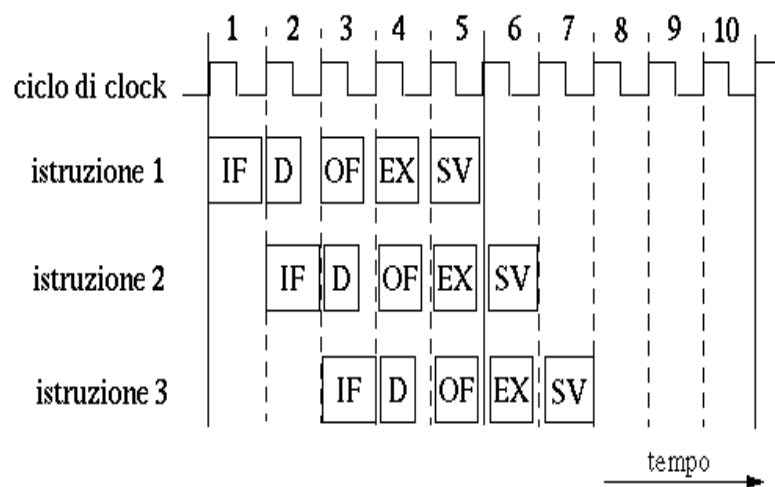


Figura 1.5: Esecuzione contemporanea delle diverse fasi del ciclo sulle istruzioni su di un pipeline a cinque stadi.

ciclo di clock. La riduzione ad $1/5$ del tempo necessario a completare la sequenza di istruzioni in sequenziale è uguale alla lunghezza della pipeline.

△

Il tempo che trascorre tra l'inizio dell'operazione e l'apparire del primo risultato è detto *tempo di latenza* di una pipeline. Nell'esempio citato è $4t_u$. In generale esso dipende dal numero di segmenti da cui è composta la struttura e dal tempo necessario per eseguire ogni compito³. Una struttura a catena con un tempo di latenza elevato è efficiente solo quando la sequenza di dati su cui opera è sufficientemente lunga, solo in tal caso infatti, il termine $(N - 1)t_u$ prevale rispetto a $4t_u$, che, quindi, può considerarsi trascurabile.

Un esempio di calcolatori nel quale il pipeline ha trovato una applicazione efficiente è fornito dai calcolatori vettoriali. Tali calcolatori consentono l'esecuzione di operazioni su vettori mediante l'utilizzo di componenti hardware opportunamente adibite a ciò.

Tutte le soluzioni menzionate in precedenza - strutture pipelined e vettoriali, calcolatori VLIW⁴ - sono utili per incrementare la velocità e l'efficienza di una singola unità di elaborazione. Tuttavia, vi sono applicazioni nelle quali l'uso di una singola unità di elaborazione, per quanto veloce, non è sufficiente a fornire una risposta in un tempo "utile".

³ È possibile costruire pipeline anche per altri tipi di operazioni di base come la moltiplicazione e più in generale concatenare due operazioni diverse. L'architettura RISC 6000/590, di cui si parlerà nel dettaglio più avanti, ha una unità pipeline a 2 passi, uno per la moltiplicazione e l'altro per l'addizione. Ad esempio, dovendo calcolare $a(i) = b(i) * c(i) + d(i)$ con $i = 1, \dots, 4$, al passo $i = 1$ il primo segmento calcola $b(1) * c(1) = e(1)$; ai passi $i = 2, 3, 4$ il primo segmento calcola $b(i) * c(i) = e(i)$, mentre il secondo segmento calcola $e(i - 1) + d(i - 1)$ (il secondo segmento inizierà, quindi, a lavorare al secondo passo); all'ultimo passo, $i = 5$, solo il secondo segmento calcola $e(4) + d(4)$.

⁴ VLIW: Very Long Instruction Word. Il principio di funzionamento delle architetture VLIW, la cui struttura è pipelined, si basa sulla specifica di un certo numero di istruzioni, che costituiscono più *Word*, caricate ed eseguite contemporaneamente dal processore.

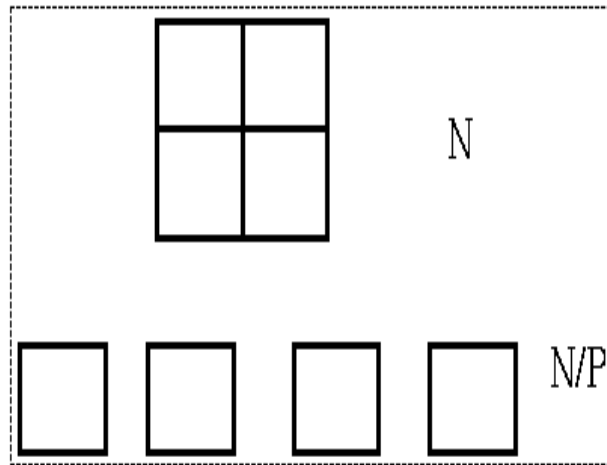


Figura 1.6: *Un problema di dimensione N può essere suddiviso in P sottoproblemi di dimensione N/P da risolvere contemporaneamente.*

Ritorniamo all'analogia con il lavoro degli operai. Nella costruzione di una casa il lavoro viene suddiviso in modo opportuno tra gli operai in base alle competenze e alla necessità di ridurre i tempi di costruzione. L'idea più naturale allora è quella di **decomporre un problema di dimensione N in P sottoproblemi di dimensione N/P e risolverli contemporaneamente su più calcolatori** (Fig. 1.6), riorganizzando opportunamente le componenti del calcolatore. Si possono ad esempio inserire nella CPU più unità adibite alle operazioni aritmetiche (ALU) o connettere due o più processori (CPU) aumentando il numero di istruzioni eseguite nell'unità di tempo o collegare in rete più calcolatori (processore, memoria, I/O) che eseguono la stessa applicazione.

Questa semplice idea è alla base della metodologia del **Calcolo Parallelo**.

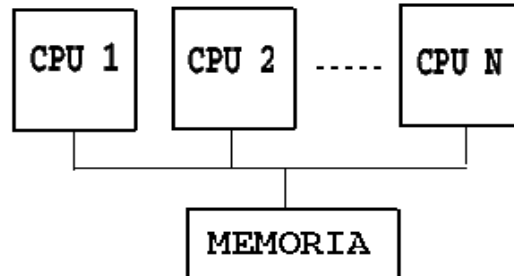


Figura 1.7: Sistema costituito da più CPU che condividono un'unica memoria.

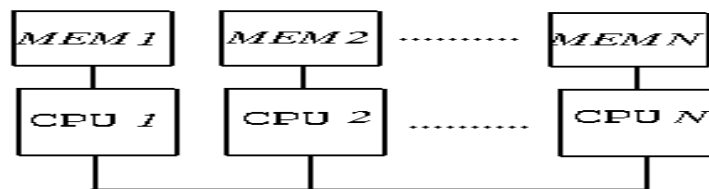


Figura 1.8: Sistema a memoria distribuita.

Nel paragrafo successivo consideriamo un semplice problema, il calcolo della somma di N numeri, e analizziamo come si affronta il progetto di un algoritmo parallelo se si tiene conto dell'architettura che caratterizza l'ambiente di calcolo nel quale l'algoritmo verrà implementato. Introduciamo, per semplicità, due tipi di calcolatori paralleli: quelli che hanno diverse CPU che condividono un'unica memoria (Fig. 1.7) e quelli che sono costituiti da processori ognuno con una propria memoria (Fig. 1.8).

1.1 CASE STUDY: l'operazione di somma

Problema

Calcolo della somma S di $N = 16$ numeri:

$$a_0, a_1, \dots, a_{15}$$

Su un calcolatore tradizionale la somma è calcolata eseguendo $N - 1$ addizioni una per volta secondo un ordine prestabilito, cioè, ad esempio, seguendo l'ordine naturale, si ha:

$$S = a_0 + a_1 + \dots + a_{15}$$

Vediamo come può essere modificato l'algoritmo se eseguito da un calcolatore costituito da più CPU che condividono un'unica memoria; lo schema funzionale di un siffatto calcolatore è schematizzato in Fig. 1.7.

Supponiamo di disporre di quattro processori che indicheremo nel seguito con P_i , $i = 0, 1, 2, 3$. L'idea più naturale è quella di suddividere la somma in somme parziali ed assegnare il calcolo di ciascuna somma parziale ad un processore.

Ogni processore può quindi effettuare la somma parziale ad esso assegnata. Inoltre, tale operazione può essere eseguita da ciascun processore *indipendentemente* dagli altri e *concorrentemente* agli altri. Ad esempio, come schematizzato nella Fig. 1.9, il processore P_0 calcola:

$$s_0 = a_0 + a_1 + a_2 + a_3 ;$$

il processore P_1 calcola:

$$s_1 = a_4 + a_5 + a_6 + a_7 ;$$

$P_0 \rightarrow$	$s_0 = a_0 + a_1 + a_2 + a_3$
$P_1 \rightarrow$	$s_1 = a_4 + a_5 + a_6 + a_7$
$P_2 \rightarrow$	$s_2 = a_8 + a_9 + a_{10} + a_{11}$
$P_3 \rightarrow$	$s_3 = a_{12} + a_{13} + a_{14} + a_{15}$

Figura 1.9: Schema per il calcolo delle somme parziali da parte di ciascun processore.

il processore P_2 calcola:

$$s_2 = a_8 + a_9 + a_{10} + a_{11} ;$$

ed infine il processore P_3 calcola:

$$s_3 = a_{12} + a_{13} + a_{14} + a_{15} .$$

In memoria saranno quindi presenti le somme parziali s_0, s_1, s_2 , ed s_3 . Si pone ora il problema di calcolare la somma totale. Supponiamo che in memoria sia stata definita la variabile $SUMTOT$, che dovrà contenere la somma totale. Le strategie per calcolare la somma totale sono molteplici: un solo processore si può occupare di fare la somma o più processori possono coordinarsi per calcolare la somma totale.

Nel primo caso, ad esempio, il solo processore P_0 accede in memoria alle somme parziali s_i , per $i = 0, \dots, 3$, e calcola:

$$SUMTOT = SUMTOT + s_i \text{ per } i = 0, \dots, 3$$

Nel secondo caso, invece, nasce il problema della **sincronizzazione**. Affinché il valore di $SUMTOT$ sia correttamente aggiornato è necessario che ciascun processore abbia accesso esclusivo e in maniera coordinata a tale variabile durante il suo aggiornamento, cioè è necessario *sincronizzare* gli accessi in memoria. Cosa può succedere se non c'è sincronizzazione negli accessi in memoria?

$$\begin{array}{l} P_1 \rightarrow \text{SUMTOT} = \text{SUMTOT} + s_1 \\ P_3 \rightarrow \text{SUMTOT} = \text{SUMTOT} + s_3 \end{array}$$

Figura 1.10: Operazioni eseguite contemporaneamente da P_1 e P_3

Può ad esempio capitare che entrambi i processori P_1 e P_3 leggano il valore di SUMTOT in memoria e aggiornino tale variabile (Figura 1.10), il processore P_1 calcolando:

$$\text{SUMTOT} = \text{SUMTOT} + s_1$$

e il processore P_3 calconado:

$$\text{SUMTOT} = \text{SUMTOT} + s_3$$

Quindi il processore P_1 sovrascriverà SUMTOT . Lo stesso farà il processore P_3 così che SUMTOT non conterrà il contributo dovuto a s_1 .

Una tecnica per sincronizzare l'accesso in memoria fa uso dei *Semafori*, contatori utilizzati per controllare l'accesso alle risorse condivise. Sono usati come meccanismi di blocco delle variabili per evitare che altri processori accedano alla memoria condivisa contemporaneamente al processore che la sta già utilizzando. In questo caso, un modo per evitare aggiornamenti sbagliati della variabile SUMTOT è stabilire che, per $i = 0, 1, 2, 3$, il processore P_i , e solo lui, acceda alla memoria e calcoli:

$$\text{SUMTOT} = \text{SUMTOT} + s_i$$

Al quarto aggiornamento la variabile SUMTOT conterrà la somma totale.

In definitiva, l'algoritmo parallelo prevede che tutti i processori calcolino la somma parziale e addizionino tale valore alla variabile *SUMTOT* che contiene infine la somma totale.

```

SOMMA DI N=kp NUMERI SU UNA MACCHINA
MIMD - Shared Memory

begin
  forall  $P_i, 0 \leq i \leq p-1$  do
    sumtot := 0
    sum := 0
     $h := i * (n/p)$ 
    for j=h to  $h+(n/p)-1$  do
       $sum := sum + a_j$ 
    endfor
    lock (sumtot)
     $sumtot := sumtot + sum$ 
    unlock (sumtot)
  endforall
end

```

Procedura 1.1 - Algoritmo per il calcolo della somma di N numeri.

L'algoritmo riportato nella *Procedura 1.1* apparentemente sembra, per la maggior parte delle operazioni, simile ad uno sequenziale. Fanno eccezione le istruzioni *forall*, *lock* e *unlock*.

La direttiva *forall* indica che le operazioni che seguono devono essere eseguite da tutti i processori, la direttiva *lock* sincronizza gli accessi in memoria: fa sì che solo un processore per volta abbia accesso alla variabile SUMTOT. La direttiva *unlock* fa terminare la sincronizzazione degli accessi in memoria.

Esaminiamo ora l'esecuzione dello stesso problema su un calcolatore costituito da più CPU ognuna dotata di una propria memoria (Fig. 1.8).

Supponiamo che ogni processore P_i , con $i = 0, \dots, 3$, abbia, nella

$P_0 \rightarrow$	$s_0 = a_0 + a_1 + a_2 + a_3$
$P_1 \rightarrow$	$s_1 = a_4 + a_5 + a_6 + a_7$
$P_2 \rightarrow$	$s_2 = a_8 + a_9 + a_{10} + a_{11}$
$P_3 \rightarrow$	$s_3 = a_{12} + a_{13} + a_{14} + a_{15}$

Figura 1.11: Schema per il calcolo delle somme parziali in un sistema costituito da più CPU.

propria memoria, i valori con cui calcolare la corrispondente somma parziale s_i . Quindi (Figura 1.11), come prima, il processore P_0 calcola:

$$s_0 = a_0 + a_1 + a_2 + a_3 ;$$

il processore P_1 calcola:

$$s_1 = a_4 + a_5 + a_6 + a_7 ;$$

il processore P_2 calcola:

$$s_2 = a_8 + a_9 + a_{10} + a_{11} ;$$

e infine il processore P_3 calcola:

$$s_3 = a_{12} + a_{13} + a_{14} + a_{15} .$$

In questo caso ogni processore contiene nella propria memoria i 4 numeri da sommare e la corrispondente somma parziale s_i .

Come calcolare la somma totale? Le strategie possibili sono diverse.

I strategia:

ogni processore calcola la sua somma parziale e invia tale valore ad un processore prestabilito, ad esempio il processore P_0 , che conterrà la somma finale.

Di seguito denoteremo con il termine “*passo*” il “*passo temporale*”, ciascun passo è scandito da unità di tempo successive.

Al I passo il processore P_1 “*invia*” al processore P_0 la sua somma parziale s_1 e il processore P_0 calcola:

$$s_0 = s_0 + s_1$$

Al II passo il processore P_2 “*invia*” al processore P_0 la sua somma parziale s_2 e il processore P_0 calcola:

$$s_0 = s_0 + s_2$$

Al III passo il processore P_3 “*invia*” al processore P_0 la sua somma parziale s_3 e il processore P_0 calcola:

$$s_0 = s_0 + s_3$$

In generale al k -esimo passo, $k = 1, 2, 3$, il processore il cui identificativo coincide con il passo corrente, ovvero il processore P_i con $i \equiv k$ “*invia*” al processore P_0 la propria somma parziale s_i e il processore P_0 calcola:

$$s_{0,i} = s_{0,i-1} + s_i$$

con $s_{0,0} = s_0$. Alla fine, dopo 3 passi, P_0 conterrà la somma totale .

Uno schema delle comunicazioni tra i processori è mostrato in Fig. 1.12 , mentre un primo schema dell'algoritmo è riportato nella *Procedura 1.2* .

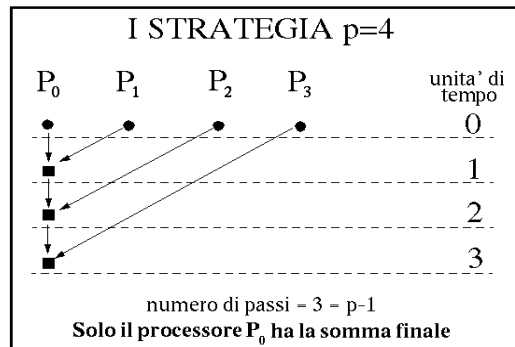


Figura 1.12: Schema delle comunicazioni nella prima strategia.

ALGORITMO 1

SOMMA DI $N=kp$ NUMERI - I STRATEGIA

```

begin
  forall  $P_i$   $0 \leq i \leq p-1$  do
     $h := i * (n/p)$ 
     $s_i := 0$ 
    for  $j=h$  to  $h+(n/p)-1$  do
       $s_i := s_i + a_j$ 
    endfor
    if  $P_0$  then
       $s_{0,0} := s_0$ 
      for  $k = 1$  to  $p-1$  do
         $recv(s_k, P_k)$ 
         $s_{0,k} := s_{0,k-1} + s_k$ 
      endfor
    else if  $P_i$  then
       $send(s_i, P_0)$ 
    endif
  endforall
end

```

Procedura 1.2 - Algoritmo per la somma di N numeri - I Strategia.

La differenza tra questo algoritmo ed uno sequenziale è nelle due istruzioni di *invia* (send) e *ricevi* (recv). L'istruzione $send(s_i, P_i)$ implica l'invio al processore P_i della variabile s_i , mentre l'istruzione $recv(s_i, P_i)$ implica la ricezione della variabile s_i dal processore P_i .

Tale strategia appare immediatamente poco efficace. Si osserva infatti che ad ogni passo solo un processore interagisce con P_0 mentre gli altri due restano in attesa.

II strategia:

ogni processore calcola la sua somma parziale s_i , poi, coppie distinte di processori comunicano tra loro le somme calcolate. In ogni coppia un processore invia all'altro la sua somma parziale. Il risultato finale è in un unico processore prestabilito.

Al I passo P_1 invia s_1 a P_0 e P_0 calcola:

$$s_{0,1} = s_0 + s_1$$

P_3 invia s_3 a P_2 e P_2 calcola:

$$s_{2,3} = s_2 + s_3$$

Al II passo P_2 invia $s_{2,3}$ a P_0 e P_0 calcola:

$$s_{0123} = s_{0,1} + s_{2,3}$$

Alla fine P_0 conterrà la somma totale.

Se è pur vero che la II strategia sia più efficace della I perché i passi temporali sono dimezzati e in ciascun passo coppie distinte di processori operano concorrentemente, un'attenta analisi dell'algoritmo 2 (*Procedura 1.3*) mostra una delle caratteristiche tipiche degli algo-

ritmi strutturati ad albero, ovvero lo **sbilanciamento del carico di lavoro** dei processori. Nella Fig. 1.13 possiamo osservare, infatti, che al secondo passo i processori P_1 e P_3 rimangono inattivi e che alla fine dell'algoritmo solo un processore, P_0 , è attivo.

```

                                ALGORITMO 2
                                SOMMA DI N=kp NUMERI - II STRATEGIA

begin
  forall  $P_i$   $0 \leq i \leq p-1$  do
     $h := i * (n/p)$ 
     $s_i := 0$ 
    for  $j = 1$  to  $\log_2 p$  do
       $s_i := s_i + a_j$ 
    endfor
    for  $k = 1$  to  $\log_2 p$  do
       $q := 2^{k-1}$ 
      forall  $P_i$   $i = q$  to  $p - q$  step  $2^k$  do
         $send(s_i, P_{i-q})$ 
      endforall
      forall  $P_i$   $i = 0$  to  $p - q$  step  $2^k$  do
         $recv(s_{i+q}, P_{i+q})$ 
         $s_i := s_i + s_{i+q}$ 
      endforall
    endfor
  endforall
end

```

Procedura 1.3 - Algoritmo per la somma di N numeri - II Strategia.

III strategia:

ogni processore calcola la sua somma parziale, poi, ad ogni passo successivo, coppie distinte di processori comunicano contemporaneamente. In ogni coppia i processori si scambiano le proprie somme

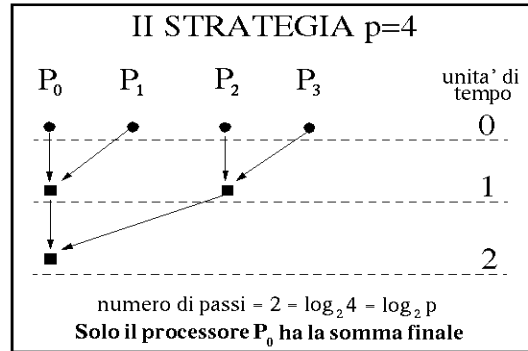


Figura 1.13: Schema delle comunicazioni nella seconda strategia.

parziali. Il risultato finale è in tutti i processori.

Al I passo P_0 e P_1 si scambiano i valori delle proprie somme parziali ed entrambi calcolano:

$$s_{0,1} = s_0 + s_1$$

Analogamente fanno P_2 e P_3 con le rispettive somme parziali.

Al II passo P_0 e P_2 si scambiano i valori delle somme parziali $s_{0,1}$ e $s_{2,3}$ ed entrambi calcolano la somma totale:

$$s_{0123} = s_{0,1} + s_{2,3}$$

Analogamente fanno P_1 e P_3 . Alla fine, tutti i processori hanno la somma totale.

In Fig. 1.14 viene riportato lo schema delle comunicazioni tra i processori mentre, nella *Procedura* 1.4, viene descritto l'algoritmo per la somma di N numeri utilizzando la terza strategia.

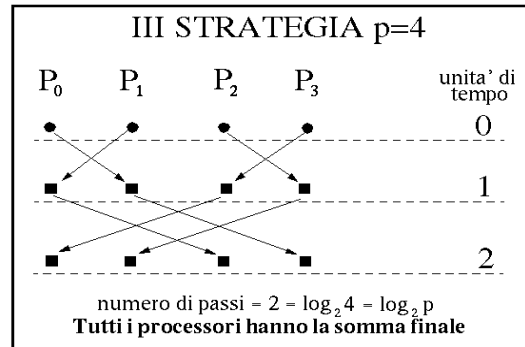


Figura 1.14: Schema delle comunicazioni nella terza strategia.

ALGORITMO 3

SOMMA DI $N=kp$ NUMERI - III STRATEGIA

```

begin
  forall  $P_i$   $0 \leq i \leq p-1$  do
     $h := i * (n/p)$ 
     $s_i := 0$ 
    for  $j = 1$  to  $\log_2 p$  do
       $s_i := s_i + a_j$ 
    endfor
    for  $k = 1$  to  $\log_2 p$  do
       $resto := i - (i/2^k) \times 2^k$ 
      if ( $resto < 2^{k-1}$ ) then
        send( $s_i$ ,  $P_{i+2^{k-1}}$ )
        recv( $s_{i+2^{k-1}}$ ,  $P_{i+2^{k-1}}$ )
         $s_i := s_i + s_{i+2^{k-1}}$ 
      else
        send( $s_i$ ,  $P_{i-2^{k-1}}$ )
        recv( $s_{i-2^{k-1}}$ ,  $P_{i-2^{k-1}}$ )
         $s_i := s_i + s_{i-2^{k-1}}$ 
      endif
    endfor
  endforall
end

```

La III strategia è, praticamente, identica alla II, l'unica differenza risiede nel fatto che in questa strategia le coppie di processori si scambiano i valori calcolati dalle somma parziali in maniera tale che, alla fine, tutti i processori abbiano il risultato finale.

In tutte e tre le strategie ogni processore calcola la propria somma parziale e invia tale valore agli altri processori in modo da ottenere la somma totale. Osserviamo che, alla fine della prima e della seconda strategia, solo il processore P_0 ha il risultato finale, mentre, alla fine della terza strategia lo hanno tutti i processori.

Inoltre, per ottenere il risultato finale nella prima strategia si effettuano $p - 1$ passi, avendo indicato con p il numero di processori, in ciascuno dei passi si esegue 1 operazione, mentre la seconda e la terza impiegano $\log_2 p$ passi. Nella seconda strategia, inoltre, ad ogni passo, vengono eseguite $p - k - 1$ operazioni, avendo indicato con k il generico passo, mentre nella terza strategia in ciascun passo vengono eseguite p operazioni.

Nella Tab. 1.1 sono riassunti il numero di passi, le operazioni ad ogni generico passo k ed il numero totale di operazioni per ciascuna strategia.

	# passi	# operazioni concorrenti eseguite al passo k	# operazioni totali
I Strategia	$p - 1$	1	$p - 1$
II Strategia	$\log_2 p$	$p - k - 1$	$(p - 2) + \dots + (p - \log_2 p - 1)$
III Strategia	$\log_2 p$	p	$p \cdot \log_2 p$

Tabella 1.1: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di operazioni al passo k e totali.

Volendo scegliere una delle tre strategie, basandoci sul numero di passi, siamo indotti a preferire la seconda o la terza strategia, essendo $\log_2 p < p - 1$, mentre, considerando le operazioni totali, la scelta cade sulla prima strategia. In effetti, la valutazione basata soltanto

sul numero di operazioni concorrenti eseguite complessivamente da un algoritmo in ambiente parallelo, è scorretta perché, come vedremo nel capitolo 4, ciò che influenza le prestazioni di un algoritmo in ambiente parallelo è il numero di “*passi temporali*”, non il numero di operazioni eseguite. Quindi, è giusto optare per la seconda o la terza strategia. In altre parole, poiché in ciascun passo temporale, più operazioni sono eseguite concorrentemente, ovvero nello stesso tempo, una corretta valutazione dell'algoritmo della somma in ciascuna delle tre strategie, ad ogni passo temporale deve tener conto del tempo di esecuzione di una sola operazione, come illustrato nella Tab. 1.2.

	# passi	# operazioni concorrenti eseguite al passo k	# operazioni concorrenti
I Strategia	$p - 1$	1	$p - 1$
II Strategia	$\log_2 p$	$p - k - 1$	$\log_2 p$
III Strategia	$\log_2 p$	p	$\log_2 p$

Tabella 1.2: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di operazioni al passo k e totali eseguite in parallelo.

È ora più evidente che la scelta tra le strategie cada sulla seconda o la terza.

Una valutazione completa, in ambiente di calcolo parallelo, non può però prescindere dal numero di comunicazioni, essendo gli algoritmi costituiti da operazioni e comunicazioni. Nella prima strategia viene effettuata una comunicazione ad ogni passo, si hanno così $p - 1$ comunicazioni totali. Nella seconda e terza strategia, invece, ad ogni passo k si hanno più comunicazioni, in particolare $p/2^k$ nella seconda strategia e p nella terza. Esse però avvengono nello stesso passo temporale, quindi sia la seconda sia la terza strategia richiedono un

numero totale di comunicazioni uguale a $\log_2 p$.

Uno schema sul numero di passi e le comunicazioni effettuate nelle singole strategie analizzate è illustrato nella Tab. 1.3.

	# passi	# comunicazioni al passo k	# comunicazioni
I Strategia	$p - 1$	1	$p - 1$
II Strategia	$\log_2 p$	$p/2^k$	$\log_2 p$
III Strategia	$\log_2 p$	p	$\log_2 p$

Tabella 1.3: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di comunicazioni al passo k e totali eseguite concorrentemente.

La II e la III strategia sono, pertanto, quelle che richiedono il minor numero di comunicazioni e di operazioni.

Nell'algoritmo della somma si è evidenziato un primo aspetto fondamentale nella progettazione di un algoritmo per un calcolatore parallelo e che riguarda la gestione **della concorrenza dell'esecuzione delle operazioni**: utilizzando un calcolatore costituito da più CPU che condividono un'unica memoria, attraverso un "*meccanismo di sincronizzazione degli accessi*" in memoria si controlla la concorrenza delle operazioni. Nel caso, invece, di un calcolatore costituito da più CPU ognuna dotata di una propria memoria, la concorrenza delle operazioni viene gestita attraverso esplicite operazioni di "*scambio di messaggi*", che consentono il controllo e la sincronizzazione delle operazioni.

*“Non è degno di uomini eccellenti
perdere ore come schiavi
nell’attività manuale di calcolare,
che potrebbe essere sicuramente
demandata ad una macchina.”*

Eottfried W. Leibniz (1647-1716)

Capitolo 2

Le architetture parallele

2.1 Classificazione delle architetture parallele

Una classificazione ormai standard delle architetture parallele¹ è quella ad opera di Flynn (1966). La classificazione si basa sulla nozione di flusso di “informazione”, dove per “informazione” si intende una istruzione o un dato. In Fig. 2.1 è rappresentata tale classificazione. Sulle colonne è riportato il tipo di flusso di istruzioni. Sulle righe è riportato, invece, il tipo di flussi di dati. Il flusso è “singolo” se viene elaborata una sola istruzione o un solo dato, è “multiplo” se vengono elaborate più istruzioni o più dati.

Nella tipologia SISD (**S**ingle **I**nstruction **S**ingle **D**ata) rientra il calcolatore mono processore. Esso ha un'unica unità di calcolo e di controllo e prevede un unico flusso di dati su cui vengono eseguite le istruzioni. Le architetture SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) hanno una singola unità di controllo e più ALU per eseguire una stessa istruzione su diversi insiemi di dati. Le architetture MISD (**M**ultiple **I**nstructions **S**ingle **D**ata) si riferiscono a calcolatori in grado di eseguire istruzioni diverse su uno stesso insieme di

¹ La classificazione delle architetture parallele descritta in questo paragrafo è volutamente semplificata perché è essenzialmente rivolta a evidenziare gli aspetti funzionali che interessano la computazione parallela.

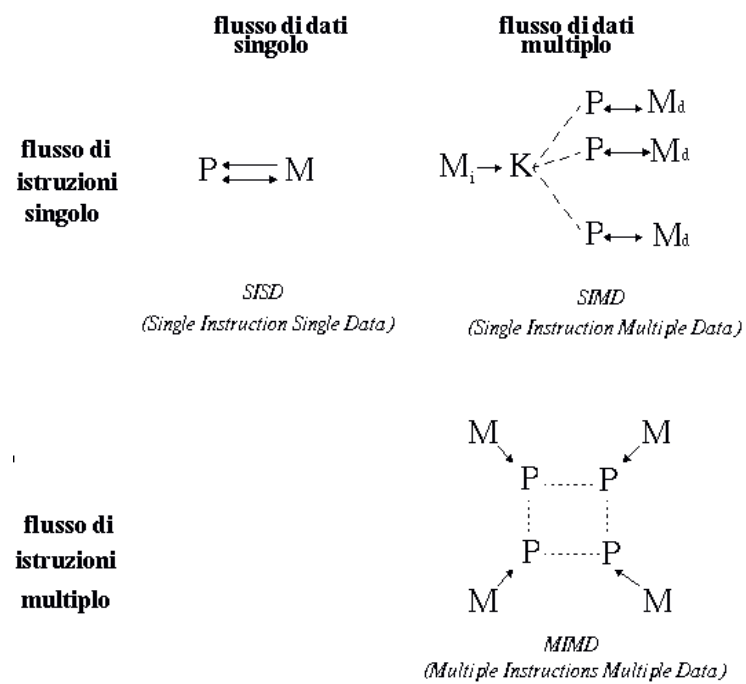


Figura 2.1: *Tassonomia di Flynn. P=unità di calcolo M=memoria*

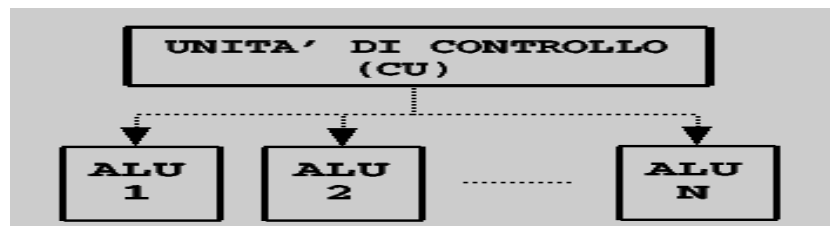


Figura 2.2: Schema funzionale di una macchina SIMD

dati. Alcuni considerano le macchine pipeline come MISD [48, 4]. Infine nella tipologia MIMD (**M**ultiple **I**nstructions **M**ultiple **D**ata) rientrano le architetture costituite da più CPU che consentono l'esecuzione di istruzioni diverse su dati differenti simultaneamente.

In un'architettura SIMD il parallelismo è realizzato a livello di unità funzionale (ALU): più unità aritmetico-logiche operano sotto un controllo comune, come mostrato in Fig. 2.2, eseguendo contemporaneamente la stessa istruzione su dati diversi. Un esempio di architettura SIMD è la Connection Machine 2 della Thinking Machine Corporation [46].

Di nuovo, tornando all'analogia della costruzione di una casa (Capitolo 1), si può pensare a più muratori che eseguono contemporaneamente la stessa azione su mattoni diversi (**parallelismo spaziale**) (Fig. 2.3).

In un'architettura MIMD il parallelismo è realizzato a livello di CPU (vedi Fig. 2.4): più CPU eseguono le proprie istruzioni su dati diversi. Esempio di sistemi MIMD sono l'SP dell'IBM, il CRAY T3D della Cray Research e l'Intel iPSC.

È come se più operai eseguissero contemporaneamente azioni diverse su parti diverse (**parallelismo asincrono**) (vedi Fig. 2.5).



Figura 2.3: Più operai eseguono contemporaneamente la stessa azione su mattoni diversi (parallelismo spaziale)

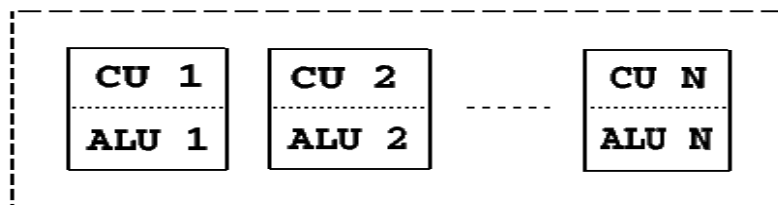


Figura 2.4: Schema funzionale di una macchina MIMD



Figura 2.5: Più operai eseguono contemporaneamente azioni diverse su parti diverse (parallelismo asincrono)

Negli anni successivi alla classificazione ad opera di Flynn, si sono realizzate nuove architetture parallele, caratterizzate dall'essere combinazioni delle due tipologie fondamentali, MIMD e SIMD. Esaminiamo alcune delle nuove architetture.

Le *SMP* (**S**ymmetric **M**ulti **P**rocessor) sono costituite da più processori che operano indipendentemente accedendo alla stessa memoria globale; ogni processore può avere anche una propria memoria (cash), mentre, l'accesso alla memoria condivisa avviene tramite bus o switch.

Nel modello *DSM* (**D**istributed **S**hared **M**emory), invece, la memoria è fisicamente distribuita a livello hardware, ma è globalmente condivisa dal punto di vista software ovvero a livello delle operazioni di accesso (scrittura, lettura) così che all'utente il sistema si presenta con un'unica memoria.

Particolare attenzione merita il *cluster* [1]^a, sistema costituito da calcolatori differenti collegati tra di loro attraverso una rete.

L'idea che ha portato allo sviluppo di tale tipo di sistema ha iniziato a diffondersi negli anni '60 ad opera dell'IBM. L'idea era di collegare tra loro grandi *mainframes* per ottenere sistemi di calcolo ad alte prestazioni più economici.

Tuttavia, i sistemi cluster hanno iniziato a diffondersi negli anni '80 anche in virtù del miglioramento delle prestazioni dei microprocessori, delle reti di connessione e dello sviluppo di strumenti software per il calcolo parallelo e distribuito.

I motivi che per cui i cluster trovano oggi un grande interesse sono i seguenti:

- le workstation sono sempre più potenti a parità di costo;
- le tecnologie di rete stanno migliorando, aumentando la larghezza di banda e diminuendo il tempo di latenza;
- sono più facili da integrare in reti già esistenti;
- sono più economici e costituiscono un'alternativa a piattaforme di calcolo fortemente specializzate;
- possono “scalare” facilmente; la capacità di un nodo può essere aumentata aggiungendo memoria e processori.

I cluster costruiti utilizzando componenti hardware economiche e facilmente reperibili (COTS Commodity off the shelf) e software *free* svolgono un ruolo fondamentale nella definizione di *supercalcolatore* parallelo del tipo Beowulf.

^aCluster, Network of Workstation (NOW) e Cluster of Workstation (COW) sono sinonimi.

I computer che fanno parte di un cluster sono generalmente connessi mediante una rete LAN (**L**ocal **A**rea **N**etwork). Le reti LAN hanno tipicamente un'alta latenza e una bassa larghezza di banda. Un ruolo fondamentale, in questi sistemi, è svolto dallo switch di interconnessione.

I *sistemi distribuiti* sono sistemi molto simili ai cluster. A differenza di questi, però, sono distribuiti su aree geografiche più estese e coinvolgono distanze dell'ordine delle migliaia di chilometri (WAN, **W**ide **A**rea **N**etwork).

Come naturale evoluzione dei sistemi distribuiti sta emergendo il *Grid Computing* [22], la tecnologia di calcolo del XXI secolo. L'idea alla base del Grid computing è di gestire le risorse di calcolo (hw e sw) e le applicazioni condivise tra organizzazioni diverse in maniera dinamica. La condivisione riguarda soprattutto l'accesso diretto alle macchine, al software, ai dati e ad altre risorse (strumenti di acquisizione dati, database, etc. ...). Questa condivisione deve essere necessariamente controllata e l'obiettivo è quindi di realizzare protocolli, servizi e strumenti che consentano una condivisione delle risorse sicura, trasparente ed efficiente tra le organizzazioni.

Un'ulteriore classificazione delle architetture parallele è legata alla modalità di accesso della CPU alla memoria. Distinguiamo, quindi, i sistemi paralleli a memoria condivisa (*shared memory*) da quelli a memoria distribuita (*distributed memory*).

Nei sistemi a memoria condivisa le CPU condividono la stessa memoria come mostrato in Fig. 2.6. In questo caso la memoria può essere fisicamente e globalmente condivisa, esiste cioè un'unica memoria eventualmente fisicamente distribuita nel sistema, ma dal punto di vista funzionale è condivisa da tutti i processori.

Il calcolatore della SGI-CRAY Power Challenge è un sistema a memoria condivisa.

In un sistema a memoria distribuita la memoria è associata ad un singolo processore e ognuno di questi può indirizzare solo la propria memoria. Se un processore richiede un dato dalla memoria di un altro processore, è necessario attivare un trasferimento del dato dalla

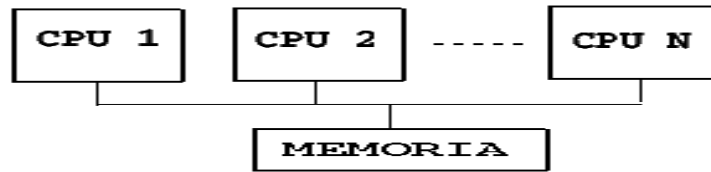


Figura 2.6: *Sistema a memoria condivisa.*

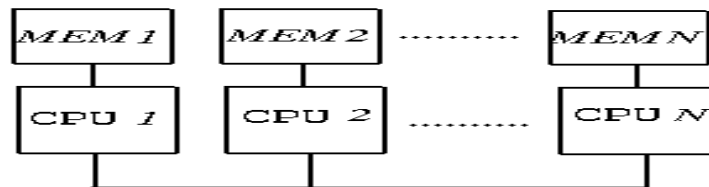


Figura 2.7: *Sistema a memoria distribuita.*

memoria di un processore a quella dell'altro (Fig. 2.7).

Il calcolatore SP dell'IBM e l'Intel iPSC sono esempi di sistemi a memoria distribuita, così come i cluster.

La rete di interconnessione influisce in maniera considerevole sulle prestazioni del sistema. Se questo tempo è identico per tutti gli accessi alla memoria il calcolatore prende il nome di multiprocessore con accesso uniforme alla memoria (UMA, **U**niform **M**emory **A**ccess) (vedi Fig. 2.8), al contrario, se il tempo di accesso alla memoria non è lo stesso, il sistema prende il nome di multiprocessore con accesso non uniforme (NUMA). Un esempio di architettura Shared Memory di tipo UMA è quella di un **S**ymmetric **M**ulti **P**rocessor (**SMP**). Un esempio di architettura Shared Memory di tipo NUMA è quella di un calcolatore **D**istributed **S**hared **M**emory (**DSM**).

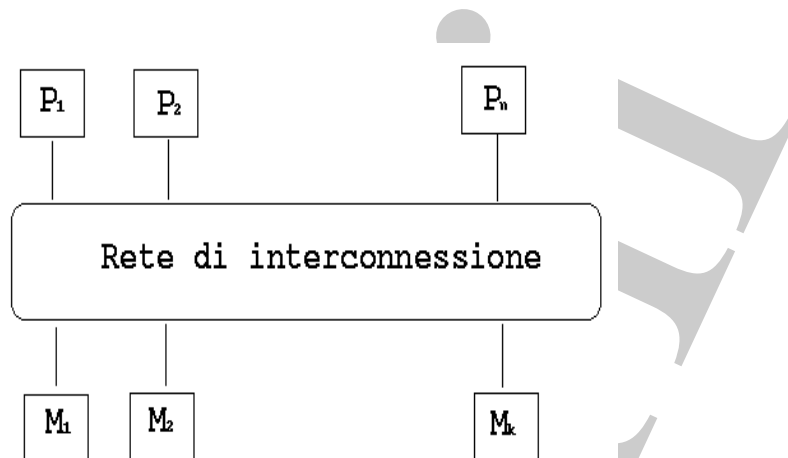


Figura 2.8: *Esempio di multiprocessore UMA.*

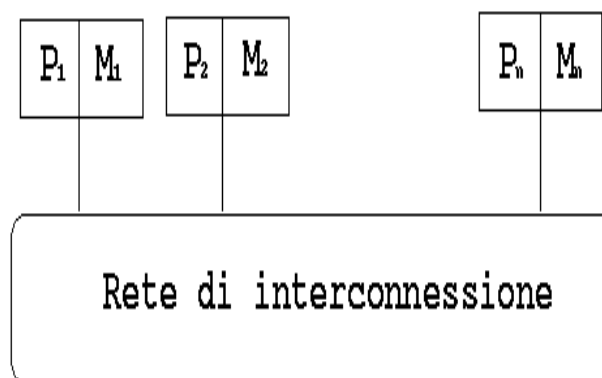


Figura 2.9: *Esempio di multiprocessore NUMA.*

La funzione dell'unità di memoria è contenere programmi e dati. Si distinguono due tipi di dispositivi di memoria, quella principale^a e quella secondaria. La capacità massima della memoria è determinata dallo schema di indirizzamento. Un calcolatore con indirizzi a k bit è in grado di indirizzare fino a 2^k locazioni di memoria. Il numero di locazioni rappresenta la dimensione dello spazio di indirizzamento del calcolatore. I programmi risiedono nella memoria principale durante l'esecuzione: l'istruzione e i dati vengono scritti o letti dalla memoria, sotto il controllo diretto del processore. Il tempo necessario per accedere ad una world costituita da $m \cdot \text{locazioni}$ è detto *tempo di accesso* (T_a) e può essere espresso come:

$$T_a = t_l + m \cdot t_{tw}$$

dove con t_l si intende il tempo di latenza, equivalente al tempo di start up di una comunicazione^b, ed è dovuto solo ai tempi hw e sw di connessione di tutte le componenti coinvolte nella comunicazione; t_{tw} rappresenta il tempo di trasferimento di una locazione, mentre m è il numero complessivo di locazioni che costituiscono la *world*. Nella maggior parte dei calcolatori attuali, il tempo di accesso varia da 10 a 100 nsec^(c). Il trasferimento dei dati fra la memoria e la CPU avviene mediante l'utilizzo di due registri della CPU, chiamati MAR (Memory Address Register), registro degli indirizzi della memoria, e MDR (Memory Data Register), registro dei dati della memoria. Se il registro MAR contiene k bit e il registro MDR ne contiene n , allora l'unità di memoria può contenere fino a 2^k locazioni indirizzabili e durante un "*ciclo di memoria*" un dato di n bit viene trasferito tra memoria e CPU. Questo trasferimento avviene mediante il bus del processore che ha k linee degli indirizzi. Il bus comprende anche le linee di controllo per la lettura, la scrittura e il segnale di completamento della funzione di memoria (Memory Function Completed, MFC) che servono per coordinare i trasferimenti dei dati. Negli ultimi venti anni si è assistito ad una considerevole crescita delle prestazioni dei microprocessori.

^aLa memoria principale è quella che nello schema funzionale della macchina di Von Neumann è direttamente collegata alla CPU.

^bAnche tempo di comunicazione di un messaggio vuoto.

^cUn processore Pentium IV, ad esempio, con una frequenza di clock pari a 1.5 GHz, e con 512 M di RAM ha due livelli di cache, L1 di 8K ed L2 di 256K

Mediamente, la velocità di un microprocessore è raddoppiata ogni 12-18 mesi (legge di Moore^a [38]). D'altra parte la velocità dell'accesso alla memoria non ha avuto lo stesso tasso di crescita facendo così aumentare il *gap* tra le prestazioni dei processori e quella degli accessi in memoria (Fig. 2.10).

Poiché ogni locazione della memoria principale deve essere direttamente accessibile in un tempo brevissimo, molti calcolatori hanno memorie secondarie più lente, più economiche e di solito anche più grandi. Le memorie secondarie sono usate per contenere insiemi di dati molto più grandi di quanto la memoria centrale stessa possa contenere. Esiste una vasta gamma di dispositivi di memoria secondaria che include dischi magnetici, nastri, floppy disk, CD ROM.

Uno schema logico dei diversi tipi di memorie presenti in un calcolatore è descritto in Fig. 2.11. Il livello più alto è costituito dalla memoria a più rapido accesso, ma a costo/bit maggiore. Subito dopo si trova la memoria cache. In un sistema dotato di memoria cache, l'esecuzione di un programma avviene trasferendo istruzione e dati dalla memoria principale alla memoria cache appena sono necessari. Questa operazione viene eseguita ad una velocità relativamente lenta, tipica della memoria principale e dei trasferimenti mediante bus. Dopo questo trasferimento, successive richieste delle stesse istruzioni e dati vengono soddisfatte dalla cache: tutto ciò avviene a una velocità notevolmente superiore. Poiché l'informazione contenuta nella cache viene utilizzata ripetutamente prima di venire sostituita, si ottiene un significativo miglioramento delle prestazioni.

^aLa legge di Moore, formulata nel 1965 da Gordon Moore, uno dei fondatori della Intel, teorizza il raddoppio ogni 12-18 mesi del numero di transistor ospitati in un chip, quindi delle sue prestazioni. Ha caratterizzato l'evoluzione dell'industria del personal computer ed è considerata valida ancora oggi. Seguendo il ritmo della legge, in un solo chip si è addensato un numero sempre crescente di componenti: nel 1964 un chip conteneva 32 elementi; nel 1974 65mila, oggi circa 28 milioni. Come conseguenza la potenza e la velocità dei calcolatori sono andate aumentando in modo esponenziale e allo stesso tempo le dimensioni dei singoli componenti sono diminuite: oggi la media è di 180 nanometri, in altre parole milionesimi di millimetro. È importante sottolineare, comunque, che, proprio grazie al parallelismo implementato nelle componenti hw degli attuali microprocessori, la Legge di Moore continua ad essere ancora vera.

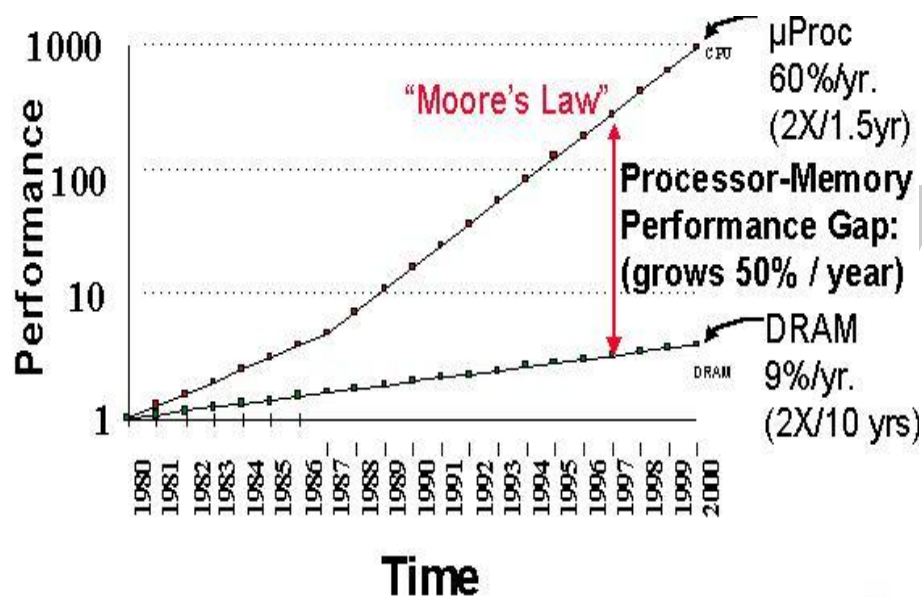


Figura 2.10: Mentre le prestazioni dei microprocessori raddoppiano ogni 12-18 mesi (legge di Moore), la velocità degli accessi in memoria raddoppia ogni dieci anni. Ogni anno il gap processore/memoria aumenta del 50% (<http://www.netlib.org/utk/people/JackDongarra>).

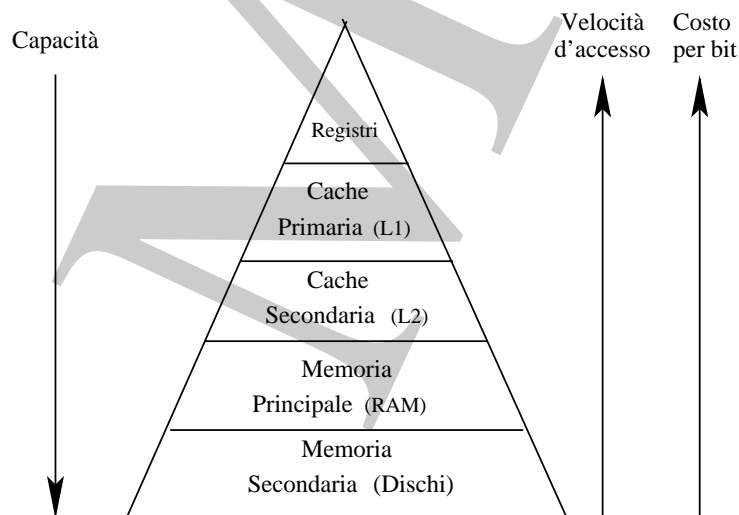


Figura 2.11: Gerarchia della memoria. Le frecce indicano il verso crescente della capacità, della velocità di accesso e del costo per bit delle memorie.

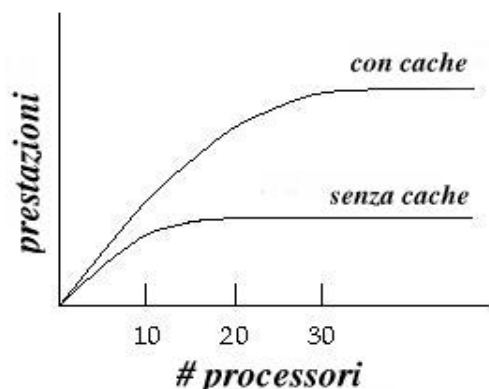


Figura 2.12: Evoluzione della performance (misurata in Mflops) di un sistema di calcolo con e senza memoria cache.

2.2 Reti di interconnessione per sistemi a memoria distribuita

In questo paragrafo esaminiamo alcune possibili realizzazioni di una rete di interconnessione tra nodi, intendendo per nodo il sistema cosistito da CPU e memoria. Il tipo di connessione è detto *topologia* del calcolatore.

In generale, il collegamento deve consentire il trasferimento di informazioni tra le componenti del sistema. Il traffico nella rete è costituito dal trasferimento di dati e istruzioni.

Il tipo di rete da adottare viene valutato in base al costo, alla larghezza di banda, alla quantità effettiva di informazioni trasmesse nell'unità di tempo e dalla facilità di realizzazione. Il termine *larghezza di banda* (bandwidth) fa riferimento alla velocità di un collegamento di trasmissione di trasferimento dati ed è espressa in bit o byte per secondo. La quantità effettiva di informazione trasmessa nell'unità di tempo (*effective throughput*) è la velocità effettiva di trasferimento dei dati.

Possono essere utilizzate topologie differenti per realizzare la rete di interconnessione. Vediamone alcune.

- **Reti ad anello**

Una delle topologie di rete più semplice utilizza un anello per collegare i nodi del sistema, come mostrato in Fig. 2.13.

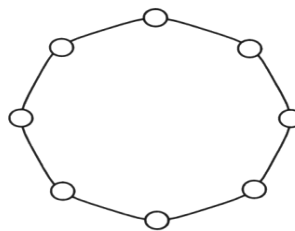


Figura 2.13: *Topologia ad anello (ring)*

Il vantaggio principale di questa organizzazione è costituito dal fatto che l'anello è di facile realizzazione. I collegamenti dell'anello possono essere ampi. Tuttavia, non è utile costruire un anello molto lungo per collegare molti nodi, poiché la latenza del trasferimento di informazioni diverrebbe troppo elevata.

- **Connessione totale**

In questo tipo di topologia ogni nodo ha un legame diretto con tutti gli altri nodi (Fig. 2.14).

Una connessione completa di n nodi fornisce un'ampiezza di banda proporzionale ad n^2 . Infatti, essendo ogni nodo collegato a tutti gli altri nodi si hanno $n(n - 1)$ collegamenti e tutti possono comunicare con tutti. Sfortunatamente anche il costo

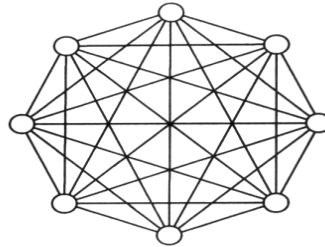


Figura 2.14: *Topologia a comunicazione totale (total connection)*

cresce proporzionalmente ad n^2 , rendendola inadatta per grossi sistemi.

- **Reti a ipercubo**

È un cubo a n dimensioni che collega 2^n nodi (Fig. 2.15). Gli archi del cubo rappresentano collegamenti di comunicazione bidirezionali tra nodi adiacenti. In un cubo n -dimensionale, ogni nodo è collegato direttamente ad n nodi vicini.

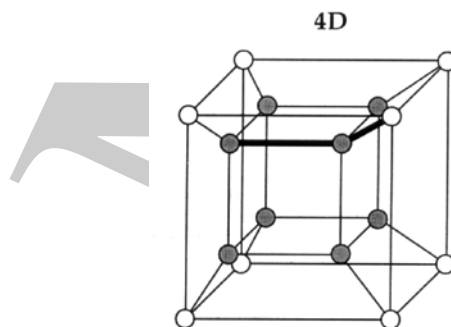


Figura 2.15: *Topologia ad ipercubo*

Tali reti furono utilizzate per numerosi sistemi utilizzando ti-

picamente una trasmissione seriale. Esempi di tali macchine sono iPSC della Intel, NCUBE/10 della NCUBE e CM-2 della Thinking Machines.

- **Reti a matrice (mesh)**

Uno dei modi più naturali di collegare un numero elevato di nodi è l'impiego di una topologia simile ad una matrice, o mesh, come quella rappresentata in Fig. 2.16. Ogni nodo costituisce un elemento della matrice e sono collegati a due a due i nodi adiacenti. Anche in questo caso i collegamenti tra due nodi sono bidirezionali.

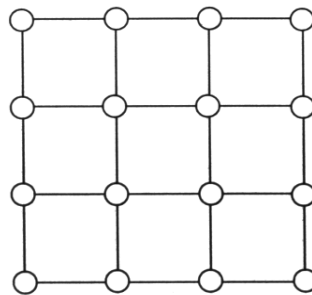


Figura 2.16: *Topologia mesh*

L'instradamento in una rete a matrice è effettuato selezionando il percorso tra un nodo sorgente e un nodo destinatario in modo tale che il trasferimento avvenga prima nella direzione orizzontale. Una volta raggiunta la colonna in cui è presente il nodo destinatario, il trasferimento procede in direzione verticale lungo questa colonna.

Se si effettua una connessione periodica tra i nodi appartenenti a bordi opposti del mesh, il risultato è una rete che comprende

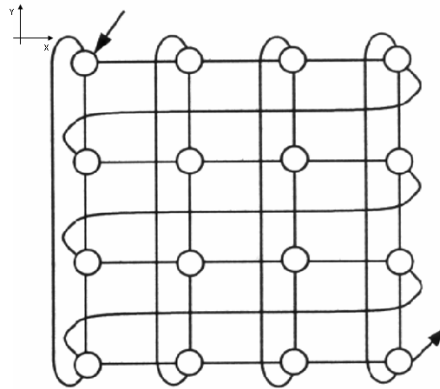


Figura 2.17: *Topologia toroidale*

un insieme di anelli bidirezionali nella direzione x connesso a un insieme di anelli nella direzione y ². Questa rete è chiamata toro ed è rappresentata in Fig. 2.17.

- **Le reti Omega**

Le reti omega (Fig. 2.18) utilizzano interruttori 2×2 ³. L'interruttore ha due input e due output. I messaggi che arrivano sulle due linee in input possono essere commutati sulle due linee in output. Per n CPU ed n memorie sono necessari $\log_2 n$ stadi, con $n/2$ interruttori per stadio, per un totale di $(n/2)\log_2 n$ interruttori.

Per comunicare con un modulo di memoria, la CPU manda un messaggio all'interruttore al primo stadio contenente l'indirizzo del modulo. All' i -esimo stadio l'interruttore prende l' i -esimo bit e lo usa per l'instradamento.

²Le direzioni a cui si fa riferimento sono quelle di un sistema ortogonale $x0y$.

³Le reti Omega sono alla base della connessione tra i nodi della Butterfly.

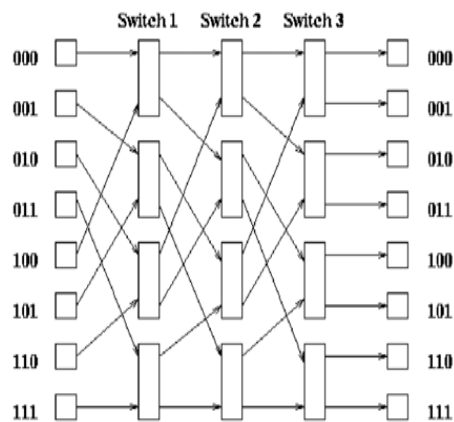


Figura 2.18: Rete omega

2.3 Il principio della località dei dati

Nella maggior parte del software vi sono istruzioni, spesso situate in aree ben localizzate del codice, che vengono eseguite ripetutamente, e si accede al resto delle istruzioni di rado. Se tali segmenti di software potessero risiedere nella memoria cache, il tempo totale di esecuzione verrebbe ridotto in modo significativo. In altre parole, sarebbe opportuno, in fase di memorizzazione delle istruzioni, tener conto del principio della località temporale, ovvero della probabilità che un'istruzione eseguita di recente venga eseguita nuovamente entro breve tempo. Accanto alla località temporale esiste un altro principio di località, ovvero la località spaziale che rappresenta la probabilità che istruzioni vicine⁴ ad un'istruzione già eseguita di recente siano anch'esse eseguite quanto prima. L'aspetto temporale suggerisce di memorizzare un elemento (istruzioni o dati) nella cache quando questo viene richiesto per la prima volta, in modo tale che esso rimanga a disposizione nel caso di una nuova richiesta. L'a-

⁴La vicinanza è espressa in termini di indirizzi delle istruzioni.

spetto spaziale suggerisce di prelevare dalla memoria principale alla cache un insieme di elementi che risiedono in indirizzi adiacenti.

Come vedremo di seguito, bisogna, in fase di progettazione di un algoritmo (sia in ambiente di calcolo sequenziale sia in quello parallelo), tener conto sia della località temporale sia di quella spaziale organizzando opportunamente il flusso delle istruzioni e la fruizione dei dati.

♣ Esempio 7

Si consideri un processore ad 1 *GHz* (1 nsec clock) connesso ad una DRAM con una latenza di 100 nsec. Si assuma, inoltre, che il processore abbia due unità per le operazioni FLOPs⁽⁵⁾ e che sia in grado di eseguire 4 istruzioni in ogni ciclo di 1 nsec. Il picco di prestazione del processore considerato è quindi di 4 *GFLOPS*, mentre il tempo di latenza della memoria è di 100 cicli e la dimensione di blocco è una parola. Con tali caratteristiche, ogni volta che viene effettuata una richiesta alla memoria, si deve attendere 100 cicli prima di poter accedere al dato.

Si consideri il calcolo del prodotto di due matrici A e $B \in \mathbb{R}^{n \times n}$:

$$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \quad \text{con } i, j = 0, n-1$$

dove $C \in \mathbb{R}^{n \times n}$ è la matrice prodotto. Procedendo secondo lo schema di calcolo, effettuando il prodotto delle righe per le colonne, il prodotto $A \cdot B$ richiede n^2 prodotti scalari tra due vettori di dimensione n . L'esecuzione di un prodotto scalare, tra due vettori di dimensione n , richiede una moltiplicazione ed una somma su ogni coppia di elementi corrispondenti dei vettori, ovvero n FLOPs. Per calcolare $A \cdot B$ bisogna, quindi, effettuare $n \cdot n^2$ FLOPs. Se $n = 32$ il numero totale di FLOPs (N_F) è

$$32^3 \approx 32K$$

Ogni FLOPs richiede una fase di fetch, quindi il picco della velocità di calcolo è limitato dalla possibilità di eseguire un FLOPs ogni 100 nsec. Per seguire $N_F = 32K$ FLOPs occorre quindi un tempo T :

$$T = N_F \cdot 100 \text{ nsec} = 32K \cdot 10^2 \text{ nsec} = 32 \cdot 10^2 \mu\text{sec}$$

⁵Un FLOPs coincide con una operazione di somma e una di prodotto:

$$y := y + \alpha \cdot x$$

Si ha così un picco di 10 MFLOPS, una frazione molto piccola della prestazione massima della macchina.

Supponiamo ora che il processore sia dotato di una cache di dimensione 32 KB con una latenza di 1 nsec o di 1 ciclo. Consideriamo nuovamente il prodotto tra le matrici A e B di dimensioni 32×32 .

Si sono scelte con attenzione queste dimensioni in modo che la cache sia abbastanza grande da contenere le matrici A e B oltre alla matrice prodotto C . Inoltre, assumiamo una particolare strategia di allocazione della cache in modo che non vi sia sovrapposizione di dati.

Il caricamento delle due matrici nella cache corrisponde ad una fase di fetch di $2K$ words, che richiede circa $200\text{ }\mu\text{sec}$. Le $32K$ operazioni richieste possono essere effettuate in $8k$ cicli che coincidono con $8\text{ }\mu\text{sec}$, potendo eseguire 4 istruzioni per ciclo. Il tempo totale per la computazione è dato dalla somma del tempo per l'acquisizione dei dati e del tempo per l'esecuzione delle operazioni stesse:

$$T = 200\text{ }\mu\text{sec} + 8\text{ }\mu\text{sec} = 208\text{ }\mu\text{sec}$$

A tale valore di T corrisponde un picco computazionale di $32K/208\text{ }\mu\text{sec} \approx 157$ MFLOPS.

In questo caso l'introduzione di una seppur piccola cache permetta di passare da un picco di 10 MFLOPS a 157 MFLOPS, ovvero si possono migliorare considerevolmente le prestazioni del processore.

Vediamo ora cosa succede se portiamo la dimensione di blocco a 4 words (=1 long words = 1 lwords) nell'eseguire l'operazione di prodotto matrice per matrice⁶. Il caricamento delle due matrici nella cache corrisponde ad una fase di fetch di 512 lwords, che richiede circa $51\text{ }\mu\text{sec}$.

Le $32K$ operazioni richieste possono essere effettuate in $8k$ cicli che coincidono con $8\text{ }\mu\text{sec}$, potendo eseguire 4 istruzioni per ciclo. Il tempo totale per la computazione è dato dalla somma del tempo per l'acquisizione dei dati e del tempo per l'esecuzione delle operazioni stesse:

$$T = 51\text{ }\mu\text{sec} + 8\text{ }\mu\text{sec} = 59\text{ }\mu\text{sec}$$

A tale valore di T corrisponde un picco computazionale di $32K/59\text{ }\mu\text{sec} \approx 555$ MFLOPS.

⁶Si sta ora considerando un'architettura VLIW, introdotta nel capitolo 1

Si è così passati da 10 MPLOPS ai 555 MFLOPS, ottenuti introducendo una memoria cache e l'utilizzo di un blocco di dimensione 4 words.

△

Capitolo 3

Il paradigma dello scambio di messaggi (message passing)

Il modello di programmazione per calcolatori sequenziali è ormai standard. L'algoritmo viene progettato per essere eseguito da un singolo processore che può accedere alla memoria eseguendo una istruzione dopo l'altra¹.

Il “*message passing*” è un modello per la progettazione di algoritmi in un ambiente di calcolo parallelo per calcolatori MIMD a memoria distribuita. Nel progettare l'algoritmo, in questo caso, si deve tenere conto che esistono più processori, ognuno con una “*propria*” memoria, capaci di eseguire ciascuno un “*proprio*” insieme di istruzioni. Chiaramente, i processori devono sincronizzarsi per risolvere un medesimo problema, e ciò avviene attraverso lo scambio di messaggi. Il concetto fondamentale alla base del paradigma *message-passing* è che i processori² comunicano tra loro inviando e ricevendo dati.

Nel prossimo paragrafo illustreremo alcuni algoritmi di calcolo

¹In questo modello si assimilano anche i calcolatori VLSI.

²Con il termine “*processo*” si indica un qualunque programma in esecuzione. Con il termine “*processore*” si indica l'hardware che esegue il processo. Un processore può eseguire più processi. Nel seguito si supporrà che su ciascun processore venga eseguito un singolo processo. Quindi i termini processore e processo verranno utilizzati anche come sinonimi.

matriciale mettendo in evidenza in ciascuno la strategia alla base dell'introduzione del parallelismo.

3.1 CASE STUDY: l'operazione matrice per vettore

Problema

Calcolo del prodotto:

$$Ax = y \quad , \quad A \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^n$$

su un calcolatore MIMD a memoria distribuita con p processori, con A matrice quadrata di ordine n ed x ed y vettori di dimensione n .

L'algoritmo sequenziale più naturale è quello che prevede il calcolo del vettore y componente per componente:

$$y_i = \sum_{j=1}^n A_{i,j} \cdot x_j, \quad \text{con } i = 1, n$$

effettuando i prodotti scalari di ciascuna riga di A per il vettore x , viene così calcolata una componente per volta secondo un ordine prestabilito³. Supponendo di seguire l'ordine naturale delle componenti y_i di y , per $i = 0, n-1$, viene effettuato il prodotto della prima riga di A per il vettore x , ottenendo y_0 , la prima componente di y . Viene poi moltiplicata la seconda riga di A per il vettore x ottenendo così la seconda componente, y_1 , di y , e così via fino a calcolare tutte le n componenti del vettore y . L'algoritmo è descritto nella *Procedura 3.1*.

³ A titolo di esempio è stata presa in considerazione la versione “*per righe*” dell'algoritmo matrice-vettore perché implementa l'usuale definizione “*righe per colonne*” del prodotto di una matrice per un vettore, anche se questa risulta essere non sempre la più efficiente.

```

procedure matvet( $A, x, n, y$ )
integer  $n, i$ 
real  $A(n, n), x(n), y(n)$ 
for  $i = 0$  to  $n - 1$  do
    % inizializzazione a zero della componente
    %  $i$ -esima del vettore  $y$ 
     $y_i := 0$ 
    % prodotto della riga  $i$ -esima della matrice  $A$ 
    % per il vettore  $x$ 
    for  $j = 0$  to  $n - 1$  do
         $y_i := y_i + A_{ij}x_j$ 
    endfor
endfor
return
end matvet

```

Procedura 3.1 - Algoritmo sequenziale per il calcolo del prodotto matrice-vettore (versione righe per colonne).

Osserviamo che il calcolo di ciascun prodotto delle righe della matrice A per il vettore x può essere effettuato in modo indipendente dagli altri prodotti tra le altre righe di A e il vettore x .

Per eseguire la stessa operazione in un ambiente MIMD a memoria distribuita, l'idea più naturale è quella di distribuire il calcolo delle componenti di y , ovvero di distribuire il calcolo degli n prodotti scalari tra le righe di A e il vettore y .

Supponiamo che A sia una matrice quadrata di ordine 6, x e y vettori di lunghezza 6. Supponiamo inoltre di risolvere il problema su $p = 2$ processori. Le operazioni da effettuare sono le seguenti:

1) Distribuzione dei dati.

Disponendo di 2 processori la decomposizione più naturale è quella in cui si divide la matrice A in 2 blocchi di righe: il primo costituito

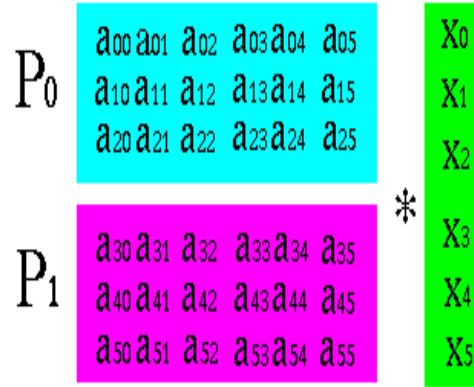


Figura 3.1: Distribuzione dei dati: ad ogni processore viene assegnato un blocco di righe della matrice A e tutto il vettore x .

dalle prime tre righe e il secondo costituito dalle seconde tre righe di A^4 . Il vettore x verrà assegnato invece ad entrambi i processori (tale distribuzione è rappresentata in Fig. 3.1).

Notiamo che, avendo supposto di avere 2 processori con 2 memorie locali, i dati distribuiti vengono memorizzati localmente nella memoria di ciascun processore (Fig. 3.2).

Introducendo una notazione algoritmica⁵, i dati saranno così distribuiti tra i processori:

⁴Analogamente, si può pensare di utilizzare una distribuzione ciclica delle righe di A ai 2 processori; più precisamente la prima riga a P_0 , la seconda a P_1 , la terza a P_0 , la quarta a P_1 e così continuando.

⁵Qui e nel seguito si userà la notazione, tipica, ad esempio, del MATLAB, per indicare una matrice e/o una parte di questa. In particolare, la notazione $A(i : j, :)$ indica che si stanno considerando dalla $(i+1)$ -esima alla $(j+1)$ -esima riga e tutte le colonne della matrice A ; la notazione $A(:, i : j)$ indica che si stanno considerando tutte le righe della matrice A e le colonne dalla $(i+1)$ -esima alla $(j+1)$ -esima.

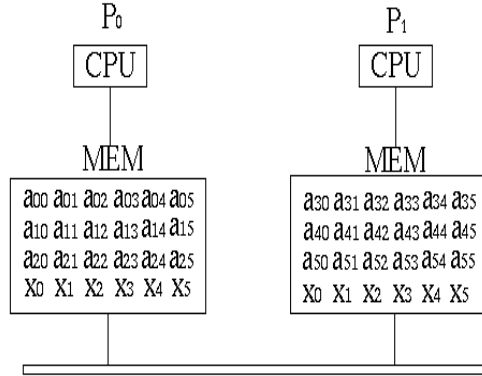


Figura 3.2: Distribuzione degli elementi della matrice A e del vettore x tra i processori. Al processore P_0 vengono assegnate le prime tre righe di A e tutto il vettore x , al processore P_1 vengono assegnate le seconde tre righe di A e tutto il vettore x .

Algoritmo processore P_0	Algoritmo processore P_1
$A_{loc}(0 : 2, :) := A(0:2,:)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$	$A_{loc}(3 : 5, :) := A(3:5,:)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$

A_{loc} , x_{loc} ed y_{loc} sono variabili locali residenti nella memoria di ciascun processore. È importante ricordare che ogni nodo non ha accesso alle variabili locali di un altro nodo e che ognuno dei due processori calcola tre delle componenti del vettore soluzione y .

2) Calcolo dei prodotti parziali.

Ciascun processore effettua il prodotto delle righe di A righe per il vettore y utilizzando i dati residenti nella propria memoria:

Algoritmo processore P_0
$y_{loc}(0) := a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 + a_{04}x_4 + a_{05}x_5$ $y_{loc}(1) := a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5$ $y_{loc}(2) := a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5$
Algoritmo processore P_1
$y_{loc}(3) := a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5$ $y_{loc}(4) := a_{40}x_0 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5$ $y_{loc}(5) := a_{50}x_0 + a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5$

I processori eseguono lo stesso algoritmo su dati diversi, secondo lo schema mostrato nella *Procedura 3.2*.

Algoritmo processore P_0	Algoritmo processore P_1
\vdots for $i=0$ to 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor \vdots	\vdots for $i=3$ to 5 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor \vdots

Procedura 3.2 - Algoritmo per il calcolo delle componenti di y - I Strategia

3) Combinazione dei risultati.

Terminata la fase di calcolo, il processore P_0 ha le prime 3 componenti del vettore soluzione, mentre, il processore P_1 ha le ultime 3. Se richiediamo che ambedue i processori abbiano tutte le componenti del vettore y , dobbiamo considerare uno scambio dei dati tra

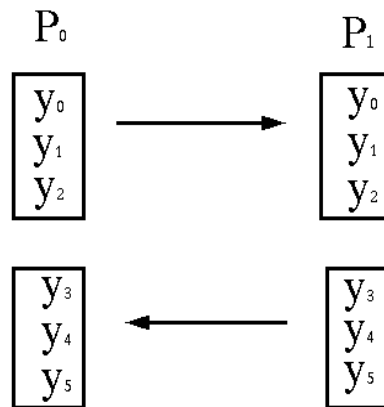


Figura 3.3: In figura è rappresentato lo scambio delle componenti del vettore y tra i due processori.

i 2 processori. L'operazione consta di un solo passo:

il processore P_0 invia al processore P_1 la prima, la seconda e la terza componente del vettore y e il processore P_1 invia al processore P_0 la quarta, la quinta e la sesta componente del vettore y (Fig 3.3). In questo modo entrambi i processori avranno a disposizione tutto il vettore soluzione.

Si hanno, così, le operazioni di comunicazione seguenti:

Algoritmo processore P_0	Algoritmo processore P_1
$\text{send}(y_{loc}(0 : 2), 1)$ $\text{recv}(y_{loc}(3 : 5), 1)$	$\text{send}(y_{loc}(3 : 5), 0)$ $\text{recv}(y_{loc}(0 : 2), 0)$

La funzione *send* ha come argomenti gli elementi da inviare e l'identificativo del processore a cui inviarli. La funzione *recv* ha come argomenti gli elementi da ricevere e l'identificativo del processore da cui riceverli. Gli algoritmi per i due processori sono riportati nella *Procedura 3.3*.

I STRATEGIA	
procedure MATVET (proc. 0)	procedure MATVET (proc. 1)
begin % distribuzione dei dati $A_{loc} := A(0 : 2, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$ % calcolo di alcune componenti % del vettore soluzione for $i=0$ to 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % scambio dei risultati send ($y_{loc}(0 : 2), 1$) recv ($y_{loc}(3 : 5), 1$) end	begin % distribuzione dei dati $A_{loc} := A(3 : 5, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$ % calcolo di alcune componenti % del vettore soluzione for $i=3$ to 5 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % scambio dei risultati recv ($y_{loc}(0 : 2), 0$) send ($y_{loc}(3 : 5), 0$) end

Procedura 3.3 - Algoritmo per il calcolo del prodotto matrice-vettore - I Strategia.

Osserviamo che l'aver introdotto il parallelismo nel calcolo delle componenti di y , ovvero l'aver distribuito tra i 2 processori il calcolo di prodotti scalari delle righe di A per il vettore y , si riflette nell'algoritmo in una decomposizione del primo ciclo iterativo “*for ... end for*”; per il processore P_0 il ciclo su i procede da 0 a 2 mentre per il processore P_1 da 3 a 5. In questo modo il processore P_0 calcola i primi 3 prodotti scalari mentre il processore P_1 gli altri 3.

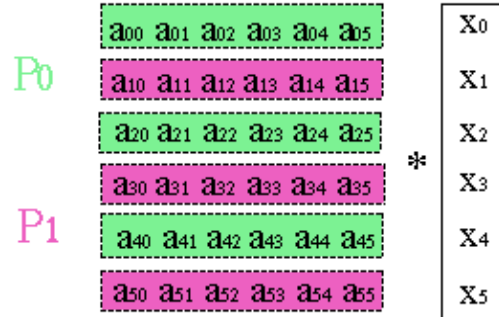


Figura 3.4: Distribuzione dei dati: ad ogni processore vengono assegnate ciclicamente le righe della matrice A e tutto il vettore x .

♣ Esempio 8

Eseguiamo il prodotto

$$A \cdot x = y$$

con $A \in \mathbb{R}^{6 \times 6}$, x e $y \in \mathbb{R}^6$.

1) Distribuzione dei dati.

Invece di suddividere la matrice in 2 blocchi di righe e assegnare il primo blocco al processore P_0 ed il secondo al processore P_1 , assegnamo la prima, la terza e la quinta riga della matrice A al processore P_0 , le righe rimanenti al processore P_1 . Questo tipo di distribuzione è detta *ciclica a blocchi di righe*. Il vettore x verrà assegnato ad entrambi i processori (tale distribuzione è rappresentata in Fig. 3.4).

I dati così distribuiti vengono memorizzati localmente nella memoria di ciascuno dei due processori; tale schema è mostrato graficamente in Fig. 3.5.

Introducendo una notazione algoritmica, i dati saranno così distribuiti tra i processori:

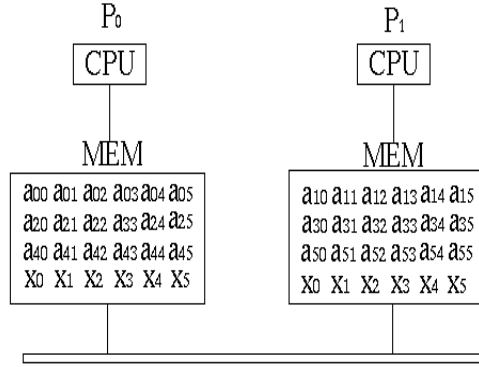


Figura 3.5: Distribuzione degli elementi della matrice A e del vettore x tra i processori. Al processore P_0 vengono assegnate le righe di indice pari della matrice A e tutto il vettore x ; al processore P_1 vengono assegnate le righe di indice dispari della matrice A e tutto il vettore x .

Algoritmo processore P_0	Algoritmo processore P_1
$A_{loc}(0 : 2 : 5, :) := A(0:2:5, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$	$A_{loc}(1 : 2 : 5, :) := A(1:2:5, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$

A_{loc} , x_{loc} ed y_{loc} sono variabili locali residenti nella memoria di ciascun processore. Con tale distribuzione dei dati ogni processore calcola tre delle componenti del vettore soluzione y .

2) Calcolo dei prodotti parziali.

Ciascun processore effettua il prodotto righe per colonne con i dati che possiede nella propria memoria:

Algoritmo processore P_0
$y_{loc}(0) := a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 + a_{04}x_4 + a_{05}x_5$ $y_{loc}(2) := a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5$ $y_{loc}(4) := a_{40}x_0 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5$

Algoritmo processore P_1
$y_{loc}(1) := a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5$ $y_{loc}(3) := a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5$ $y_{loc}(5) := a_{50}x_0 + a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5$

I processori eseguono lo stesso algoritmo su dati diversi, secondo lo schema mostrato nella *Procedura 3.4*.

Algoritmo processore P_0	Algoritmo processore P_1
\vdots for $i=0$ to 5 step 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor \vdots	\vdots for $i=1$ to 5 step 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor \vdots

Procedura 3.4 - Algoritmo per il calcolo delle componenti di y relative ai dati assegnati - Distribuzione ciclica delle righe della matrice A .

3) Combinazione dei risultati.

Terminata la fase di calcolo il processore P_0 ha le componenti 0, 2 e 4 del vettore soluzione, mentre, il processore P_1 ha le componenti 1, 3 e 5. Quindi, deve esserci uno scambio dei dati tra i processori in modo che abbiano ambedue tutte le componenti del vettore soluzione y . L'operazione consta di un solo passo:

Il processore P_0 invia al processore P_1 la prima, la terza e la quinta componente del vettore y e il processore P_1 invia al processore P_0 la seconda, la quarta e la sesta componente del vettore y (Fig. 3.6). In questo modo entrambi i processori avranno a disposizione tutto il vettore soluzione.

Si hanno, così, le operazioni di comunicazione seguenti:

Algoritmo processore P_0	Algoritmo processore P_1
$\text{send}(y_{loc}(0 : 2 : 5), 1)$ $\text{recv}(y_{loc}(1 : 2 : 5), 1)$	$\text{send}(y_{loc}(1 : 2 : 5), 0)$ $\text{recv}(y_{loc}(0 : 2 : 5), 0)$

Gli algoritmi per i due processori sono riportati nella *Procedura 3.5*.

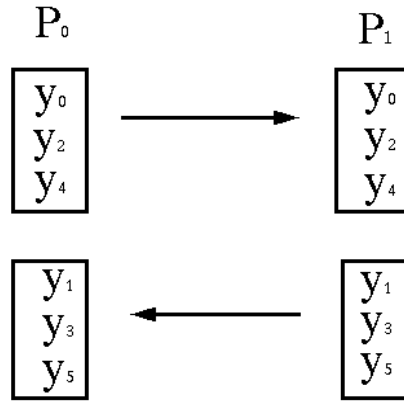


Figura 3.6: In figura è rappresentato lo scambio delle componenti del vettore y tra i due processori.

Distribuzione ciclica delle righe di A	
procedure MATVET (proc. 0)	procedure MATVET (proc. 1)
begin % distribuzione dei dati $A_{loc} := A(0 : 2 : 5, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$ % calcolo di alcune componenti % del vettore soluzione for $i=0$ to 5 step 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % scambio dei risultati $\text{send}(y_{loc}(0 : 2 : 5), 1)$ $\text{recv}(y_{loc}(1 : 2 : 5), 1)$ end	begin % distribuzione dei dati $A_{loc} := A(1 : 2 : 5, :)$ $x_{loc} := x(0 : 5)$ $y_{loc} := y(0 : 5)$ % calcolo di alcune componenti % del vettore soluzione for $i=1$ to 5 step 2 do $y_{loc}(i) := 0$ for $j = 0$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % scambio dei risultati $\text{recv}(y_{loc}(0 : 2 : 5), 0)$ $\text{send}(y_{loc}(1 : 2 : 5), 0)$ end

Procedura 3.5 - Algoritmo per il calcolo del prodotto matrice per vettore - la matrice A è distribuita ciclicamente lungo le righe.

△

La distribuzione dei dati utilizzata nell'esempio 8 è un caso particolare della *Distribuzione Ciclica a Blocchi di Righe*⁶ in cui le righe della matrice A sono suddivise in blocchi di dimensione nb e poi, ciascun blocco viene assegnato ad un processore, in modo che la k -esima riga di A appartenga al processore il cui identificativo è:

$$id = \text{mod}(\lfloor k/nb \rfloor, p)$$

dove p indica il numero di processori ed il simbolo $\lfloor x \rfloor$ la parte intera di x . Nell'esempio 8 $nb = 1$ e $p = 2$.

Una strategia diversa di distribuzione della matrice A si basa sulla decomposizione di A in blocchi di colonne⁷. In tal caso le operazioni da effettuare sono le seguenti:

1) Distribuzione dei dati.

Dividiamo la matrice A in 2 blocchi di colonne: il primo costituito dalle prime tre colonne e il secondo costituito dalle restanti tre colonne. Dividiamo il vettore x in blocchi di righe (Fig. 3.7). In un linguaggio algoritmico, introducendo le variabili locali A_{loc} , x_{loc} e y_{loc} , residenti nella memoria di ciascun nodo, si ha:

Algoritmo processore P_0	Algoritmo processore P_1
$A_{loc}(:, 0 : 2) := A(:, 0 : 2)$ $x_{loc} := x(0 : 2)$ $y_{loc} := y(0 : 5)$	$A_{loc}(:, 3 : 5) := A(:, 3 : 5)$ $x_{loc}(3 : 5) := x(3 : 5)$ $y_{loc} := y(0 : 5)$

Tale suddivisione è rappresentata graficamente in Fig. 3.8.

⁶Tale distribuzione è utilizzata nelle librerie matematiche Blas2 e Blas3 allo scopo di sfruttare al meglio la struttura della matrice A e rendere gli algoritmi ben bilanciati.

⁷Analogamente, anche in questo caso, si può adottare una distribuzione ciclica delle colonne delle matrici; più precisamente la prima colonna a P_0 , la seconda a P_1 , la terza a P_0 , la quarta a P_1 e così continuando.

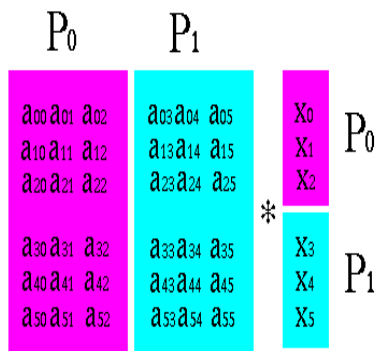


Figura 3.7: Distribuzione dei dati nella II strategia: ad ogni processore viene assegnato un blocco di colonne della matrice A e un blocco di righe del vettore x .

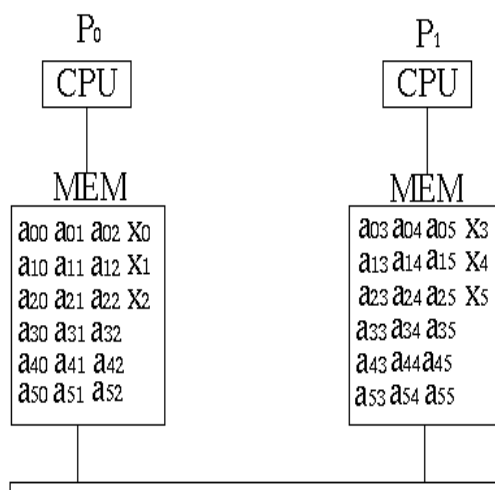


Figura 3.8: Distribuzione degli elementi della matrice A e del vettore x tra i processori. Al processore P_0 verranno assegnate le prime tre colonne di A e le prime tre componenti di x , al processore P_1 verranno assegnate le seconde tre colonne di A e le seconde tre componenti di x .

2) Calcolo dei prodotti parziali.

Ciascun processore effettua il prodotto righe per colonne con i dati che ha nella propria memoria:

Algoritmo processore P_0	Algoritmo processore P_1
$y_{loc}(0) := a_{00}x_0 + a_{01}x_1 + a_{02}x_2$	$y_{loc}(0) := a_{03}x_3 + a_{04}x_4 + a_{05}x_5$
$y_{loc}(1) := a_{10}x_0 + a_{11}x_1 + a_{12}x_2$	$y_{loc}(1) := a_{13}x_3 + a_{14}x_4 + a_{15}x_5$
$y_{loc}(2) := a_{20}x_0 + a_{21}x_1 + a_{22}x_2$	$y_{loc}(2) := a_{23}x_3 + a_{24}x_4 + a_{25}x_5$
$y_{loc}(3) := a_{30}x_0 + a_{31}x_1 + a_{32}x_2$	$y_{loc}(3) := a_{33}x_3 + a_{34}x_4 + a_{35}x_5$
$y_{loc}(4) := a_{40}x_0 + a_{41}x_1 + a_{42}x_2$	$y_{loc}(4) := a_{43}x_3 + a_{44}x_4 + a_{45}x_5$
$y_{loc}(5) := a_{50}x_0 + a_{51}x_1 + a_{52}x_2$	$y_{loc}(5) := a_{53}x_3 + a_{54}x_4 + a_{55}x_5$

I processori eseguono lo stesso algoritmo su dati diversi, secondo lo schema riportato nella *Procedura 3.6*.

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : : for $i = 0$ to 5 do $y_{loc}(i) := 0$ for $j=0$ to 2 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor : : </pre>	<pre> : : for $i = 0$ to 5 do $y_{loc}(i) := 0$ for $j=3$ to 5 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor : : </pre>

Procedura 3.6 - Algoritmo per il calcolo dei prodotti parziali- II Strategia

In questa seconda strategia, come si può osservare dall'algoritmo, l'aver distribuito il calcolo di ciascun prodotto scalare, ovvero l'aver decomposto la matrice A per colonne, si riflette nell'algoritmo nella decomposizione del secondo ciclo "*for ... end for*"; per il processore P_0 il ciclo su j procede da 0 a 2, per il processore P_1 da 3 a 5.

In questo modo ciascun processore può calcolare somme parziali di ciascuna componente del vettore y .

3) Combinazione dei risultati.

Terminata la fase di calcolo, i processori devono collezionare i dati in modo da avere ambedue il risultato finale. Entrambi i processori, P_0 ed P_1 , hanno una somma parziale di tutte le componenti del vettore risultato. Per ottenere, quindi, il risultato finale i due processori devono scambiare tra loro le somme parziali ottenute e sommare le analoghe. L'operazione consta di tre passi:

I PASSO

Il processore P_0 invia al processore P_1 la seconda parte del vettore y e il processore P_1 invia al processore P_0 la prima parte del vettore y (Fig. 3.9). In questo modo entrambi i processori avranno a disposizione i dati di cui hanno bisogno per calcolare il vettore somma finale. Indicata con $aus(0 : 2)$ la variabile locale nella quale ogni processore memorizza la porzione di vettore inviatagli dall'altro, si hanno le operazioni di comunicazione seguenti:

Algoritmo processore P_0	Algoritmo processore P_1
$send(y_{loc}(3 : 5), 1)$	$send(y_{loc}(0 : 2), 0)$
$recv(aus(0 : 2), 1)$	$recv(aus(0 : 2), 0)$

II PASSO

Per far sì che ciascun processore abbia una parte del vettore soluzione y , il processore P_0 calcola la somma delle prime tre componenti del vettore y_{loc} e le tre componenti del vettore aus inviatogli dal processore P_1 , il processore P_1 calcola la somma tra le ultime tre componenti del vettore y_{loc} e le tre componenti del vettore aus inviatogli dal processore P_0 (*Procedura 3.7*)

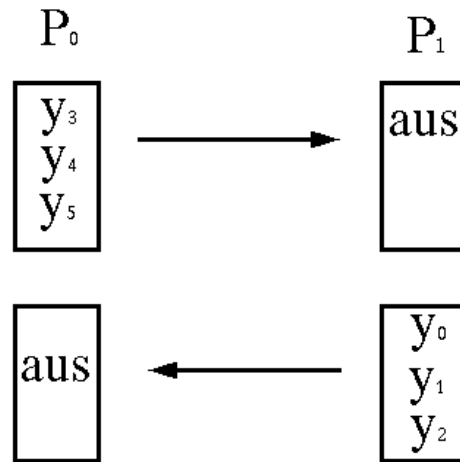


Figura 3.9: Il processore P_0 invia al processore P_1 la seconda parte del vettore y e il processore P_1 invia al processore P_0 la prima parte del vettore y . Il processore che riceve memorizza il dato inviatogli nella variabile locale aus .

Algoritmo processore P_0	Algoritmo processore P_1
\vdots for $i=0$ to 2 do $y_{loc}(i) := y_{loc}(i) + aus(i)$ endfor \vdots	\vdots for $i=3$ to 5 do $y_{loc}(i) := y_{loc}(i) + aus(i - 3)$ endfor \vdots

Procedura 3.7 - Calcolo delle componenti del vettore soluzione y nei due processori - II Strategia

III PASSO

Ciascun processore invia all'altro la propria parte del vettore y aggiornata al passo precedente (Fig. 3.10). In questo modo ogni processore avrà tutte le componenti del vettore prodotto y in y_{loc} .

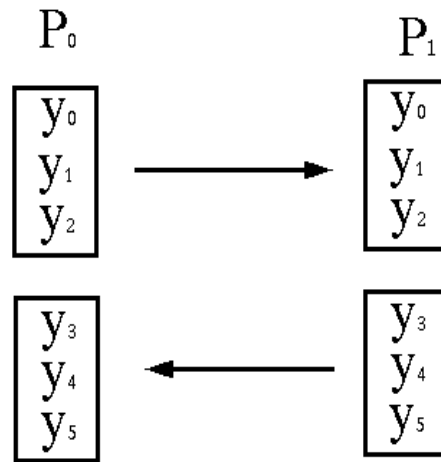


Figura 3.10: Il processore P_0 invia al processore P_1 la prima parte del vettore risultante y e il processore P_1 invia al processore P_0 la seconda parte del vettore risultante y . Entrambi i processori avranno così l'intero vettore risultante y .

Algoritmo processore P_0	Algoritmo processore P_1
$send(y_{loc}(0 : 2), 1)$ $recv(y_{loc}(3 : 5), 1)$	$send(y_{loc}(3 : 5), 0)$ $recv(y_{loc}(0 : 2), 0)$

Gli algoritmi per i due processori sono riportati nella *Procedura 3.8*.

II STRATEGIA	
procedure MATVET proc. 0	procedure MATVET proc. 1
<pre> begin % distribuzione dei dati A_{loc} := A(:, 0 : 2) x_{loc} := x(0 : 2) y_{loc} := y(0 : 5) % calcolo dei prodotti parziali for i = 0 to 5 do y_{loc}(i) := 0 for j = 0 to 2 do y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j) endfor endfor % combinazione dei risultati parziali send(y_{loc}(3 : 5), 1) recv(aus(0 : 2), 1) for i=0 to 2 do y_{loc}(i) := y_{loc}(i) + aus(i) endfor % scambio delle componenti aggiornate send(y_{loc}(0 : 2), 1) recv(y_{loc}(3 : 5), 1) end </pre>	<pre> begin % distribuzione dei dati A_{loc} := A(:, 3 : 5) x_{loc} := x(3 : 5) y_{loc} := y(0 : 5) % calcolo dei prodotti parziali for i = 0 to 5 do y_{loc}(i) := 0 for j = 3 to 5 do y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j) endfor endfor % combinazione dei risultati parziali recv(aus(0 : 2), 0) send(y_{loc}(0 : 2), 0) for i=3 to 5 do y_{loc}(i) := y_{loc}(i) + aus(i - 3) endfor % scambio delle componenti aggiornate recv(y_{loc}(0 : 2), 0) send(y_{loc}(3 : 5), 0) end </pre>

Procedura 3.8 - Algoritmi per il calcolo del prodotto matrice-vettore - II Strategia

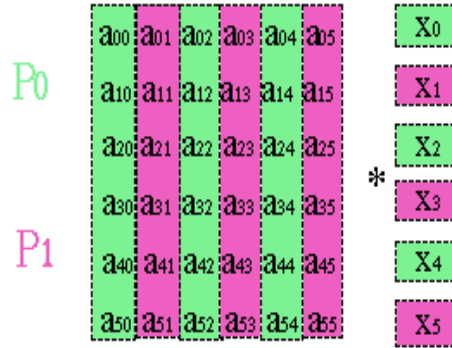


Figura 3.11: *Distribuzione dei dati: ad ogni processore vengono assegnate ciclicamente le colonne della matrice A e le righe del vettore x .*

♣ Esempio 9

Eseguiamo il prodotto

$$A \cdot x = y$$

con $A \in \mathbb{R}^{6 \times 6}$, x e $y \in \mathbb{R}^6$. Le operazioni da eseguire sono le seguenti:

1) Distribuzione dei dati.

Assegnamo la prima, la terza e la quinta colonna della matrice A al processore P_0 , le rimanenti colonne al processore P_1 . Questo tipo di distribuzione è denominata *ciclica a blocchi di colonne*. Le componenti di indice pari del vettore x verranno assegnate al processore P_0 , quelle di indice dispari al processore P_1 (tale distribuzione è rappresentata in Fig. 3.11).

I dati così distribuiti vengono memorizzati localmente nella memoria di ciascuno dei due processori; tale schema è mostrato graficamente in Fig. 3.12.

Introducendo una notazione algoritmica, i dati saranno così distribuiti tra i processori:

Algoritmo processore P_0	Algoritmo processore P_1
$A_{loc}(:, 0 : 2 : 5) := A(:, 0 : 2 : 5)$ $x_{loc}(0 : 2 : 5) := x(0 : 2 : 5)$ $y_{loc} := y(0 : 5)$	$A_{loc}(:, 1 : 2 : 5) := A(:, 1 : 2 : 5)$ $x_{loc}(1 : 2 : 5) := x(1 : 2 : 5)$ $y_{loc} := y(0 : 5)$

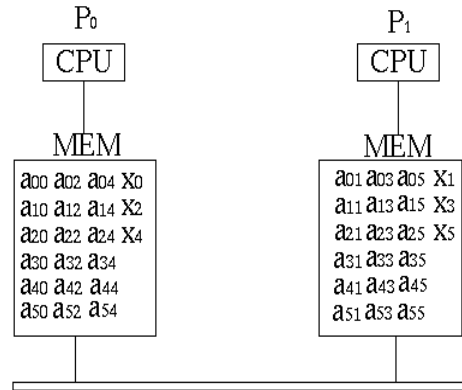


Figura 3.12: Distribuzione degli elementi della matrice A e del vettore x tra i processori. Al processore P_0 vengono assegnate le colonne di indice pari della matrice A e le componenti di indice pari del vettore x ; al processore P_1 vengono assegnate le colonne di indice dispari della matrice A e le componenti di indice dispari del vettore x .

A_{loc} , x_{loc} ed y_{loc} sono variabili locali residenti nella memoria di ciascun processore. Con tale distribuzione dei dati ogni processore calcola tre delle componenti parziali del vettore soluzione y .

2) Calcolo dei prodotti parziali.

Ciascun processore effettua il prodotto righe per colonne con i dati che ha nella propria memoria:

Algoritmo processore P_0	Algoritmo processore P_1
$y_{loc}(0) := a_{00}x_0 + a_{02}x_2 + a_{04}x_4$	$y_{loc}(0) := a_{01}x_1 + a_{03}x_3 + a_{05}x_5$
$y_{loc}(1) := a_{10}x_0 + a_{12}x_2 + a_{14}x_4$	$y_{loc}(1) := a_{11}x_1 + a_{13}x_3 + a_{15}x_5$
$y_{loc}(2) := a_{20}x_0 + a_{22}x_2 + a_{24}x_4$	$y_{loc}(2) := a_{21}x_1 + a_{23}x_3 + a_{25}x_5$
$y_{loc}(3) := a_{30}x_0 + a_{32}x_2 + a_{34}x_4$	$y_{loc}(3) := a_{31}x_1 + a_{33}x_3 + a_{35}x_5$
$y_{loc}(4) := a_{40}x_0 + a_{42}x_2 + a_{44}x_4$	$y_{loc}(4) := a_{41}x_1 + a_{43}x_3 + a_{45}x_5$
$y_{loc}(5) := a_{50}x_0 + a_{52}x_2 + a_{54}x_4$	$y_{loc}(5) := a_{51}x_1 + a_{53}x_3 + a_{55}x_5$

I processori eseguono lo stesso algoritmo su dati diversi, secondo lo schema riportato nella *Procedura 3.9*.

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : : for i = 0 to 5 do $y_{loc}(i) := 0$ for j=0 to 5 step 2 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor : : </pre>	<pre> : : for i = 0 to 5 do $y_{loc}(i) := 0$ for j=1 to 5 step 2 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor : : </pre>

Procedura 3.9 - Algoritmo per il calcolo dei prodotti parziali - Distribuzione ciclica per colonne.

In questa seconda strategia, come si può osservare dall'algoritmo, il parallelismo è stato introdotto nel secondo ciclo “for ... end for”; per il processore P_0 il ciclo su j procede da 0 a 5 di passo 2, per il processore P_1 da 1 a 5 di passo 2. In questo modo ciascun processore può calcolare somme parziali di ciascuna componente del vettore y .

3) Combinazione dei risultati.

Terminata la fase di calcolo, i processori devono collezionare i dati in modo da avere ambedue il risultato finale. Entrambi i processori, P_0 ed P_1 , hanno una somma parziale di tutte le componenti del vettore risultato. Per ottenere quindi il risultato finale i due processori devono scambiare tra loro le somme parziali ottenute e sommare le corrispondenti. L'operazione consta di tre passi che coincidono con i tre passi analizzati al punto 3) nella II Strategia.

Gli algoritmi per i due processori sono riportati nella Procedura 3.10 .

Distribuzione Ciclica per Colonne	
procedure MATVET proc. 0	procedure MATVET proc. 1
<pre> begin % distribuzione dei dati $A_{loc}(:, 0 : 2 : 5) := A(:, 0 : 2 : 5)$ $x_{loc}(0 : 2 : 5) := x(0 : 2 : 5)$ $y_{loc} := y(0 : 5)$ </pre>	<pre> begin % distribuzione dei dati $A_{loc}(:, 1 : 2 : 5) := A(:, 1 : 2 : 5)$ $x_{loc}(1 : 2 : 5) := x(1 : 2 : 5)$ $y_{loc} := y(0 : 5)$ </pre>

Procedura 3.10 - Algoritmi per il calcolo del prodotto matrice-vettore - Distribuzione ciclica per colonne della matrice A - continua

<pre> % calcolo dei prodotti parziali for $i = 0$ to 5 do $y_{loc}(i) := 0$ for $j = 0$ to 5 step 2 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % combinazione dei risultati parziali send($y_{loc}(3 : 5), 1$) recv($aus(0 : 2), 1$) for $i=0$ to 2 do $y_{loc}(i) := y_{loc}(i) + aus(i)$ endfor % scambio delle componenti aggiornate send($y_{loc}(0 : 2), 1$) recv($y_{loc}(3 : 5), 1$) end </pre>	<pre> % calcolo dei prodotti parziali for $i = 0$ to 5 do $y_{loc}(i) := 0$ for $j = 1$ to 5 step 2 do $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ endfor endfor % combinazione dei risultati parziali recv($aus(0 : 2), 0$) send($y_{loc}(0 : 2), 0$) for $i=3$ to 5 do $y_{loc}(i) := y_{loc}(i) + aus(i - 3)$ endfor % scambio delle componenti aggiornate recv($y_{loc}(0 : 2), 0$) send($y_{loc}(3 : 5), 0$) end </pre>
---	---

Procedura 3.10 - Algoritmi per il calcolo del prodotto matrice-vettore - Distribuzione ciclica per colonne della matrice A - fine

△

La distribuzione dei dati utilizzata esempio 9 è un caso particolare della *Distribuzione Ciclica a Blocchi di Colonne* in cui le colonne della matrice A sono prima divise in blocchi di dimensione nb e poi, ciascun blocco, viene assegnato ad un processore, in modo che la k -esima colonna di A appartenga al processore il cui identificativo è:

$$id = \text{mod}(\lfloor k/nb \rfloor, p)$$

dove il simbolo $\lfloor x \rfloor$ la parte intera di x . Nell'esempio 9 $nb = 1$ e $p = 2$.

In sintesi, in un modello message passing possiamo individuare le tre fasi algoritmiche seguenti:

```
begin
  1) distribuzione dei dati
  2) esecuzione delle operazioni concorrenti
  3) combinazione dei risultati
end
```

Sia nella I che nella II strategia, gli algoritmi risolutivi appaiono, a prima vista, differenti. Sembrerebbe quindi necessario scrivere algoritmi diversi da eseguire su ciascun processore.

Il modello di programmazione che prevede la progettazione di un algoritmo per ciascun processore prende il nome di *MPMD* (*M*ultiple *P*rogram *M*ultiple *D*ata) e può essere implementato solo su macchine di tipo MIMD in quanto richiede un flusso di istruzioni multiplo. In tale modello ogni processore esegue un proprio algoritmo indipendentemente dagli altri.

Analizzando più attentamente gli algoritmi mostrati nella *Procedura* 3.10 notiamo che essi si differenziano, in effetti, solo per i dati su cui ogni processore opera, mentre le operazioni da eseguire sono le stesse per entrambi i processori. Tale modello di programmazione prende il nome di *SPMD* (*S*ingle *P*rogram *M*ultiple *D*ata).

Nel modello SPMD, ogni processore esegue lo stesso algoritmo asincronamente e la sincronizzazione avviene solo quando i processori devono scambiarsi i dati. In questo caso, appare naturale domandarsi se sia possibile scrivere un solo algoritmo adatto per tutti i processori. Vediamo quindi come, introducendo opportune variabili, sia possibile scrivere un solo algoritmo che vada bene per tutti e due i processori.

Innanzitutto, osserviamo che tutti i processori devono avere due informazioni fondamentali:

- il numero di processori
- il proprio identificativo

Tale operazione, detta “*inizializzazione dell’ambiente*”, viene denotata con l’istruzione

loc.init (*p*, *myid*)

la quale restituisce in *p* il numero dei processori ed in *myid* l’identificativo del processore che l’ha invocata.

Nel nostro caso, se *loc.init* (*p*, *myid*) viene invocata dai due processori, restituirà ad entrambi $p = 2$; inoltre, al primo processore verrà restituito l’identificativo $myid = 0$, mentre al secondo $myid = 1$. Tali informazioni caratterizzano l’ambiente computazionale parallelo.

Vediamo, quindi come, utilizzando queste due variabili, l’algoritmo che realizza il prodotto matrice-vettore, mostrato nelle *Procedure* 3.11 (I Strategia) e 3.12 (II Strategia), può essere riscritto in un unico modo.

L’algoritmo calcola innanzitutto la dimensione locale, cioè il numero di righe (o di colonne) della matrice assegnata ai processori. Tale dimensione è il rapporto tra il numero complessivo di righe (colonne) della matrice e il numero di processori $k = n/p$ ⁸. Le variabili k_1 e k_2 specificano per ciascun processore l’indice rispettivamente della prima e dell’ultima riga (colonna) del blocco di matrice da assegnare ad ogni processore⁹. Nella seconda strategia le variabili k_1 e k_2 indicano anche le righe del vettore x da assegnare ad ogni processore. Vengono quindi distribuiti i dati e calcolate le componenti del vettore y_{loc} . Ricordiamo che, mentre nella prima strategia il parallelismo viene introdotto nel ciclo esterno su i , nella seconda

⁸Per semplicità di calcolo si assume che n divida esattamente p .

⁹Se ad esempio $n = 9$ e $p = 3$ allora ad ogni processore verranno assegnate $k = n/p = 3$ righe (colonne). In particolare la prima riga (colonna) della matrice che deve essere assegnata al processore P_0 è la riga (colonna) $k_1 = k * myid = 3 * 0 = 0$. La prima riga (colonna) della matrice che deve essere assegnata al processore P_1 è la riga (colonna) $k_1 = k * myid = 3 * 1 = 3$. La prima riga (colonna) della matrice che deve essere assegnata al processore P_2 è la riga (colonna) $k_1 = k * myid = 3 * 2 = 6$.

strategia il parallelismo viene introdotto nel ciclo interno su j , cioè nella prima si effettuano prodotti scalari in parallelo, nella seconda i prodotti vengono “spezzati” in somme parziali. Avviene quindi lo scambio di informazioni tra i processori: ciascuno calcola l’identificativo del processore con cui comunicare, *proccom*, e gli indici dei blocchi che deve inviare e ricevere: $l1$, $l2$, $l3$ ed $l4$. Aggiornato il vettore y_{loc} ogni processore possiede una copia del vettore soluzione y .

Nelle **Procedure** 3.11, 3.12 - 3.13 sono riportati gli algoritmi della I e II strategia rispettivamente, questa volta però osserviamo che l’algoritmo è lo stesso indipendentemente dal processore dal quale viene eseguito.

Procedure MATVET(A,x,y,n) I STRATEGIA

```
begin
% inizializzazione dell'ambiente
  loc.init(p,myid)
% numero di righe da assegnare a ciascun processore
   $k := n/p$ 
% determinazione degli indici delle righe da distribuire
   $k1 := myid \times k$ 
   $k2 := k1 + k - 1$ 
% distribuzione dei dati
   $A_{loc} := A(k1 : k2, :)$ 
   $x_{loc} := x(:)$ 
   $y_{loc} := y(0 : n - 1)$ 
% calcolo delle componenti di  $y_{loc}$  relative ai dati acquisiti
  for  $i = 0$  to  $k - 1$  do
     $y_{loc}(k * myid + i) := 0$ 
    for  $j = 0$  to  $n - 1$  do
       $y_{loc}(k * myid + i) := y_{loc}(k * myid + i) + A_{loc}(i, j) x_{loc}(j)$ 
    endfor
  endfor
% calcolo dell'identificativo del processore con cui comunicare
% e degli indici degli elementi da spedire e ricevere
  if( $myid = 0$ ) then
     $proccom := myid + 1$ 
     $l1 := 0; l2 := n/2 - 1; l3 := n/2; l4 := n - 1$ 
  else
     $proccom = myid - 1$ 
     $l1 := n/2; l2 := n - 1; l3 := 0; l4 := n/2 - 1$ 
  endif
% scambio delle componenti di  $y_{loc}$  calcolate
  send( $y_{loc}(l1 : l2), idcom$ )
  recv( $y_{loc}(l3 : l4), idcom$ )
end
```

Procedura 3.11 - Procedura per il prodotto matrice-vettore su $p = 2$ processori utilizzando la prima strategia.

Procedure MATVET(A,x,y,n) II STRATEGIA

```

begin
% inizializzazione dell'ambiente
  loc.init(p,myid)
% numero di colonne da assegnare a ciascun processore
   $k := n/p$ 
% determinazione degli indici delle colonne di A
% e delle righe di x da distribuire
   $k1 := myid \times k$ 
   $k2 := k1 + k - 1$ 
% distribuzione dei dati
   $A_{loc} := A(:, k1 : k2)$ 
   $x_{loc} := x(k1 : k2)$ 
   $y_{loc} := y(0 : n - 1)$ 
% calcolo delle somme parziali per il vettore  $y_{loc}$ 
  for  $i = 0$  to  $n - 1$  do
     $y_{loc}(i) := 0$ 
    for  $j = 0$  to  $k - 1$  do
       $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ 
    endfor
  endfor
% calcolo dell'identificativo del processore con cui comunicare
% e degli indici degli elementi da spedire e ricevere
  if( $myid = 0$ ) then
     $proccom := myid + 1$ 
     $l1 := 0; l2 := n/2 - 1; l3 := n/2; l4 := n - 1$ 
  else
     $proccom := myid - 1$ 
     $l1 := n/2; l2 := n - 1; l3 := 0; l4 := n/2 - 1$ 
  endif
% combinazione delle somme parziali
  call SUM2( $proccom, y_{loc}, l1, l2, l3, l4$ )
end

```

Procedura 3.12 - Procedura per il prodotto matrice-vettore su $p = 2$ processori utilizzando la seconda strategia.

Procedure $SUM2(idcom, y_{loc}, l1, l2, l3, l4)$

begin

% il proc. invia ad idcom la prima (seconda) metà del
 % “proprio” vettore da aggiornare e ne riceve la seconda (prima)
 % metà da idcom

send($y_{loc}(l3 : l4)$, $idcom$)

recv($aus(0 : n/2 - 1)$, $idcom$)

% ciascun processore esegue la somma di $n/2$ componenti

for $i = l1$ **to** $l2$ **do**

$y_{loc}(i) := y_{loc}(i) + aus(i - l1)$

endfor

% i processori si scambiano parti del vettore

% y_{loc} aggiornato

send($y_{loc}(l1 : l2)$, $idcom$)

recv($y_{loc}(l3 : l4)$, $idcom$)

end

Procedura 3.13 - Procedura per la combinazione dei risultati su $p = 2$ processori.

Generalizziamo le due strategie per il prodotto di una matrice per un vettore al caso in cui $A \in \mathbb{R}^{n \times n}$ con $n = p \times k$ e $k \in \mathbb{N}$ ¹⁰.

I Strategia

Si suddivide la matrice A in p blocchi di k righe ed n colonne:

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} \cdot x = y \text{ con } A_i = \begin{bmatrix} a_{i * \frac{n}{p}, 0} & \dots & a_{i * \frac{n}{p}, n-1} \\ \vdots & \dots & \vdots \\ a_{i * \frac{n}{p} + \frac{n}{p} - 1, 0} & \dots & a_{i * \frac{n}{p} + \frac{n}{p} - 1, n-1} \end{bmatrix}$$

¹⁰Si considera questo caso per semplicità. In generale se n non è multiplo di p , ovvero $n = p \cdot k + r$, con $r < p$ si dovranno distribuire le r righe (colonne) rimanenti ai processori e tener conto del fatto che, essendo $r < p$, non tutti i processori avranno queste ultime righe (colonne).

Il prodotto

$$Ax = y$$

viene decomposto nei prodotti:

$$A_i \cdot x = y_i \text{ con } i = 0, \dots, p-1, A_i \in \mathbb{R}^{k \times n}, x \in \mathbb{R}^n \text{ e } y_i \in \mathbb{R}^k$$

Riscriviamo (*Procedura 3.14*) l'algoritmo sequenziale, mettendo in evidenza le operazioni matriciali relative a ciascun blocco; si ha, in questo modo, la versione “a blocchi” dell'algoritmo nella *Procedura 3.11* (I Strategia).

Procedure MATVET(A,x,y,p,n) A Blocchi di Righe

```
integer p, i, n
real A(n, n), x(n), y(n)
begin
% calcolo delle componenti di y suddivise in p blocchi
  for i = 0 to p - 1 do
    yi = 0
    yi = Ai · x
  endfor
end
```

Procedura 3.14 - *Procedura che esegue il prodotto matrice × vettore considerando i blocchi riga in cui è stata scomposta la matrice A.*

La riformulazione dell'algoritmo in termini di operazioni sui blocchi consente, in modo naturale, la formulazione dell'algoritmo parallelo ottenuto assegnando l'operazione su ciascun blocco ad un processore. Più precisamente, il processore P_i , per $0 \leq i \leq p-1$, ha:

- il blocco $A_i = \begin{bmatrix} a_{i*\frac{n}{p},0} & \dots & a_{i*\frac{n}{p},n-1} \\ \vdots & \dots & \vdots \\ a_{i*\frac{n}{p}+\frac{n}{p}-1,0} & \dots & a_{i*\frac{n}{p}+\frac{n}{p}-1,n-1} \end{bmatrix}$ di A ;

- tutte le componenti del vettore x ;
- tutte le componenti del vettore y ;

Tutti i P_i calcolano un prodotto matrice-vettore:

$$y_i = A_i \cdot x$$

Affinché ciascun processore abbia tutte le componenti del vettore prodotto y è quindi necessario effettuare delle comunicazioni tra i processori.

Se, invece, distribuiamo la matrice A ciclicamente in blocchi di righe di dimensione 1 tra i p processori, il processore P_i , per $0 \leq i \leq p-1$, ha:

- il blocco A_i costituito dalle righe k , per $k = 0, \dots, n-1$, tali che:

$$\text{mod}(k, p) = i$$

- tutte le componenti del vettore x ;
- tutte le componenti del vettore y ;

Quindi, anche in questo caso, tutti i P_i calcolano un prodotto matrice-vettore:

$$y_i = A_i \cdot x$$

in cui il vettore y_i contiene le componenti k , per $k = 0, \dots, n-1$,

del vettore risultante, tali che :

$$\text{mod}(k, p) = i$$

e sono, quindi, necessarie delle comunicazioni tra i processori affinché ciascuno abbia tutte le componenti del vettore prodotto y .

II Strategia

Si suddivide la matrice A in p blocchi di n righe e k colonne ed il vettore x in p blocchi di k componenti:

$$\begin{bmatrix} A_0 & A_1 & \dots & A_{p-1} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{p-1} \end{bmatrix}$$

con

$$A_i = \begin{bmatrix} a_{0,i*\frac{n}{p}} & \dots & a_{0,i*\frac{n}{p}+\frac{n}{p}-1} \\ \vdots & \dots & \vdots \\ a_{n-1,i*\frac{n}{p}} & \dots & a_{n-1,i*\frac{n}{p}+\frac{n}{p}-1} \end{bmatrix}, \quad x_i = \begin{bmatrix} x_{i*\frac{n}{p}} \\ \vdots \\ x_{i*\frac{n}{p}+\frac{n}{p}-1} \end{bmatrix}$$

e

$$r_i = \begin{bmatrix} r_i^0 \\ \vdots \\ r_i^{n-1} \end{bmatrix}$$

Il prodotto

$$Ax = y$$

viene decomposto nei prodotti:

$$A_i \cdot x_i = r_i \quad \text{con} \quad A_i \in \mathbb{R}^{n \times k}, x_i \in \mathbb{R}^k \text{ e } r_i \in \mathbb{R}^n$$

e riscrivendo (*Procedura 3.15*) l'algoritmo analogamente a quanto fatto in precedenza, cioè mettendo in evidenza le operazioni matriciali relative a ciascun blocco, si ha la versione “a blocchi” dell'algoritmo nella *Procedura 3.12* (II Strategia).

Procedure MATVET(A,x,y,p,n) A Blocchi di Colonne

```

integer  $p, i, n$ 
real  $A(n, n), x(n), y(n)$ 
begin
  % inizializzazione a zero del vettore  $y$ 
   $y = 0$ 
  % calcolo del vettore  $y$ 
  for  $i = 0$  to  $p - 1$  do
    % calcolo del contributo  $i$ -esimo per la determinazione di  $y$ 
     $r_i = A_i \cdot x_i$ 
    % aggiornamento del vettore  $y$ 
     $y = y + r_i$ 
  endfor
end

```

Procedura 3.15 - Procedura che esegue il prodotto matrice \times vettore considerando i blocchi colonna in cui è stata scomposta la matrice A .

A partire dalla formulazione a blocchi dell'algoritmo, nasce in modo naturale il corrispondente algoritmo parallelo ottenuto assegnando l'operazione su ciascun blocco ad un processore. Più precisamente, il processore P_i , $0 \leq i \leq p - 1$, ha:

• il blocco $A_i = \begin{bmatrix} a_{0, i * \frac{n}{p}} & \dots & a_{0, i * \frac{n}{p} + \frac{n}{p} - 1} \\ \vdots & \dots & \vdots \\ a_{n-1, i * \frac{n}{p}} & \dots & a_{n-1, i * \frac{n}{p} + \frac{n}{p} - 1} \end{bmatrix}$ di A ;

- il blocco $x_i = \begin{bmatrix} x_{i* \frac{n}{p}} \\ \vdots \\ x_{i* \frac{n}{p} + \frac{n}{p} - 1} \end{bmatrix}$ di x ;

- tutte le componenti del vettore y .

Tutti i processori P_i calcolano:

$$r_i = A_i \cdot x_i, \text{ con } A_i \in \mathbb{R}^{n \times k}, x_i \in \mathbb{R}^k \text{ e } r_i \in \mathbb{R}^n$$

Quindi è necessario, per ottenere il risultato finale, effettuare la somma delle componenti dei p risultati locali distribuiti tra i p processori:

$$y_j = \sum_{i=0}^{p-1} r_i^j \text{ con } j = 0, \dots, n-1$$

Tale operazione richiede delle comunicazioni tra i processori.

Infine, se distribuiamo la matrice A ciclicamente in blocchi di colonne di dimensione 1 tra i p processori, il processore P_i , per $0 \leq i \leq p-1$, ha:

- il blocco A_i costituito dalle colonne k , per $k = 0, \dots, n-1$, tali che:

$$\text{mod}(k, p) = i$$

- le componenti k , per $k = 0, \dots, n-1$, del vettore x tali che:

$$\text{mod}(k, p) = i$$

- tutte le componenti del vettore y ;

Anche in questo caso, tutti i processori P_i calcolano un prodotto

matrice-vettore:

$$r_i = A_i \cdot x_i$$

in cui il vettore r_i contiene le componenti parziali k , per $k = 0, \dots, n-1$, del vettore risultante y , tali che :

$$\text{mod}(k, p) = i$$

Quindi è necessario, per ottenere il vettore finale y , effettuare la somma delle componenti dei p risultati locali r_i distribuiti tra i p processori:

$$y_j = \sum_{i=0}^{p-1} r_i^j \text{ con } j = 0, \dots, n-1$$

3.2 CASE STUDY: somma di p vettori

Nel caso di due processori abbiamo visto che la somma di due vettori è descritta dall'algoritmo riportato nella *Procedura 3.13*.

Generalizzando l'algoritmo al caso in cui i processori sono $p = 2^m$, saranno necessari $m = \log_2 p$ passi. Ad ogni passo, coppie distinte di processori comunicano. Vediamo come determinare ad ogni passo gli identificativi delle coppie di processori che devono comunicare tra di loro. Aiutiamoci con un esempio.

♣ Esempio 10

Determinazione ad ogni passo delle coppie di identificativi di processori che comunicano tra loro.

Consideriamo il caso in cui si hanno $p = 4 = 2^2$ processori (Fig. 3.13).

I passo

$P_0 = 00$ comunica con $P_1 = 01$

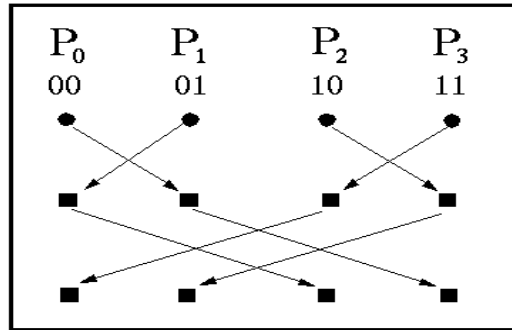


Figura 3.13: Nella figura sono evidenziate le comunicazioni ad ogni passo tra i processori in base al loro identificativo, nel caso in cui $p = 4$.

$P_2 = 10$ comunica con $P_3 = 11$

comunicano i processori il cui identificativo differisce solo nel secondo bit.

II passo

$P_0 = 00$ comunica con $P_2 = 10$

$P_1 = 01$ comunica con $P_3 = 11$

comunicano i processori il cui identificativo differisce solo nel primo bit.

Consideriamo ora il caso in cui $p = 8 = 2^3$.

I passo

$P_0 = 000$ comunica con $P_1 = 001$

$P_2 = 010$ comunica con $P_3 = 011$

$P_4 = 100$ comunica con $P_5 = 101$

$P_6 = 110$ comunica con $P_7 = 111$

comunicano i processori il cui identificativo differisce solo nel terzo bit.

II passo

$P_0 = 000$ comunica con $P_2 = 010$

$P_1 = 001$ comunica con $P_3 = 011$

$P_4 = 100$ comunica con $P_6 = 110$

$P_5 = 101$ comunica con $P_7 = 111$

comunicano i processori il cui identificativo differisce solo nel secondo bit.

III passo

$P_0 = 000$ comunica con $P_4 = 100$

$P_1 = 001$ comunica con $P_5 = 101$

$P_2 = 010$ comunica con $P_6 = 110$

$P_3 = 011$ comunica con $P_7 = 111$

comunicano i processori il cui identificativo differisce solo nel primo bit.

△

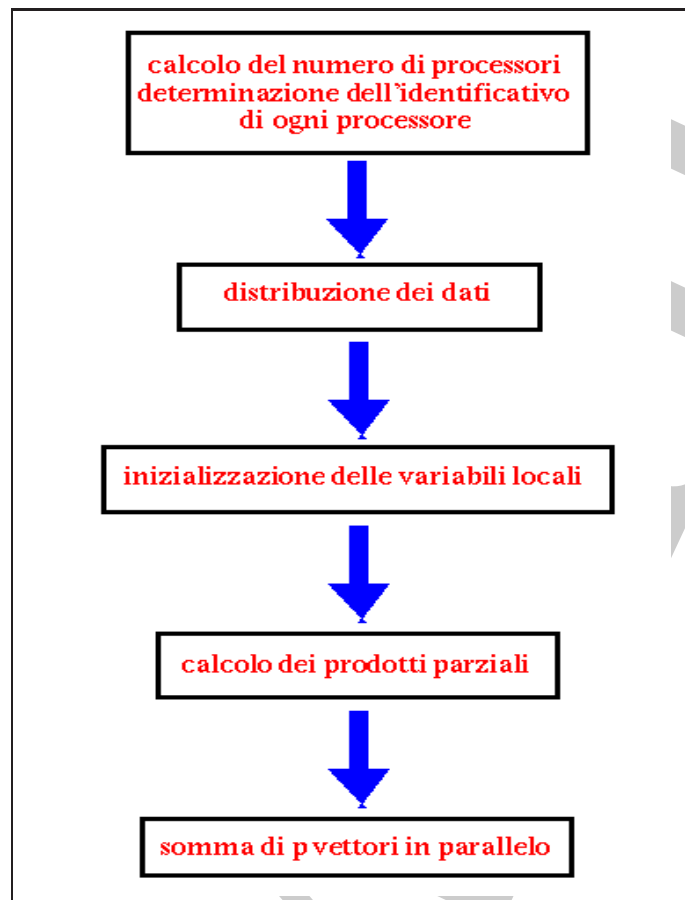


Figura 3.14: Schema generale delle fasi fondamentali per il calcolo del prodotto matrice per vettore.

In generale, al passo i -esimo comunicano i processori il cui identificativo differisce solo nel bit $(m - i + 1)$ -mo.

In definitiva, l'algoritmo per il prodotto matrice-vettore su un calcolatore MIMD a memoria distribuita con $p = 2^m$ processori ed $n = kp$ è mostrato nelle *Procedure* 3.16, 3.17, 3.18, 3.19 . In particolare, in Fig. 3.14 è mostrato inizialmente uno schema grafico delle macro operazioni (flow chart).

Algoritmo MATVET , I STRATEGIA $p = 2^m, n = kp$

```
begin procedure matvet(n,p,A,x,y)
  % calcolo del numero di processori p
  % e del proprio identificativo myid
  loc.init(p, myid)
  % k=numero di righe distribuite a ciascun processore
  k := n/p
  % k1=indice della prima riga distribuita
  % al processore di id. myid
  k1 := k · myid
  % k2=indice dell'ultima riga distribuita
  % al processore di id. myid
  k2 := k1 + k - 1
  % distribuzione di dati
  % inizializzazione delle variabili locali
  Aloc := A(k1 : k2, :)
  xloc := x(0 : n - 1)
  yloc := y(0 : n - 1)
  % calcolo dei prodotti parziali
  for i=0,k-1 do
    yloc(i) := 0
    for j=0,n-1 do
      yloc(i) := yloc(i) + Aloc(i, j)xloc(j)
    endfor
  endfor
  % chiamata alla procedura che effettua il
  % collezionamento dei dati
  call collezione (n,p,myid,y)
end
```

*Procedura 3.16 - Procedura per il prodotto matrice \times vettore su p processori utilizzando la prima strategia (distribuzione della matrice per blocchi di righe). Viene prima determinato il numero di righe k da distribuire a ciascun processore. Vengono quindi determinati gli indici della prima e dell'ultima riga da distribuire a ciascun processore e vengono distribuiti i dati. Ciascun processore calcola le proprie componenti del vettore y e viene chiamata la procedura **collezione** che colleziona i dati distribuiti sui processori.*

Algoritmo per collezionare i dati distribuiti su p processori

```

begin collezione (n,p,myid,y)
  for i=1,log2p do
    % idcom=identificativo del processore con cui comunicare
    % resto=i-mo bit di myid
    % numel= numero di elementi che devono essere scambiati.
    resto := myid -  $\lfloor \frac{myid}{2^i} \rfloor \times 2^i$ 
    numel=2i * n/p
    % Pmyid comunica con Pmyid+2i-1
    if (resto < 2i-1) then
      idcom := myid + 2i-1
      l1= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p}$ , l2= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} + numel - 1$ 
      l3= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} + numel$ , l4= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} + 2 * numel - 1$ 
    % oppure con Pmyid-2i-1
    else
      idcom := myid - 2i-1
      l1= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p}$ , l2= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} + numel - 1$ 
      l3= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} - numel$ , l4= $\lfloor \frac{myid}{2^{i-1}} \rfloor * 2^{i-1} * \frac{n}{p} - 1$ 
    endif
    % scambio dei dati
    send(y(l1:l2,idcom))
    rcv(y(l3:l4),idcom)
  endfor
end

```

Procedura 3.17 - Procedura per il collezionamento dei dati distribuiti su p processori. Ad ogni passo viene determinato l'identificativo del processore con cui comunicare e il numero di elementi che devono essere scambiati. Vengono inoltre determinati gli indici l1 ed l2 della prima e dell'ultima componente da inviare e gli indici l3 ed l4 della prima e dell'ultima componente che devono essere ricevute. Infine vengono effettuate le comunicazioni.

Algoritmo MATVET , II STRATEGIA $p = 2^m, n = kp$

```
begin procedure matvet(n,p,A,x,y)
  % calcolo del numero di processori p
  % e del proprio identificativo myid
  % k=numero di colonne distribuite a ciascun processore
   $k := n/p$ 
  % k1=indice della prima colonna distribuita
  % al processore di id. myid
   $k1 := k \cdot myid$ 
  % k2=indice dell'ultima colonna distribuita
  % al processore di id. myid
   $k2 := k1 + k - 1$ 
  % distribuzione di dati
  % inizializzazione delle variabili locali
   $A_{loc} := A(:, k1 : k2)$ 
   $x_{loc} := x(k1 : k2)$ 
   $y_{loc} := y(0 : n - 1)$ 
  % calcolo dei prodotti parziali
  for i=0,n-1 do
     $y_{loc}(i) := 0$ 
    for j=0,k-1 do
       $y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$ 
    endfor
  endfor
  % chiamata alla procedura che effettua la
  % somma di p vettori in parallelo
  call somma (n,p,myid,y)
end
```

Procedura 3.18 - Procedura per il prodotto matrice \times vettore su p processori utilizzando la seconda strategia (distribuzione della matrice per blocchi di colonne). Viene prima determinato in numero di colonne k da distribuire a ciascun processore. Vengono quindi determinati gli indici della prima e dell'ultima colonna da distribuire a ciascun processo e vengono distribuiti i dati. Ciascun processore calcola i prodotti parziali delle componenti del vettore y e viene chiamata la procedura somma che colleziona i dati distribuiti sui processori.

Algoritmo somma di p vettori

```
begin somma (n,p,myid,y)
  for i=1,log2p do
    % idcom=identificativo del processore con cui
    % effettuare la somma
    % resto=i-mo bit di myid
    resto := myid -  $\lfloor \frac{myid}{2^i} \rfloor \times 2^i$ 
    %  $P_{myid}$  comunica con  $P_{myid+2^{i-1}}$ 
    if (resto <  $2^{i-1}$ ) then
      idcom := myid +  $2^{i-1}$ 
      l1 := 0; l2 := n/2 - 1; l3 := n/2; l4 := n - 1
    % oppure con  $P_{myid-2^{i-1}}$ 
    else
      idcom := myid -  $2^{i-1}$ 
      l1 := n/2; l2 := n - 1; l3 := 0; l4 := n/2 - 1
    endif
    % procedura per la somma di 2 vettori
    % distribuiti tra 2 processori
    procedure SUM2(idcom,y,l1,l2,l3,l4)
  endfor
end
```

Procedura 3.19 - Procedura per la somma di p vettori distribuiti su p processori. Ad ogni passo viene determinato l'identificativo del processore con cui comunicare. Vengono inoltre determinati gli indici $l1$ ed $l2$ che individuano la parte del vettore da inviare e gli indici $l3$ ed $l4$ che individuano la parte del vettore che deve essere ricevuto. Infine vengono effettuate le comunicazioni.

Analisi delle prestazioni dell'algoritmo

Sia t_{calc} il tempo impiegato per l'esecuzione di una operazione floating point (1 flops) e t_{com} il tempo necessario alla comunicazione di un dato floating point. Si osserva che nella II strategia, per il calcolo dei prodotti parziali ogni processore effettua il calcolo di y_{loc}

nel modo seguente:

$$y_{loc}(i) := y_{loc}(i) + A_{loc}(i, j)x_{loc}(j)$$

per $i = 0, \dots, n - 1$ e $j = 0, \dots, k - 1$, che richiede

$$T_{calc} = 2 \cdot n \cdot k \cdot t_{calc}$$

avendo indicato con T_{calc} il tempo totale di calcolo; n è l'ordine della matrice A e la dimensione di x ed y , mentre k è il numero di righe o colonne distribuite a ciascun processore.

Tenuto conto che $k = n/p$, si ha:

$$T_{calc} = 2 \cdot \frac{n^2}{p} t_{calc}$$

Inoltre, ad ogni passo i processori scambiano tra loro $n/2$ componenti di y_{loc} . Tenuto conto che il numero di passi è $\log_2 p = m$, vengono effettuate

$$T_{com} = m \cdot \frac{n}{2} \text{ comunicazioni}$$

avendo indicato con T_{com} il tempo totale di comunicazione.

In definitiva si ha:

$$T_{calc} = 2 \cdot \frac{n^2}{p} t_{calc}$$

$$T_{com} = m \cdot \frac{n}{2} t_{com}$$

3.3 Lo standard MPI

Il criterio per progettare algoritmi paralleli su calcolatori MIMD è quello di pensare ad un insieme di processi concorrenti che coordinano la propria attività attraverso la comunicazione di messaggi. Tale modello di programmazione è noto con il nome di MESSAGE PASSING.

PVM e MPI sono due delle più note librerie di software alla base del paradigma del message passing: sono cioè un insieme di funzioni che consentono ai processori di scambiarsi informazioni¹¹.

PVM ([25]) è l'acronimo di Parallel Virtual Machine e consente di vedere una rete di calcolatori UNIX come un singolo calcolatore parallelo a memoria distribuita (la macchina virtuale). Lo sviluppo di PVM è iniziato nel 1989 all'Oak Ridge National Laboratory (ORNL).

MPI è l'acronimo di “*Message Passing Interface*” [14]. MPI nasce dalla necessità di definire uno standard unico per le librerie di message passing per gli Stati Uniti e l'Europa. In Europa dal 1991 il *Gruppo di Interesse Speciale sulla Standardizzazione dei Linguaggi per Macchine MIMD a Memoria Distribuita dell'ESPRIT* si era preoccupato di disegnare uno standard per il message passing. Questo intento portò al progetto SHAPES, che tentava di unificare i modelli di programmazione di UBIK ASI e di PARMACS^a. Purtroppo i due approcci erano troppo differenti per essere unificati. PARMACS fu una delle basi per MPI.

Negli Stati Uniti già dai primi anni '90 erano presenti sul mercato vari calcolatori a memoria distribuita e alcuni gruppi di ricerca avevano sviluppato sistemi di message passing che cercavano di fornire portabilità attraverso piattaforme differenti. PVM era uno di questi.

Jack Dongarra e Tony Hey furono tra i primi a riconoscere la necessità di coordinare i tentativi di standardizzazione europei e di oltre oceano al fine di evitare la scelta di standard differenti. Nell'estate del 1992 Jack Dongarra, Rolf Hempel, Tony Hey e David Walker decisero di definire uno standard per il message passing.

^aPARMACS sta per PARallel MACroS

¹¹ Gli indirizzi relativi alle librerie PVM ed MPI sono rispettivamente:
http://www.csm.ornl.gov/pvm/pvm_home.html
<http://www-unix.mcs.anl.gov/mmpi>

Questo processo ebbe inizio con il *Workshop su Standard per il Message Passing in Ambiente a Memoria Distribuita* che ebbe luogo il 29-30 Aprile del 1992 a Williamsburg, Virginia (USA). Nel Workshop, sponsorizzato dal CRPC (Center for Research on Parallel Computing) della Rice University, furono discusse differenti interfacce per il message passing ed esaminate le principali caratteristiche che avrebbe dovuto avere uno standard per il message passing. Su proposta di Ken Kennedy della Rice University fu istituito un gruppo di lavoro per lo sviluppo dello standard.

Nell'Ottobre del 1992 fu presentata una prima versione dello standard: MPI-0. MPI-0 comprendeva le principali caratteristiche individuate durante il Workshop, ma mancavano le routine per le comunicazioni collettive. Fu deciso allora di dare al processo di standardizzazione un'impronta più formale. Fu quindi istituito l'MPI Forum e furono create sottocommissioni per i settori più importanti dello standard.

Gli obiettivi originali dell'MPI Forum erano di sviluppare uno standard per il message passing che fosse largamente utilizzato, pratico, portabile, efficiente ed estensibile. Più precisamente disegnare un'interfaccia che:

- potesse essere utilizzata nei progetti di sviluppo software a vari livelli, dalle applicazioni per gli utenti alle librerie parallele ottimizzate;
- fosse non troppo differente da quelle già esistenti come PARMACS o PVM;
- potesse essere implementata su differenti piattaforme, senza sostanziali modifiche nelle comunicazioni e nel software di sistema;
- consenta l'implementazione su sistemi eterogenei;
- realizzi chiamate al C e al Fortran rendendo contemporaneamente la semantica dell'interfaccia indipendente dal linguaggio;

Per concludere questo ambizioso progetto in poco tempo, l'MPI Forum si è incontrato per i primi nove mesi del 1993 ogni sei settimane a Dallas nel Texas. Alla conferenza Supercomputing del 1993 a Portland, nell'Oregon, fu presentato un draft della nuova versione MPI-1. È seguito un periodo di consultazione, fino al Febbraio del 1994, durante il quale le persone erano invitate a commentare le specifiche. A Gennaio del 1994 fu organizzata una riunione dell'MPI Forum all'INRIA Sophia Antipolis in Francia, per consentire anche alla comunità dell'High Performance Computing Europea di contribuire al nuovo standard. Infine, il 5 Maggio 1994, la versione 1.0 di MPI fu distribuita su Internet. Negli anni successivi sono state rilasciate nuove versioni di MPI.

PVM è stato originalmente sviluppato per essere utilizzato su reti di workstation e per rispondere ai problemi di eterogeneità, di fault tolerance^a, di interoperabilità.

^aCapacità di affrontare situazioni di non disponibilità di una risorsa hw e/o software.

La costruzione di MPI invece si è basata sulle capacità di message passing e sulla possibilità di ottenere alte prestazioni su architetture parallele omogenee.

Al contrario di MPI, PVM è stato un progetto di ricerca con una vita finita. Il suo sviluppo e supporto sono stati resi possibili da un piccolo gruppo di ricercatori, e pur essendo stato un progetto di grande successo, non avrebbe potuto raggiungere la stessa funzionalità e sofisticatezza richiesta dall'MPI forum per uno standard del message passing. È da notare che il processo di consultazione che ha portato allo sviluppo di MPI è stato importante non solo per aver generato delle ottime specifiche, ma anche per aver dato ad MPI una credibilità che altre librerie per il message passing, sviluppate da gruppi di ricerca o da venditori, non hanno.

Lo standard MPI comprende i seguenti argomenti:

- comunicazioni punto-punto;
- operazioni collettive;
- gruppi di processori;
- domini di comunicazione;
- topologie di processori;
- gestione dell'ambiente;
- interfaccia per il profiling;
- chiamate al Fortran e al C.

Lo scambio di informazioni tra due o più processori avviene attraverso la comunicazione di messaggi:

- comunicazioni “*uno a uno*”;
- comunicazioni “*collettive*”.

Le comunicazioni “*uno a uno*” coinvolgono solo due processori. Le comunicazioni “*collettive*”, invece, coinvolgono più processori contemporaneamente. Sono possibili tipi differenti di comunicazioni collettive:

- comunicazione di uno a più di uno (broadcast);
- scambio di informazioni tra più processori (operazioni di riduzione, come la ricerca del massimo di una lista distribuita, somma degli elementi di un vettore distribuito, ecc.).

MPI fornisce le routine che implementano nella maniera più efficiente le varie modalità di comunicazioni esaminate.

Nel disegno di un algoritmo basato su MPI la prima cosa da fare è richiamare il file contenente le definizioni necessarie al preprocessore per l'utilizzo di MPI. Questa operazione si realizza mediante la direttiva (MACRO):

```
#include "/usr/mpich/include/mpi.h"
```

Le altre operazioni di base sono le seguenti:

1. *Inizializzazione di MPI*

```
MPI_Init()
```

Questa routine deve essere chiamata prima di ogni altra routine MPI. Definisce l'insieme dei processori attivati (contesto).

2. *Identificativo dei processori*

```
MPI_Comm_rank(MPI_Comm comm,int *menum)
```

Assegna ad ogni processore del **communicator** comm l'identificativo menum. In MPI si distingue tra **contesto** e **communicator**. Il primo è un identificativo associato ad un gruppo di processori. Il secondo è un ulteriore identificativo, associato ad un contesto e racchiude tutte le informazioni dell'ambiente di comunicazione, come la topologia, quali e quanti processori sono coinvolti, etc.

MPI_COMMON_WORLD indica il *communicator* a cui appartengono tutti i processori attivati e non può essere alterato dopo l'inizializzazione.

3. *Numero di processori del communicator*

```
MPI_Comm_size(MPI_COMM comm,int *nproc)
```

Restituisce ad ogni processore del communicator comm il numero totale di processori del contesto. Permette di conoscere quati processori stanno concorrendo per una determinata operazione.

4. *Termine di un programma MPI*

```
MPI_Finalize()
```

Determina la fine del programma MPI. Dopo questa routine non è possibile richiamare nessuna altra routine MPI.

5. *Spedizione di messaggi*

```
MPI_Send(void *buf,int count,  
          MPI_Datatype datatype,int dest,  
          int tag,MPI_Comm comm)
```

Spedisce i primi count elementi di buf, del tipo datatype, al processore dest. L'identificativo tag individua univocamente il messaggio nel contesto comm.

6. *Ricezione di un messaggio*

```
MPI_Recv(void *buf,int count,  
          MPI_Datatype datatype,int source,  
          int tag,MPI_Comm comm,  
          MPI_Status *status)
```

Riceve i primi count elementi di buf, del tipo datatype, dal processore source. L'identificativo tag individua univocamente il messaggio nel contesto comm. status è l'indicatore di errore.

A scopo illustrativo riportiamo di seguito un programma scritto in linguaggio C che utilizza le routine di MPI per eseguire il prodotto matrice vettore secondo la I Strategia.

```
1  #include <stdio.h>
2  #include "mpi.h"
3  main(int argc, char *argv){
4      int menum, nproc;
5      int n, nloc, tag, i, j, k, rest;
6      int passi, proc, step, tmp;
7      float *x, **a, **aloc, *y, *yloc, sum;
8      MPI_Status status;
9
10     /* Inizializzazione dell'ambiente MPI */
11     MPI_Init(&argc, &argv);
12     /* Calcolo del numero di processori */
13     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
14     /* Calcolo dell'identificativo */
15     MPI_Comm_rank(MPI_COMM_WORLD, &menum);
16     tag=1;
17     /* Il processore 0 legge la matrice a */
18     if (menum == 0 ){
19         printf("Inserire n \n");
20         scanf("%d", &n);
21         y = (float*)calloc(n, sizeof(float));
22         a = (float**)calloc(n, sizeof(float *));
23         for (i=0; i<n; i++)
24             *(a+i) = (float*)calloc(n, sizeof(float));
25         printf("Inserire a \n");
26         for (i=0; i<n; i++)
27             for (j=0; j<n; j++)
28                 scanf("%f", *(a+i)+j);
29     }
30     /* Il processore 0 esegue un Broadcast del dato n */
31     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
32
33     /* Calcolo della dimensione locale */
34     nloc = n / nproc;
35     /* Calcolo del resto */
36     rest = n % nproc;
```

Procedura 3.20 - Procedura che esegue il prodotto matrice per vettore secondo la I strategia - continua

```

37  /* I primi rest processori si distribuiscono rest */
38  if (rest != 0 && menum < rest)
39      nloc++;
40  aloc = (float**)calloc(nloc,sizeof(float *));
41  for (i=0; i<nloc; i++)
42      *(aloc+i) = (float*)calloc(n,sizeof(float));
43  x = (float*)calloc(n,sizeof(float));
44  yloc = (float*)calloc(nloc,sizeof(float));
45  /* Il processore 0 legge il vettore x */
46  if (menum == 0){
47      printf("Inserire x \n");
48      for (i=0; i<n; i++)
49          scanf("%f",x+i);
50  }
51  /* Il processore 0 esegue un Broadcast del vettore x */
52  MPI_Bcast(x,n,MPI_FLOAT,0,MPI_COMM_WORLD);
53
54  /* Distribuzione da parte di 0 dei blocchi di righe della matrice a */
55  if (menum == 0){
56      aloc = a;
57      step = nloc;
58      for (i=1; i<nproc; i++){
59          if ( rest != 0 )
60              if (i < rest)
61                  tmp = nloc;
62              else
63                  tmp = nloc -1;
64          else
65              tmp=nloc;
66          for (j=0; j<tmp; j++){
67              tag=22+i+j;
68              MPI_Send(*(a+step+j),n,MPI_FLOAT,i,tag,MPI_COMM_WORLD); }
69          step += tmp;
70      }
71  }
72  else{
73      /* Ricezione dei blocchi di righe della matrice a */
74      for (i=0; i<nloc; i++){
75          tag = 22+menum+i;
76          MPI_Recv(*(aloc+i),n,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
77      }
78  }
79  /* Calcolo dei prodotti locali */
80
81  for (i=0; i<nloc; i++){
82      *(yloc+i) = 0;
83  }

```

Procedura 3.20 - Procedura che esegue il prodotto matrice per vettore secondo la I strategia - continua

```

84     for (j=0; j<n; j++){
85         *(yloc+i) += (*aloc+i)+j * *(x+j);
86     }
87 }
88
89  /* Il processore P0 colleziona opportunamente i dati che riceve in y */
90  if (menum == 0){
91      for (k=0; k<nloc; k++){
92          *(y+k)=*(yloc+k);
93      }
94      step =nloc;
95      for (i=1; i<nproc; i++){
96          if ( rest != 0 )
97              if ( i < rest)
98                  tmp = nloc;
99              else
100                  tmp = nloc -1;
101          else
102              tmp=nloc;
103          tag=i;
104
105          MPI_Recv(y+step,tmp,MPI_FLOAT,i,tag,MPI_COMM_WORLD,&status);
106          step += tmp;
107      }
108  }
109  else{
110      /* tutti i processori spediscono a 0 il loro prodotto */
111      tag=menum;
112      MPI_Send(yloc,nloc,MPI_FLOAT,0,tag,MPI_COMM_WORLD); }
113
114  /* Il processore P0 stampa il risultato finale */
115  if (menum == 0)
116      for (i=0; i<n; i++){
117          printf("prodotto %f\n",*(y+i) );}
118
119  /* Chiusura dell'ambiente MPI */
120  MPI_Finalize();
121  return 0;
122 }

```

Procedura 3.20 - Procedura che esegue il prodotto matrice per vettore secondo la I strategia - fine

3.4 CASE STUDY: l'operazione del prodotto di due matrici

Problema

Calcolo del prodotto

$$C = A \cdot B$$

con $A \in \mathbb{R}^{m \times h}$, $B \in \mathbb{R}^{h \times n}$ e $C \in \mathbb{R}^{m \times n}$, su un calcolatore tipo MIMD a memoria distribuita con p processori.

Un algoritmo sequenziale è, ad esempio, quello che calcola la matrice prodotto componente per componente, effettuando i prodotti scalari di ciascuna riga di A per ciascuna colonna di B , una componente per volta secondo un ordine prestabilito¹²:

prodotto matrice-matrice

```
program matmat(a,b,c,m,h,n)
integer n, m, h, i, j, k
real a(m, h), b(h, n), c(n, m)
for i = 0 to m - 1 do
  for j = 0 to n - 1 do
    cij := 0
    for k = 0 to h - 1 do
      cij := cij + aik · bkj
    endfor
  endfor
endfor
```

Procedura 3.21 - Algoritmo sequenziale per il calcolo del prodotto matrice per matrice.

¹²È noto che l'efficienza del software che implementa l'algoritmo dipende fortemente dall'ordine degli indici, ijk . La scelta ottimale dipende dall'ambiente di calcolo (linguaggio, organizzazione della memoria, etc...). In questo momento si è scelto di prendere in esame la versione più naturale ijk , senza badare all'efficienza computazionale.

Gli elementi di C sono ottenuti mediante prodotto scalare delle righe di A e le colonne di B .

Le strategie per sviluppare un algoritmo parallelo per il prodotto di due matrici sono molteplici. Per descriverle meglio consideriamo il caso in cui $m = h = n = 6$, ovvero le 3 matrici sono quadrate di dimensione 6.

$$\begin{aligned}
 & \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & c_{05} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{30} & c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\ c_{50} & c_{51} & c_{52} & c_{53} & c_{54} & c_{55} \end{bmatrix} = \\
 & = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{bmatrix}
 \end{aligned}$$

I Strategia

Poiché i prodotti scalari possono essere effettuati in modo indipendente l'uno dall'altro, è naturale introdurre una prima strategia di parallelismo nel calcolo di tali prodotti. In altre parole, poiché le componenti di C sono calcolate effettuando il prodotto scalare tra le righe di A per le colonne di B , si può pensare di suddividere A in blocchi di righe e B in blocchi di colonne. Consideriamo ad esempio la decomposizione delle matrici A e B seguenti.

$$\begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \cdot \begin{bmatrix} B_0 & B_1 \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

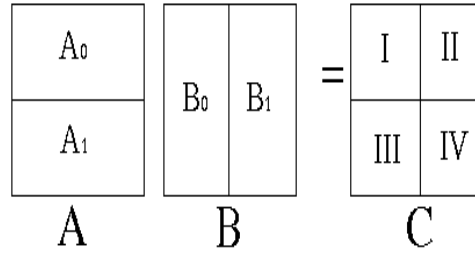


Figura 3.15: *I strategia: la matrice A viene suddivisa in due blocchi di righe, mentre la matrice B viene suddivisa in due blocchi di colonne. Con questo tipo di suddivisione dei dati ciascun processore potrà calcolare due blocchi della matrice C .*

con A_0 e $A_1 \in \mathbb{R}^{\frac{m}{2} \times h}$, B_0 e $B_1 \in \mathbb{R}^{h \times \frac{n}{2}}$, e le matrici $C_{ij} \in \mathbb{R}^{\frac{m}{2} \times \frac{n}{2}}$, per $i, j = 0, 1$.

Ad ogni processore viene assegnato un blocco di A e un blocco di B .

processore P_0	processore P_1
A_0, B_0	A_1, B_1

Nell'esempio considerato il processore P_0 possiede gli elementi di A_0 :

$$A_0 \equiv (a_{i,j}) \quad i = 0, \dots, 2; j = 0, \dots, 5$$

e quelli di B_0 :

$$B_0 \equiv (b_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

Il processore P_1 possiede gli elementi di A_1 :

$$A_1 \equiv (a_{i,j}) \quad i = 3, \dots, 5; j = 0, \dots, 5$$

e quelli di B_1 :

$$B_1 \equiv (b_{i,j}) \quad i = 0, \dots, 5; j = 3, \dots, 5$$

Con i dati così distribuiti ciascun processore può calcolare una parte degli elementi di C . In particolare:

- P_0 calcola $C_{0,0} = [c_{ij}]_{i,j=0,1,2}$ dove $c_{ij} = \sum_{k=0,\dots,5} a_{ik} b_{kj}$ per $i, j = 0, 1, 2$ cioè il blocco I di C ;
- P_1 calcola $C_{1,1} = [c_{ij}]_{i,j=3,4,5}$ dove $c_{ij} = \sum_{k=0,\dots,5} a_{ik} b_{kj}$ per $i, j = 3, 4, 5$ cioè il blocco IV di C .

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : for i=0 to 2 do for j=0 to 2 do for k = 0 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>	<pre> : for i=3 to 5 do for j=3 to 5 do for k = 0 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>

Procedura 3.22 - Algoritmi per il calcolo delle matrici C_{00} e C_{11}

A questo punto, se i due processori si scambiano i blocchi di B , possono calcolare i blocchi restanti di C , cioè P_0 calcolerà C_{01} e P_1 calcolerà C_{10} ¹³. In memoria, quindi ciascun processore ha i blocchi di A e di B seguenti:

processore P_0	processore P_1
A_0, B_1	A_1, B_0

Cioè, se il processore P_0 possiede gli elementi di B_1 :

$$B_1 \equiv (b_{i,j}) \quad i = 0, \dots, 5; j = 3, \dots, 5$$

¹³In alternativa i processori possono anche scambiarsi i blocchi di A , in questo modo il processore P_0 calcola C_{10} ed il processore P_1 calcola C_{01} .

e il processore P_1 possiede gli elementi di B_0 :

$$B_0 \equiv (b_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

possono entrambi effettuare i calcoli seguenti:

- P_0 calcola $C_{0,1} = [c_{ij}]_{i=0,1,2 \ j=3,4,5}$ dove $c_{ij} = \sum_{k=0,\dots,5} a_{ik} b_{kj}$ per $i = 0, 1, 2$ e $j = 3, 4, 5$ cioè il blocco II di C ;
- P_1 calcola $C_{1,0} = [c_{ij}]_{i=3,4,5 \ j=0,1,2}$ dove $c_{ij} = \sum_{k=0,\dots,5} a_{ik} b_{kj}$ per $i = 3, 4, 5$ e $j = 0, 1, 2$ cioè il blocco III di C .

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : for i=0 to 2 do for j=3 to 5 do for k = 0 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>	<pre> : for i=3 to 5 do for j=0 to 2 do for k = 0 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>

Procedura 3.23 - Algoritmi per il calcolo delle matrici C_{10} e C_{01}

Da notare che in questa prima strategia vengono “spezzati” entrambi i cicli sugli indici i e j .

II Strategia

Si può anche pensare di suddividere A in blocchi di colonne e B in blocchi di righe. Avremo quindi la decomposizione delle matrici seguente:

$$\begin{bmatrix} A_0 & A_1 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} = C$$

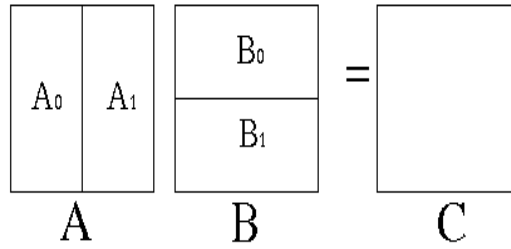


Figura 3.16: *Il strategia: la matrice A viene suddivisa in due blocchi di colonne, mentre la matrice B viene suddivisa in due blocchi di righe. Con questo tipo di suddivisione dei dati ciascun processore potrà calcolare una somma parziale di tutti gli elementi della matrice C.*

con A_0 e $A_1 \in \mathbb{R}^{m \times \frac{h}{2}}$, B_0 e $B_1 \in \mathbb{R}^{\frac{h}{2} \times n}$, e $C \in \mathbb{R}^{m \times n}$.

Ad ogni processore viene assegnato un blocco di A e un blocco di B. In particolare:

processore P_0	processore P_1
A_0, B_0	A_1, B_1

Nell'esempio considerato il processore P_0 possiede gli elementi di A_0 :

$$A_0 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

e quelli di B_0 :

$$B_0 \equiv (b_{i,j}) \quad i = 0, \dots, 2; j = 0, \dots, 5$$

Il processore P_1 possiede gli elementi di A_1 :

$$A_1 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 3, \dots, 5$$

e quelli di B_1 :

$$B_1 \equiv (b_{i,j}) \quad i = 3, \dots, 5; j = 0, \dots, 5$$

Con i dati distribuiti così ogni processore calcola gli elementi seguenti:

- P_0 calcola $x_{ij} = \sum_{k=0,1,2} a_{ik} b_{kj}$ $i, j = 0, \dots, 5$
- P_1 calcola $y_{ij} = \sum_{k=3,4,5} a_{ik} b_{kj}$ $i, j = 0, \dots, 5$

dove $c_{ij} = x_{ij} + y_{ij}$.

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : for i = 0 to 5 do for j = 0 to 5 do for k = 0 to 2 do $x_{ij} := x_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>	<pre> : for i = 0 to 5 do for j = 0 to 5 do for k = 3 to 5 do $y_{ij} := y_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>

Procedura 3.24 - Algoritmi per il calcolo delle componenti parziali della matrice C .

In questa strategia abbiamo introdotto il parallelismo nel calcolo dei prodotti scalari che forniscono gli elementi della matrice C . Si può infatti notare che è stato decomposto il ciclo sull'indice k . Ogni processore contiene solo tre elementi di ogni riga della matrice A e tre elementi di ogni colonna della matrice B e non può quindi calcolare il prodotto righe-colonne completo. I processori devono allora scambiare tra loro i risultati parziali. Al secondo passo i due processori scambiano le somme parziali e le combinano:

- P_0 invia a P_1 x_{ij} e P_1 invia a P_0 y_{ij} con $i, j = 0, \dots, 5$;
- P_0 e P_1 calcolano $c_{ij} = x_{ij} + y_{ij}$ con $i, j = 0, \dots, 5$.

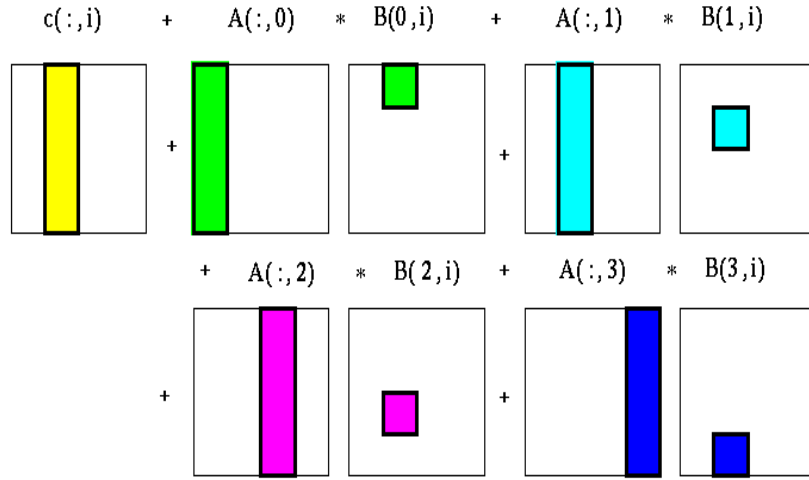


Figura 3.17: Il blocco i –esimo della matrice C è dato dalla somma dei prodotti tra i blocchi della matrice A e il blocco i –esimo della matrice B , partizionato in p sottoblocchi B_{ji} .

III Strategia

Si possono suddividere le matrici A , B e C in blocchi di colonne e assegnare al processore P_i , con $i = 0, \dots, p-1$, i blocchi A_i , B_i e C_i rispettivamente di dimensione $m \times \frac{h}{p}$, $h \times \frac{n}{p}$ e $m \times \frac{n}{p}$. Per determinare C_i , ogni processore deve partizionare il proprio blocco B_i in p blocchi riga ottenendo:

$$B_{ji} \in \mathbb{R}^{\frac{h}{p} \times \frac{n}{p}} \text{ con } j = 0, \dots, p-1$$

Con tale suddivisione si ha:

$$C_i = \sum_{k=0}^{p-1} A_k \cdot B_{ki} \text{ con } i = 0, \dots, p-1$$

In Fig. 3.17 è schematizzato l'algoritmo per il blocco $i = 1$.

Nel caso in cui $p = 2$, suddividiamo A e B in due blocchi di

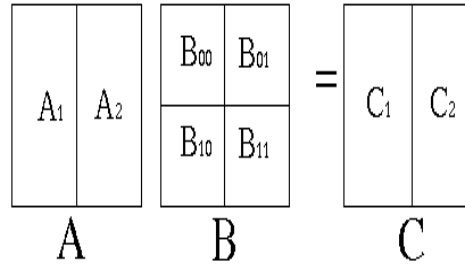


Figura 3.18: *III strategia: la matrice A viene suddivisa in due blocchi di colonne e la matrice B viene suddivisa in due blocchi di colonne. Ogni processore P_i , con $i = 0, p - 1$, partiziona il proprio blocco B_i in sottoblocchi B_{ji} , con $j = 0, p - 1$. Con questo tipo di suddivisione dei dati ciascun processore potrà calcolare un blocco di colonne della matrice C.*

colonne nel seguente modo:

$$[A_0, A_1] \cdot [B_0, B_1] = [C_0, C_1]$$

con A_0 e $A_1 \in \mathbb{R}^{m \times \frac{h}{2}}$, B_0 e $B_1 \in \mathbb{R}^{h \times \frac{n}{2}}$, e le matrici C_0 e $C_1 \in \mathbb{R}^{m \times \frac{n}{2}}$. Ad ogni processore viene assegnato un blocco di A e un blocco di B. In particolare:

processore P_0	processore P_1
A_0, B_0	A_1, B_1

Più precisamente, il processore P_0 possiede gli elementi di A_0 :

$$A_0 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

e quelli di B_0 :

$$B_0 \equiv (b_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

Il processore P_1 possiede gli elementi di A_1 :

$$A_1 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 3, \dots, 5$$

e quelli di B_1 :

$$B_1 \equiv b_{i,j} \quad i = 0, \dots, 5; j = 3, \dots, 5$$

Ogni processore P_i partiziona la matrice B_i in p blocchi riga, ottenendo:

$$B_i = \begin{bmatrix} B_{0i} \\ \vdots \\ B_{(p-1)i} \end{bmatrix}$$

Con i dati distribuiti e partizionati così ogni processore calcola gli elementi seguenti:

- P_0 calcola $c_{ij} = \sum_{k=0,1,2} a_{ik} b_{kj}$ per $i = 0, \dots, 5$ e $j = 0, 1, 2$ (P_0 avrà una somma parziale di tutti gli elementi delle prime tre colonne della matrice C ovvero calcola: $C_0 = A_0 \cdot B_{00}$);
- P_1 calcola $c_{ij} = \sum_{k=3,4,5} a_{ik} b_{kj}$ per $i = 0, \dots, 5$ e $j = 3, 4, 5$ (P_1 avrà una somma parziale di tutti gli elementi delle seconde tre colonne della matrice C ovvero calcola: $C_1 = A_1 \cdot B_{11}$).

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : for i = 0 to 5 do for j=0 to 2 do for k=0 to 2 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>	<pre> : for i = 0 to 5 do for j=3 to 5 do for k=3 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>

Procedura 3.25 - Algoritmi per il calcolo delle componenti parziali della matrice C .

Per poter calcolare il resto delle somme parziali i processori devono scambiare tra loro i blocchi di A :

processore P_0	processore P_1
A_1, B_0	A_0, B_1

Il processore P_0 possiede gli elementi di A_1 :

$$A_1 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 3, \dots, 5$$

Il processore P_1 possiede gli elementi di A_0 :

$$A_0 \equiv (a_{i,j}) \quad i = 0, \dots, 5; j = 0, \dots, 2$$

Al secondo passo i due processori calcolano i seguenti elementi:

- P_0 calcola $c_{ij} = \sum_{k=3,4,5} a_{ik} b_{kj}$ per $i = 0, \dots, 5$ e $j = 0, 1, 2$ (P_0 avrà le prime tre colonne della matrice C avendo calcolato: $C_0 = C_0 + A_1 \cdot B_{10} = A_0 \cdot B_{00} + A_1 \cdot B_{10}$);
- P_1 calcola $c_{ij} = \sum_{k=0,1,2} a_{ik} b_{kj}$ per $i = 0, \dots, 5$ e $j = 3, 4, 5$ (P_1 avrà le seconde tre colonne della matrice C avendo calcolato: $C_1 = C_1 + A_0 \cdot B_{01} = A_1 \cdot B_{11} + A_0 \cdot B_{01}$).

Algoritmo processore P_0	Algoritmo processore P_1
<pre> : for i = 0 to 5 do for j=0 to 2 do for k=3 to 5 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>	<pre> : for i = 0 to 5 do for j=3 to 5 do for k=0 to 2 do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$ endfor endfor endfor : </pre>

Procedura 3.26 - Algoritmi per il calcolo delle componenti risultanti della matrice C .

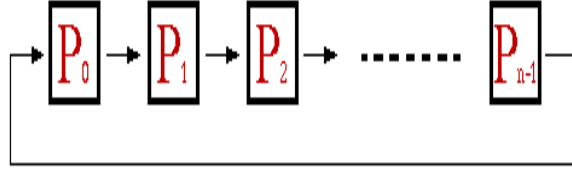


Figura 3.19: I processori sono collegati in modo tale che ognuno di essi può comunicare con il processore che si trova alla sua destra. L'ultimo processore comunica con il primo. Tale tipo di topologia è chiamata ad anello ed è stata illustrata nel terzo capitolo.

Dalla Fig. 3.17 risulta chiaro che per ottenere il risultato finale è necessario ad ogni passo scambiare i blocchi della matrice A . L'idea alla base di questa strategia è far calcolare ad ogni processore la quantità:

$$C(:, id) = C(:, id) + A(:, id) * B(id, id)$$

avendo indicato con id l'identificativo del processore, quindi ogni processore invia il proprio blocco di colonne di A al processore che si trova alla sua destra.

Tale strategia prende il nome di *1D-Systolic*¹⁴. Una strategia di comunicazione efficiente dei dati tra i processori è quella realizzata immaginando i processori collegati secondo una topologia ad anello (Fig. 3.19) in quanto ogni processore deve inviare al processore alla sua destra un blocco della matrice A . L'ultimo processore comunica con il primo. In questo caso si dice che l'anello è “*periodico*”.

L'algoritmo è il seguente:

¹⁴Tali algoritmi sono detti *sistolici* perché il flusso di dati, essendo sempre nella stessa direzione durante le fasi di comunicazione, ricorda il flusso di sangue dopo un movimento di contrazione del cuore, movimento che viene appunto detto “sistole”.

Algoritmo MATMAT 1D-Systolic (III strategia m=h=n)

```

begin
% calcolo del numero di processori  $p$  e del proprio identificativo  $myid$ 
%  $k$ =numero di colonne distribuito a ciascun processore
 $k := n/p$ 
%  $k1$ =indice della prima colonna distribuita ad ogni processore
 $k1 := k \cdot myid$ 
%  $k2$ =indice dell'ultima colonna distribuita ad ogni processore
 $k2 := k1 + k - 1$ 
% distribuzione dei dati
 $A_{loc} := A(:, k1 : k2)$ 
 $B_{loc} := B(:, k1 : k2)$ 
 $C_{loc} := C(:, k1 : k2)$ 
for  $i=0$  to  $k-1$  do
    for  $j=0$  to  $k-1$  do
         $C_{loc}(i, j) := 0$ 
    endfor
endfor
% fase di calcolo
 $l := k \cdot myid$ 
 $C_{loc} := C_{loc} + A_{loc}B_{loc}(l : l + k, :)$ 
for  $i=0$  to  $p-2$  do
%  $idsend$ : identificativo del processore a cui inviare
 $idsend := myid + 1$ 
if ( $idsend > (p - 1)$ ) then
     $idsend := 0$ 
endif
%  $idrecv$ : identificativo del processore da cui ricevere
 $idrecv = myid - 1$ 
if ( $idrecv < 0$ ) then
     $idrecv = p - 1$ 
endif
% fase di comunicazione
send( $A_{loc}, idsend$ )
recv( $A_{loc}, idrecv$ )
% fase di calcolo
 $l := mod(l - k, n)$ 
 $C_{loc} := C_{loc} + A_{loc}B_{loc}(l : l + k, :)$ 
endfor
end

```

*Procedura 3.27 - Algoritmo 1D-Systolic per il calcolo
del prodotto matrice per matrice.*

L'algoritmo 1D-Systolic è caratterizzato da una distribuzione dei dati e da uno schema di comunicazione monodimensionale.

IV Strategia

Supponiamo di avere $p = 9$ processori. Immaginiamo inoltre che i processori siano disposti secondo una topologia a griglia cartesiana (vedi paragrafo 2.2) e distribuiamo i blocchi nel modo seguente:

P_{00} $A_{00}B_{00}$	P_{01} $A_{01}B_{11}$	P_{02} $A_{02}B_{22}$
P_{10} $A_{11}B_{10}$	P_{11} $A_{12}B_{21}$	P_{12} $A_{10}B_{02}$
P_{20} $A_{22}B_{20}$	P_{21} $A_{20}B_{01}$	P_{22} $A_{21}B_{12}$

avendo indicato con $P_{i,j}$ il processore di riga i -esima e colonna j -esima. In questo caso si è suddiviso A e B in p blocchi di righe e in p blocchi di colonne, ottenendo così p^2 blocchi di A e di B :

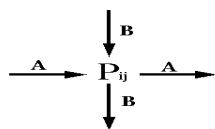
$$\begin{bmatrix} A_{00} & \dots & A_{0,p-1} \\ \vdots & & \vdots \\ A_{p-1,0} & \dots & A_{p-1,p-1} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & \dots & B_{0,p-1} \\ \vdots & & \vdots \\ B_{p-1,0} & \dots & B_{p-1,p-1} \end{bmatrix} = \begin{bmatrix} C_{00} & \dots & C_{0,p-1} \\ \vdots & & \vdots \\ C_{p-1,0} & \dots & C_{p-1,p-1} \end{bmatrix}$$

e il prodotto $A \cdot B = C$ viene decomposto in prodotti $A_{ik}B_{kj}$. Al processore P_{ij} vengono assegnati i blocchi A_{ik} e B_{kj} con:

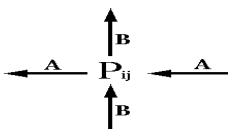
$$k = \langle i + j \rangle_p = \text{resto}(i + j, p)$$

Con questa distribuzione delle matrici, ogni processore può effettuare solo il calcolo di $C_{ij} = C_{ij} + A_{ik}B_{kj}$, cioè ha somme parziali del blocco C_{ij} di C . Per completare il calcolo ogni processore, quindi,

ha necessità di acquisire i blocchi di A e di B . Si può pensare di far comunicare ogni processore con i quattro processori ad esso vicini utilizzando una topologia di tipo toroidale (vedi paragrafo 2.3), secondo lo schema di comunicazione seguente:



O, equivalentemente, è possibile utilizzare sempre in una topologia di tipo toroidale quest'altro schema di comunicazione:



Scegliamo ad esempio quest'ultimo schema di comunicazione per illustrare la IV strategia. Ad ogni passo ogni processore su di una stessa riga invia al processore che si trova nella colonna alla sua sinistra nella griglia un blocco della matrice A e riceve dal processore che si trova nella colonna alla sua destra nella griglia un blocco della matrice A . Inoltre ogni processore su di una stessa colonna invia al processore che si trova sulla riga superiore della griglia un blocco della matrice B e riceve dal processore che si trova sulla riga inferiore della griglia un blocco della matrice B .

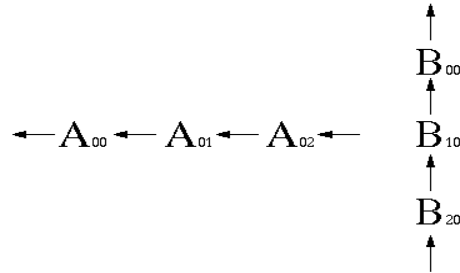
• I PASSO

Sulla riga i -esima della griglia di processori viene assegnato a P_{ij} , con $j = 0, 1, 2$, il blocco $A_{i, \text{mod}(i+j, 3)}$ della matrice A e il blocco $B_{\text{mod}(i+j, 3), j}$ della matrice B :

P_{00} $C_{00} = C_{00} + A_{00}B_{00}$	P_{01} $C_{01} = C_{01} + A_{01}B_{11}$	P_{02} $C_{02} = C_{02} + A_{02}B_{22}$
P_{10} $C_{10} = C_{10} + A_{11}B_{10}$	P_{11} $C_{11} = C_{11} + A_{12}B_{21}$	P_{12} $C_{12} = C_{12} + A_{10}B_{02}$
P_{20} $C_{20} = C_{20} + A_{22}B_{20}$	P_{21} $C_{21} = C_{21} + A_{20}B_{01}$	P_{22} $C_{22} = C_{22} + A_{21}B_{12}$

• II PASSO

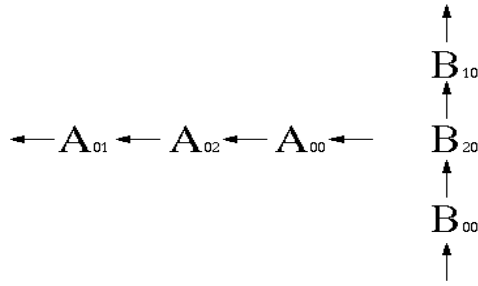
Sulla riga i -esima della griglia viene inviato ad ogni processore P_{ij} il blocco $A_{i, \text{mod}(i+j+\text{passo}-1, 3)}$ della matrice A dal processore $P_{i, \text{mod}(j+1, p)}$ posto nella colonna successiva della griglia cartesiana. Su ogni colonna di processori P_{ij} invia il suo blocco di B al processore $P_{\text{mod}(i-1, p), j}$, appartenente alla riga precedente della griglia. Ad esempio, se consideriamo la prima riga e la prima colonna, le comunicazioni possono essere schematizzate nel modo seguente:



P_{00} $C_{00} = C_{00} + A_{01}B_{10}$	P_{01} $C_{01} = C_{01} + A_{02}B_{21}$	P_{02} $C_{02} = C_{02} + A_{00}B_{02}$
P_{10} $C_{10} = C_{10} + A_{12}B_{20}$	P_{11} $C_{11} = C_{11} + A_{10}B_{01}$	P_{12} $C_{12} = C_{12} + A_{11}B_{12}$
P_{20} $C_{20} = C_{20} + A_{20}B_{00}$	P_{21} $C_{21} = C_{21} + A_{21}B_{11}$	P_{22} $C_{22} = C_{22} + A_{22}B_{22}$

• III PASSO

Sulla riga i -esima di processori viene inviato ad ogni processore P_{ij} il blocco $A_{i, \text{mod}(i+j+\text{passo}-1, 3)}$ della matrice A dal processore $P_{i, \text{mod}(j+1, p)}$ posto nella colonna successiva nella griglia cartesiana. Su ogni colonna di processori P_{ij} invia il suo blocco di B al processore $P_{\text{mod}(i-1, p), j}$ appartenente alla riga precedente della griglia. Ad esempio, se consideriamo la prima riga e la prima colonna, le comunicazioni sono schematizzate nel modo seguente:



P_{00}	P_{01}	P_{02}
$C_{00} = C_{00} + A_{02}B_{20}$	$C_{01} = C_{01} + A_{00}B_{01}$	$C_{02} = C_{02} + A_{01}B_{12}$
P_{10}	P_{11}	P_{12}
$C_{10} = C_{10} + A_{10}B_{00}$	$C_{11} = C_{11} + A_{11}B_{11}$	$C_{12} = C_{12} + A_{12}B_{22}$
P_{20}	P_{21}	P_{22}
$C_{20} = C_{20} + A_{21}B_{10}$	$C_{21} = C_{21} + A_{22}B_{21}$	$C_{22} = C_{22} + A_{20}B_{02}$

Tale strategia, essendo il flusso di dati lungo due direzioni ortogonali, e avendo lungo ciascuna direzione le stesse caratteristiche di quella *1D-Systolic*, prende il nome di *2D-Systolic*. Di seguito è riportato l'algoritmo. Sono stati denotati con north, south, west e east rispettivamente l'identificativo del processore a cui inviare i blocchi della matrice B , del processore da cui ricevere i blocchi della matrice B , del processore a cui inviare i blocchi della matrice A , e del processore da cui ricevere i blocchi della matrice A .

Algoritmo MATMAT 2D-Systolic (IV strategia m=h=n)

```

begin
  % calcolo del numero di processori  $p^2$  e del proprio identificativo
  %  $k$ =numero di righe e di colonne distribuite a ciascun processore
   $k := n/p$ 
  %  $k1x$ =prima riga di  $A$  distribuita a  $(idx, idy)$ 
   $k1x := idx \cdot k$ 
  %  $k1y$ =prima colonna di  $A$  e prima riga di  $B$  distribuite a  $(idx, idy)$ 
   $k1y := \text{mod}(idx + idy, p) \cdot k$ 
  %  $k2x$ =ultima riga di  $A$  distribuita a  $(idx, idy)$ 
   $k2x := k1x + k$ 
  %  $k2y$ =ultima colonna di  $A$  ed ultima riga di  $B$  distribuite a  $(idx, idy)$ 
   $k2y := k1y + k$ 
  %  $k1z$ =prima colonna di  $B$  distribuita a  $(idx, idy)$ 
   $k1z := idy \cdot k$ 
  %  $k2z$ =ultima colonna di  $B$  distribuita a  $(idx, idy)$ 
   $k2z := k1z + k$ 
  % distribuzione dei dati
   $A_{loc} := A(k1x : k2x, k1y : k2y)$ 
   $B_{loc} := B(k1y : k2y, k1z : k2z)$ 
   $C_{loc} := C(k1x : k2x, k1z : k2z)$ 
  % calcolo dell'identificativo dei processi con cui c'è comunicazione
  ovest= $(idx, \text{mod}(idy - 1, p))$ 
  east= $(idx, \text{mod}(idy + 1, p))$ 
  south= $(\text{mod}(idx + 1, p), idy)$ 
  north= $(\text{mod}(idx - 1, p), idy)$ 
  % fase di calcolo
   $C_{loc} = C_{loc} + A_{loc}B_{loc}$ 
  for i=0 to p-2 do
  % fase di comunicazione
    send( $A_{loc}$ , west) ; recv( $A_{loc}$ , east)
    send( $B_{loc}$ , north) ; recv( $B_{loc}$ , south)
  % fase di calcolo
     $C_{loc} = C_{loc} + A_{loc}B_{loc}$ 
  endfor
end

```

Procedura 3.28 - Algoritmo 2D-Systolic per il calcolo del prodotto matrice per matrice.

V Strategia

Come nella strategia precedente, si possono suddividere le matrici A e B in p^2 blocchi e assegnare al processore P_{ij} le sottomatrici A_{ij} e B_{ij} . Supponiamo di avere un numero di processori $p = 9$. Immaginiamo, inoltre, che i processori siano disposti secondo una griglia cartesiana e distribuiamo i blocchi nel modo seguente:

P_{00} $A_{00}B_{00}$	P_{01} $A_{01}B_{01}$	P_{02} $A_{02}B_{02}$
P_{10} $A_{10}B_{10}$	P_{11} $A_{11}B_{11}$	P_{12} $A_{12}B_{12}$
P_{20} $A_{20}B_{20}$	P_{21} $A_{21}B_{21}$	P_{22} $A_{22}B_{22}$

In questo caso si è suddiviso A e B in p blocchi di righe e in p blocchi di colonne, ottenendo così p^2 blocchi:

$$\begin{bmatrix} A_{00} & \dots & A_{0,p-1} \\ \vdots & & \vdots \\ A_{p-1,0} & \dots & A_{p-1,p-1} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & \dots & B_{0,p-1} \\ \vdots & & \vdots \\ B_{p-1,0} & \dots & B_{p-1,p-1} \end{bmatrix} = \begin{bmatrix} C_{00} & \dots & C_{0,p-1} \\ \vdots & & \vdots \\ C_{p-1,0} & \dots & C_{p-1,p-1} \end{bmatrix}$$

e il prodotto $A \cdot B = C$ viene decomposto in prodotti $A_{ik}B_{kj}$. Al processore P_{ij} vengono assegnati i blocchi A_{ij} e B_{ij} . Ciascun processore calcola un blocco della matrice C nel modo seguente:

$$\begin{aligned} P_{00} &\rightarrow C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} \\ P_{01} &\rightarrow C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} \\ P_{02} &\rightarrow C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} \\ P_{10} &\rightarrow C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} \\ P_{11} &\rightarrow C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} \\ P_{12} &\rightarrow C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22} \\ P_{20} &\rightarrow C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20} \end{aligned}$$

$$P_{21} \rightarrow C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$P_{22} \rightarrow C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Per poter completare il calcolo dei blocchi C_{ij} , ad ogni passo a partire dai processori diagonali, ogni processore invia il proprio blocco di A a tutti i processori della propria riga, ed un blocco di B al processore della riga precedente e della medesima colonna.

• I PASSO

Sulla riga i -esima della griglia di processori viene inviato a tutti i processori il blocco $A_{i,i}$ della matrice A . Ad esempio, sulla riga 0 viene inviato a tutti i processori il blocco A_{00} della matrice A , sulla riga 1 viene inviato a tutti i processori il blocco A_{11} della matrice A e sulla riga 2 viene inviato a tutti i processori il blocco A_{22} della matrice A . Tutti i processori utilizzano il blocco B_{ij} della matrice B a loro assegnato inizialmente. Nella figura sono evidenziati in rosso i prodotti che ciascun processore può calcolare.

P_{00}	P_{01}	P_{02}
$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$	$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$	$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$
P_{10}	P_{11}	P_{12}
$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$	$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$	$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$
P_{20}	P_{21}	P_{22}
$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$	$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$	$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$

• II PASSO

Sulla riga i -esima di processori viene inviato a tutti i processori il blocco $A_{i, \text{mod}(i+\text{passo}-1, 3)}$ della matrice A . Ad esempio, sulla riga 0 viene inviato a tutti i processori il blocco A_{01} della matrice A . Su ogni colonna di processori, ogni processore invia al processore della riga precedente il suo blocco di B e riceve

dal processore della riga successiva il blocco di B . Nella figura sono evidenziati in rosso i prodotti che ciascun processore può calcolare.

P_{00} $C_{00} = A_{00}B_{00} + \textcolor{red}{A_{01}B_{10}} + A_{02}B_{20}$	P_{01} $C_{01} = A_{00}B_{01} + \textcolor{red}{A_{01}B_{11}} + A_{02}B_{21}$	P_{02} $C_{02} = A_{00}B_{02} + \textcolor{red}{A_{01}B_{12}} + A_{02}B_{22}$
P_{10} $C_{10} = A_{10}B_{00} + A_{11}B_{10} + \textcolor{red}{A_{12}B_{20}}$	P_{11} $C_{11} = A_{10}B_{01} + A_{11}B_{11} + \textcolor{red}{A_{12}B_{21}}$	P_{12} $C_{12} = A_{10}B_{02} + A_{11}B_{12} + \textcolor{red}{A_{12}B_{22}}$
P_{20} $C_{20} = \textcolor{red}{A_{20}B_{00}} + A_{21}B_{10} + A_{22}B_{20}$	P_{21} $C_{21} = \textcolor{red}{A_{20}B_{01}} + A_{21}B_{11} + A_{22}B_{21}$	P_{22} $C_{12} = \textcolor{red}{A_{20}B_{02}} + A_{21}B_{12} + A_{22}B_{22}$

• III PASSO

Sulla riga i -esima di processori viene inviato a tutti i processori il blocco $A_{i, \text{mod}(i+\text{passo}-1, 3)}$ della matrice A . Ad esempio sulla riga 0 viene inviato a tutti i processori il blocco A_{02} della matrice A . Su ogni colonna di processori, ogni processore invia al processore della riga precedente il suo blocco di B e riceve dal processore della riga successiva il blocco di B . Nella figura sono evidenziati in rosso i prodotti che ciascun processore può calcolare.

P_{00} $C_{00} = A_{00}B_{00} + A_{01}B_{10} + \textcolor{red}{A_{02}B_{20}}$	P_{01} $C_{01} = A_{00}B_{01} + A_{01}B_{11} + \textcolor{red}{A_{02}B_{21}}$	P_{02} $C_{02} = A_{00}B_{02} + A_{01}B_{12} + \textcolor{red}{A_{02}B_{22}}$
P_{10} $C_{10} = \textcolor{red}{A_{10}B_{00}} + A_{11}B_{10} + A_{12}B_{20}$	P_{11} $C_{11} = \textcolor{red}{A_{10}B_{01}} + A_{11}B_{11} + A_{12}B_{21}$	P_{12} $C_{12} = \textcolor{red}{A_{10}B_{02}} + A_{11}B_{12} + A_{12}B_{22}$
P_{20} $C_{20} = A_{20}B_{00} + \textcolor{red}{A_{21}B_{10}} + A_{22}B_{20}$	P_{21} $C_{21} = A_{20}B_{01} + \textcolor{red}{A_{21}B_{11}} + A_{22}B_{21}$	P_{22} $C_{12} = A_{20}B_{02} + \textcolor{red}{A_{21}B_{12}} + A_{22}B_{22}$

Tale strategia prende il nome di *BMR* (Broadcast Multiply Rolling)¹⁵ e l'algoritmo è il seguente:

¹⁵Una strategia simile viene utilizzata anche in PBLAS, la libreria parallela di algebra lineare di cui si parlerà nel capitolo 6. La differenza risiede nel fatto che i blocchi della matrice B vengono spediti al processore della riga successiva della colonna medesima.

Algoritmo MATMAT BMR (V strategia m=h=n)

```
begin
% calcolo del numero di processori  $p^2$  e del proprio identificativo  $(idx, idy)$ 
% del proprio identificativo  $(idx, idy)$ 
%  $k$ =numero di righe e di colonne distribuite
% a ciascun processore
 $k := n/p$ 
%  $k1x$ =prima riga di A e B distribuite a  $(idx, idy)$ 
 $k1x := idx \cdot k$ 
%  $k2x$ =ultima riga di A e B distribuite a  $(idx, idy)$ 
 $k2x := k1x + k$ 
%  $k1y$ =prima colonna di A e B distribuite a  $(idx, idy)$ 
 $k1y := idy \cdot k$ 
%  $k2y$ =ultima colonna di A e B distribuite a  $(idx, idy)$ 
 $k2y := k1y + k$ 
% distribuzione dei dati
% inizializzazione delle variabili locali
 $A_{loc} := A(k1x : k2x, k1y : k2y)$ 
 $B_{loc} := B(k1x : k2x, k1y : k2y)$ 
 $C_{loc} := C(k1x : k2x, k1y : k2y)$ 
% calcolo dei processori con cui c'è comunicazione
north:=( $idx, mod(idy - 1, p)$ )
south:=( $idx, mod(idy + 1, p)$ )
for i=0 to p-1 do
% prima fase di comunicazione: broadcast sulle righe;
% il processore, dei sottocontesti riga della griglia,
% di identificativo proc comunica con tutti i processori
% che si trovano sulla sua stessa riga
 $proc := mod(idx + i, p)$ 
sendbroadcast( $A_{loc}, row, proc$ )
recvbroadcast( $A_{temp}, row, proc$ )
% fase di calcolo
 $C_{loc} := C_{loc} + A_{temp} B_{loc}$ 
% seconda fase di comunicazione
send( $B_{loc}, north$ )
recv( $B_{loc}, south$ )
endfor
end
```

Procedura 3.29 - Algoritmo per eseguire il prodotto matrice per matrice secondo lo schema BMR.

Stima teorica delle prestazioni

Valutiamo le prestazioni degli algoritmi 1D-Systolic, 2D-Systolic e BMR su un calcolatore MIMD a memoria distribuita costituito da $nproc = p \times p$ processori.

Siano

- $A, B, C \in \mathbb{R}^{n \times n}$;
- t_{com} il tempo per la comunicazione di un dato floating point tra due processori;
- t_{broad} il tempo per la comunicazione di un dato floating point in un broadcast orizzontale;
- t_{calc} il tempo per l'esecuzione di una operazione floating point;
- T_{calc} il tempo totale di esecuzione;
- T_{com} il tempo totale di comunicazione.

Nell'algoritmo 1D-Systolic, ogni processore P_i , $0 \leq i \leq p-1$, effettua il prodotto:

$$A_i \cdot B_i \quad (3.1)$$

con $A_i \in \mathbb{R}^{N \times \frac{N}{p}}$, $B_i \in \mathbb{R}^{\frac{N}{p} \times \frac{N}{p}}$.

L'operazione (3.1) richiede:

$$T_{calc} = 2 \cdot \frac{N^3}{p^2} t_{calc}$$

e viene eseguita p volte. In definitiva:

$$T_{calc} = p \cdot \frac{2N^3}{p^2} t_{calc} = \frac{2N^3}{p} t_{calc}$$

Inoltre, al passo j , con $j = 0, \dots, p-2$, ogni processore P_i deve inviare il proprio blocco di matrice A :

$$A_i \in \mathfrak{R}^{N \times \frac{N}{p}}$$

al processore $P_{mod(i+1,p)}$. Il tempo di comunicazione è quindi:

$$T_{com} = N \cdot \frac{N}{p} t_{com} = \frac{N^2}{p} t_{com}$$

Tale spedizione viene eseguita $p-1$ volte, e quindi:

$$T_{com} = \frac{N^2}{p} \cdot (p-1) t_{com} = \frac{N^2(p-1)}{p} t_{com}$$

Nell'algoritmo 2D-Systolic, ogni processore $P_{i,j}$, $0 \leq i \leq p-1$ e $0 \leq j \leq p-1$, effettua il prodotto:

$$A_{ik} \cdot B_{kj} \tag{3.2}$$

con A_{ik} e $B_{kj} \in \mathfrak{R}^{\frac{N}{p} \times \frac{N}{p}}$.

L'operazione (3.2) richiede:

$$T_{calc} = 2 \cdot \frac{N^3}{p^3} t_{calc}$$

e viene eseguita per p volte. In definitiva:

$$T_{calc} = p \cdot \frac{2N^3}{p^3} t_{calc} = \frac{2N^3}{p^2} t_{calc}$$

Inoltre, ad ogni passo h , con $h = 0, \dots, p-2$, ogni processore P_{ij} deve inviare il proprio blocco di matrice A :

$$A_{ik} \in \mathfrak{R}^{\frac{N}{p} \times \frac{N}{p}}$$

al processore $P_{i,mod(j-1,p)}$ e il proprio blocco di matrice B :

$$B_{kj} \in \Re^{\frac{N}{p} \times \frac{N}{p}}$$

al processore $P_{mod(i-1,p),j}$. Il tempo di comunicazione è quindi:

$$T_{com} = 2 \cdot \frac{N}{p} \cdot \frac{N}{p} t_{com} = \frac{2N^2}{p^2} t_{com}$$

Tale spedizione viene eseguita $p - 1$ volte. In definitiva:

$$T_{com} = \frac{2N^2}{p^2} \cdot (p - 1) t_{com} = \frac{2N^2(p - 1)}{p^2} t_{com}$$

Infine, nell'algoritmo BMR si ha una suddivisione dei blocchi di matrice simile all'algoritmo 2D-Systolic, così da avere lo stesso numero di operazioni fl. point:

$$T_{calc} = p \cdot \frac{2N^3}{p^3} t_{calc} = \frac{2N^3}{p^2} t_{calc}$$

Inoltre, ad ogni passo h , con $h = 0, \dots, p-1$ il processore $P_{mod(h+i,p)}$ dei sottocontesti riga della griglia¹⁶ deve inviare il proprio blocco di matrice A :

$$A_{ik} \in \Re^{\frac{N}{p} \times \frac{N}{p}}$$

a tutti i processi appartenenti al proprio sottocontesto mediante un broadcast. Il tempo di comunicazione è quindi:

$$T_{broad} = \frac{N}{p} \cdot \frac{N}{p} t_{broad} = \frac{N^2}{p^2} t_{broad}$$

Tale spedizione viene eseguita p volte. In definitiva:

¹⁶ Costruita una griglia cartesiana bidimensionale, di dimensione $p \times p$, si definisce sottocontesto riga l'insieme dei processori disposti lungo una delle p righe della griglia. In modo del tutto analogo si definiscono i sottocontesti colonna della griglia.

$$T_{broad} = \frac{N^2}{p^2} \cdot p \cdot t_{broad} = \frac{N^2}{p} t_{broad}$$

Inoltre ogni processore P_{ij} deve inviare il proprio blocco di matrice B :

$$B_{kj} \in \Re^{\frac{N}{p} \times \frac{N}{p}}$$

al processore $P_{mod(i-1,p),j}$. Il tempo di comunicazione è :

$$T_{com} = \frac{N}{p} \cdot \frac{N}{p} t_{com} = \frac{N^2}{p^2} t_{com}$$

Tale spedizione viene eseguita $p - 1$ volte. In definitiva:

$$T_{com} = \frac{N^2}{p^2} \cdot (p - 1) t_{com} = \frac{N^2(p - 1)}{p^2} t_{com}$$

In conclusione, risulta:

Per la strategia 1D-Systolic:

$$T_{1D} \simeq \frac{2N^3}{p} T_{calc} + \frac{N^2(p - 1)}{p} T_{com}$$

Per la strategia 2D-Systolic:

$$T_{2D} \simeq \frac{2N^3}{p^2} T_{calc} + \frac{2N^2(p - 1)}{p^2} T_{com}$$

Per la strategia BMR:

$$T_{BMR} \simeq \frac{2N^3}{p^2} T_{calc} + \frac{N^2(p - 1)}{p^2} T_{com} + \frac{N^2}{p} T_{broad}$$

Da cui segue

$$T_{1D} \geq T_{2D} \text{ per } p > 2$$

e

$$T_{BMR} \leq T_{2D} \leq T_{1D} \text{ se } t_{broad} \leq t_{com}$$

Dunque il metodo BMR risulta più efficiente dei metodi sistolici se il tempo di broadcast è inferiore al tempo di comunicazione tra i processori.

Da osservare inoltre che per tutte e tre le strategie la complessità computazionale è dell'ordine di $O(N^3)$ mentre il costo delle comunicazioni è dell'ordine di $O(N^2)$. Il costo delle comunicazioni cresce quindi più lentamente del costo computazionale:

$$\frac{T_{com}}{T_{calc}} = O\left(\frac{1}{N}\right)$$

*...Scientific Software will lead mathematics
to focus more heavily on
problem solving and algorithms ...*

J. Rice, 1998

*...Il Software Scientifico condurrà la Matematica
a porre sempre di più l'attenzione
su metodi e algoritmi per la risoluzione di problemi ...*

Capitolo 4

I parametri di valutazione del software parallelo

La varietà e la complessità delle architetture esistenti ha accresciuto notevolmente l'influenza dell'ambiente di calcolo sullo sviluppo, e quindi sulla valutazione di algoritmi e software. La struttura SIMD o MIMD, la presenza di una o più memorie condivise o distribuite, lo schema di connessione e la velocità di comunicazione tra i processori, il numero di processori, l'architettura e la potenza di ciascuno di essi, sono solo alcuni dei fattori che influenzano le prestazioni di un algoritmo in ambiente di calcolo parallelo.

Le funzioni classiche relative alla complessità di tempo e alla complessità di spazio utilizzate in ambito sequenziale, non sono più adeguate alla valutazione degli algoritmi paralleli, tanto da condurre alla individuazione di un insieme di parametri, ampio e articolato, che richiede revisioni e modifiche a mano a mano che i sistemi di calcolo evolvono.

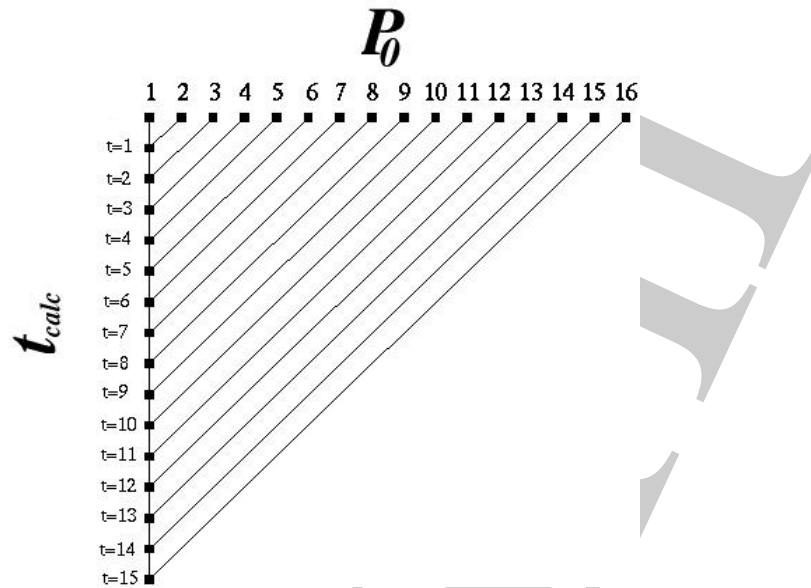


Figura 4.1: I dati sono memorizzati su un processore. Il tempo di esecuzione è $15 t_{calc}$

4.1 La definizione “classica” di speed-up ed efficienza

♣ Esempio 11

Consideriamo la somma di 16 numeri. Indichiamo con p il numero di processori, con T_p il tempo di esecuzione dell'algoritmo su p processori e con t_{calc} il tempo necessario all'esecuzione di una operazione di addizione.

Se $p = 1$ il numero di addizioni eseguite è 15, quindi il tempo di esecuzione è $T_1 = 15 t_{calc}$ (Fig. 4.1).

Se i processori sono due, $p = 2$, il numero di addizioni eseguite è sempre 15, ma $T_2 = 8 t_{calc}$ (Fig. 4.2). I dati sono infatti distribuiti tra i due processori. A ciascun processore vengono assegnati 8 numeri e ciascun processore calcola 7 somme, impiegando quindi un tempo di $7 t_{calc}$. Le somme parziali sono poi sommate in $1 t_{calc}$; il tempo complessivo di calcolo è quindi $8 t_{calc}$.

Ancora, se $p = 4$, $T_4 = 5 t_{calc}$ ogni processore possiede 4 numeri ed esegue 3 somme in un tempo $3 t_{calc}$ per ottenere il risultato parziale, quindi le quattro somme parziali ottenute dovranno essere sommate in un tempo $2 t_{calc}$ così da ottenere il risultato

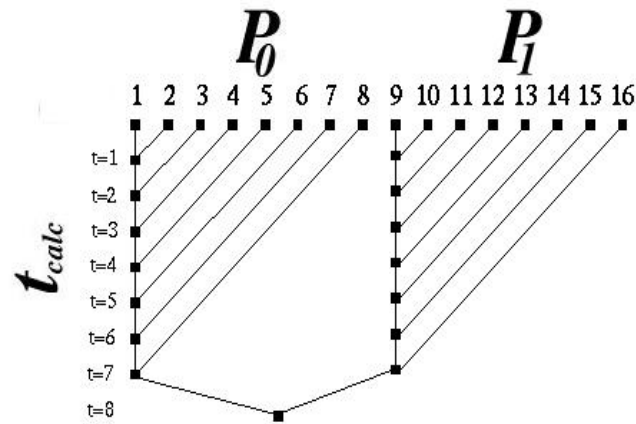


Figura 4.2: Schema esecutivo dell'algoritmo per il calcolo della somma di 16 numeri con $p = 2$ processori: i dati vengono distribuiti su 2 processori. Il tempo di esecuzione è $8 t_{calc}$

finale (Fig. 4.3).

Se invece $p = 8$, $T_8 = 4 t_{calc}$: ogni processore possiede 2 dati e deve eseguire 1 somma in $1 t_{calc}$. Quindi quattro coppie di processori sommano i risultati parziali in $1 t_{calc}$. Al passo successivo due coppie di processori sommano i risultati parziali in $1 t_{calc}$ e infine una coppia di processori ottiene il risultato parziale in $1 t_{calc}$ (Fig. 4.4).

In definitiva, per la somma di sedici numeri sono necessari i tempi di calcolo indicati nella Tab. 4.1, in cui è stato assunto t_{calc} come unità di tempo.

p	T_p
1	$15 t_{calc}$
2	$8 t_{calc}$
4	$5 t_{calc}$
8	$4 t_{calc}$

Tabella 4.1: Nelle colonne sono indicati il numero di processori e il tempo di esecuzione corrispondente.

Ci chiediamo allora quale sia l'algoritmo parallelo più veloce e quanto questo sia più veloce di quello sequenziale. A tal fine si introduce il concetto di *speed up*¹.

△

¹Qui e nel seguito assumiamo che tutti i processori siano omogenei sia rispetto alla velocità di calcolo sia rispetto al carico computazionale dovuto all'esecuzione di altre applicazioni.

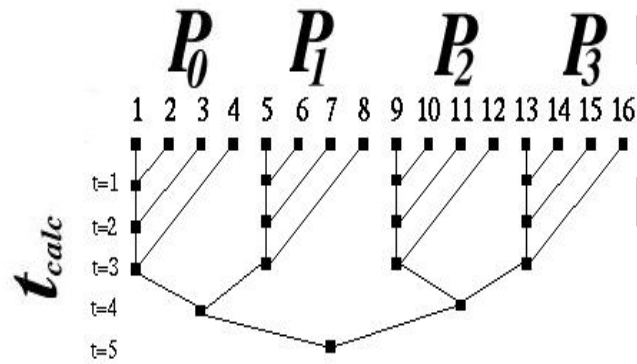


Figura 4.3: Schema esecutivo dell'algoritmo per il calcolo della somma di 16 numeri con $p = 4$ processori: i dati vengono distribuiti su 4 processori. Il tempo di esecuzione è $5 t_{calc}$

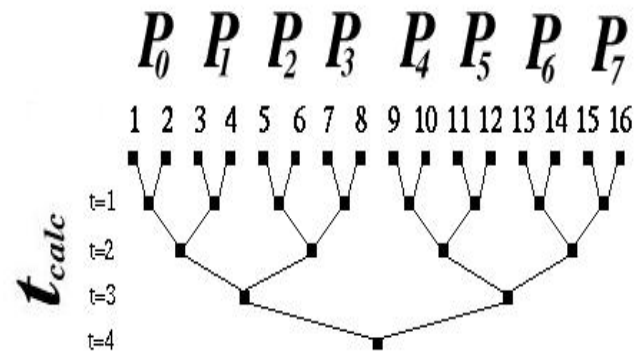


Figura 4.4: Schema esecutivo dell'algoritmo per il calcolo della somma di 16 numeri con $p = 8$ processori: i dati vengono distribuiti su 8 processori. Il tempo di esecuzione è $4 t_{calc}$

Il concetto di speed up trae la sua origine da un'osservazione abbastanza naturale: indicato con T_p il tempo di esecuzione dell'algoritmo su p processori, ci aspettiamo che se $p = 2$, T_1 sia il doppio di T_2 , cioè $T_1/T_2 = 2$. Ancora, se $p = 4$, ci aspettiamo che T_1 sia il quadruplo di T_4 , cioè $T_1/T_4 = 4$. Se si hanno a disposizione p processori per risolvere un dato problema, l'obiettivo è impiegare $1/p$ volte il tempo necessario a risolverlo con un solo processore.

Definizione: se T_1 è il tempo di esecuzione di un algoritmo su 1 processore e T_p è il tempo di esecuzione su p processori, lo speed up S_p è definito così:

$$S_p = \frac{T_1}{T_p}$$

Lo speed up misura la riduzione del tempo di esecuzione, ovvero l'aumento della velocità di esecuzione, rispetto all'algoritmo su un processore.

Come prima cosa bisogna chiarire quale sia l'algoritmo sequenziale di riferimento con cui si misura T_1 . Le scelte possibili sono due:

1. T_1 è il tempo di esecuzione del “miglior algoritmo” sequenziale;
2. T_1 è il tempo di esecuzione dell'algoritmo parallelo su un solo processore.

Se si effettua la prima scelta lo speed up fornisce informazioni sul guadagno che si ottiene risolvendo un problema con p processori anziché con uno solo, ed è questo che interessa se si pone come obiettivo finale la risoluzione del problema nel minor tempo possibile. Purtroppo tale scelta comporta alcune difficoltà: spesso non è possibile stabilire quale sia il miglior algoritmo sequenziale.

Se si effettua la seconda scelta di T_1 , lo speed up dà informazioni

su quanto un algoritmo sia adatto all'implementazione su una data architettura parallela.

La scelta di T_1 , ovvero dell'algoritmo sequenziale di riferimento, non è in generale un problema semplice, e richiede di volta in volta un'analisi attenta dell'algoritmo e delle risorse hardware/software a disposizione.

Poiché risulta al più $T_1 = p \cdot T_p$, dalla definizione di speed up segue che $S_p \leq p$ ². Si definisce quindi

$$S_p^{ideale} = p$$

lo *speed up ideale* e l'algoritmo parallelo risulta migliore quanto più S_p è prossimo a p .

Ritornando all'esempio della somma di sedici numeri si ha la seguente tabella:

p	T_p	S_p	S_p^{ideale}
1	$15 t_{calc}$	1.00	1
2	$8 t_{calc}$	1.88	2
4	$5 t_{calc}$	3.00	4
8	$4 t_{calc}$	3.75	8

Tabella 4.2: Nelle colonne sono indicati il numero di processori e i corrispondenti tempo di esecuzione, speed up e speed up ideale.

Dalla Tab. 4.2 è evidente che con 8 processori si è avuta la riduzione maggiore del tempo di esecuzione. Sembrerebbe quindi che l'algoritmo “migliore” sia quello con 8 processori. Da un'analisi più attenta ci accorgiamo però che non è così, perché sebbene in questo caso sia evidente che quanti più processori si usano tanto più si riduce il tempo di esecuzione, ciò non sempre significa che

²Sebbene esistano casi in cui può accadere che $S_p > p$. In tal caso si parla di speed up “superlineare”. Lo speed up superlineare è dovuto spesso alla presenza di situazioni di overhead nell'algoritmo sequenziale, come accessi alla memoria non efficienti.

l'ambiente di calcolo sia stato utilizzato in maniera efficace. Lo speed up su 2 processori infatti è “il più vicino” allo speed up ideale corrispondente, quindi sembrerebbe che il miglior uso dei processori si abbia per $p = 2$.

Da questo esempio appare chiaro che accanto alla misura dello speed up conviene avere una informazione di quanto siano state utilizzate le risorse di calcolo parallelo. In altre parole, bisogna “rapportare” lo speed-up al numero di processori.

Definizione: Sia p il numero di processori e S_p lo speed-up ad esso relativi. Si definisce efficienza E_p il parametro

$$E_p = \frac{S_p}{p}$$

che fornisce un'indicazione di quanto sia stato utilizzato il parallelismo del calcolatore. Evidentemente $E_p \leq 1$.

Se si considera S_p^{ideale} si definisce:

$$E_p^{ideale} = 1$$

l'efficienza ideale e l'algoritmo parallelo risulta migliore quanto più E_p è prossimo ad 1.

Nel caso della somma di 16 numeri su p processori si ha:

p	E_p
1	1.00
2	0.94
4	0.75
8	0.47

Tabella 4.3: Nelle colonne sono indicati il numero di processori utilizzati e la corrispondente efficienza.

Si osserva che per $p = 2$ il parametro efficienza è più prossimo ad 1. Quindi, l'algoritmo per il calcolo della somma di 16 numeri su $p = 2$ processori è quello che sfrutta meglio il parallelismo.

La valutazione dei tempi di esecuzione precedente è molto semplificata, perché non tiene conto di alcuni fattori di fondamentale importanza. Ad esempio, nella somma di n numeri, oltre al tempo di calcolo, l'algoritmo richiede la distribuzione dei dati tra i processori. Tutte queste operazioni devono rientrare nella valutazione delle prestazioni di un algoritmo.

♣ Esempio 12

Si esegua la somma di $N = 16$ numeri su di un calcolatore MIMD a memoria distribuita. È stato già osservato che:

- $p = 2 \implies T_2 = 8 t_{calc};$
- $p = 4 \implies T_4 = 5 t_{calc};$
- $p = 8 \implies T_8 = 4 t_{calc};$

Consideriamo ora anche il tempo necessario per la comunicazione tra i processori. Denotiamo con t_{com} il tempo necessario ad effettuare la comunicazione di un dato f.p.. Dalla Fig. 4.2 si osserva che se $p = 2$ il tempo di comunicazione è $1 t_{com}$. Dalla Fig. 4.3 si osserva che se $p = 4$ il tempo di comunicazione è $2 t_{com}$. Dalla Fig. 4.4 si osserva che se $p = 8$ il tempo di comunicazione è $3 t_{com}$.

In definitiva, se si tiene conto sia del tempo di calcolo che del tempo di comunicazione si ha:

p	T_p
2	$T_2 = 8 t_{calc} + 1 t_{com}$
4	$T_4 = 5 t_{calc} + 2 t_{com}$
8	$T_8 = 4 t_{calc} + 3 t_{com}$

Tabella 4.4: Nelle colonne sono indicati il numero di processori utilizzati ed il corrispondente tempo di esecuzione, visto come somma del tempo di calcolo e del tempo di comunicazione.

Supponiamo ora che risulti:

$$\frac{t_{com}}{t_{calc}} = 2$$

A seconda che si consideri o meno il tempo necessario alla comunicazione, essendo

$$T_1 = 15 t_{calc}$$

si ottengono i valori seguenti di T_p , S_p ed E_p :

p	T_p	S_p	E_p
2	$8 t_{calc}$	1.88	0.94
4	$5 t_{calc}$	3.00	0.75
8	$4 t_{calc}$	3.75	0.47

p	T_p	S_p	E_p
2	$10 t_{calc}$	1.50	0.75
4	$9 t_{calc}$	1.67	0.42
8	$10 t_{calc}$	1.50	0.19

Tabella 4.5: Nelle tabelle sono indicati il numero di processori ed i tempi di esecuzione, speed up ed efficienza corrispondenti. Nella prima tabella T_p coincide con il tempo di calcolo; nella seconda T_p rappresenta la somma del tempo di calcolo e del tempo di comunicazione.

Si osserva dunque che considerando il tempo di comunicazione le prestazioni dell'algoritmo possono cambiare notevolmente.

△

Definizione: si definisce overhead di comunicazione unitario la quantità:

$$OC = \frac{t_{com}}{t_{calc}}$$

L'overhead è un parametro dipendente dall'ambiente computazionale (calcolatore, software di comunicazione, ...). In generale, $OC > 1$. Indicato con T_p^{com} il tempo di comunicazione dell'algoritmo su p processori e con T_p^{calc} il tempo di calcolo dell'algoritmo

su p processori, la quantità

$$OC_p = \frac{T_p^{com}}{T_p^{calc}} = \frac{n \cdot t_{com}}{m \cdot t_{calc}} = \frac{n \cdot OC \cdot t_{calc}}{m \cdot t_{calc}} = \frac{n \cdot OC}{m}$$

è detta *overhead di comunicazione* e fornisce una misura del “peso” della comunicazione sul tempo di esecuzione dell’algoritmo.

♣ Esempio 13

Tornando all’**Esempio 12** della somma di $N = 16$ numeri, supposto $OC = 2$, si ha:

p	T_p^{calc}	T_p^{com}	OC_p
2	$8 t_{calc}$	$1 t_{com}$	0.25
4	$5 t_{calc}$	$2 t_{com}$	0.80
8	$4 t_{calc}$	$3 t_{com}$	1.50

Tabella 4.6: Nelle colonne sono indicati il numero di processori ed i tempi di calcolo, di comunicazione e l’overhead di comunicazione corrispondenti.

Si osserva che su $p = 8$ processori il tempo di comunicazione pesa di più rispetto al tempo di esecuzione. Tale risultato era, d’altra parte, atteso. Fissata, infatti, la dimensione N del problema, all’aumentare del numero di processori aumenta il numero di comunicazioni da effettuare.

△

Nel caso *ideale* si ha che

$$S_p = p \Rightarrow T_1 = p \cdot T_p$$

mentre in generale:

$$S_p < p \Rightarrow \frac{T_1}{T_p} < p$$

Abbiamo la seguente definizione:

Definizione: Si definisce Overhead totale

$$O_h = pT_p - T_1 \quad (4.1)$$

con

$$O_h > 0$$

Oltre al tempo di comunicazione bisogna tener conto del **tempo di sincronizzazione**, del tempo cioè necessario per coordinare il lavoro dei vari processi in esecuzione. Infatti questi ultimi possono avere bisogno, per il calcolo della soluzione, di scambiare dati. Ad esempio, nel caso della somma di n numeri il processore P_0 deve attendere i dati dagli altri processori per poter effettuare la somma. È importante quindi che le operazioni di spedizione e ricezione di messaggi siano sincronizzate al fine di evitare tempi di attesa (idle time) e quindi di inutilizzo dei processori. A tal proposito è quindi anche importante che il carico computazionale sia **ben bilanciato**: i dati devono essere suddivisi tra i processori in modo tale che ognuno di essi impieghi approssimativamente lo stesso tempo per la ricerca della soluzione del problema. Una non equa distribuzione del carico computazionale potrebbe far sì che alcuni processi rimangano inattivi mentre altri devono ancora effettuare numerose istruzioni, riducendo così l'efficienza dell'algoritmo.

♣ Esempio 14

Supponiamo di effettuare la somma di $N = 16$ numeri su due processori assegnando al primo processore 12 numeri e al secondo soltanto 4. Il primo passo è il calcolo delle somme parziali: il primo processore impiega $11 t_{calc}$ per sommare i 12 numeri, il secondo processore impiega $3 t_{calc}$ per sommare 4 numeri. Questo vuol dire che il

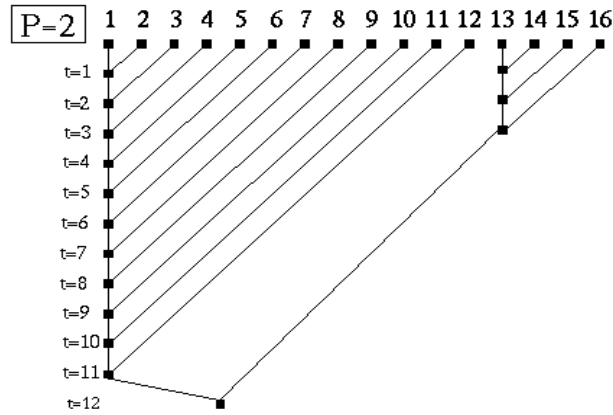


Figura 4.5: Se il carico computazionale non è ben distribuito tra i processori può accadere che alcuni di questi rimangano inattivi.

secondo processore rimane inattivo per un tempo uguale a

$$11 t_{calc} - 8 t_{calc} = 3 t_{calc}$$

in attesa che il primo processore completi la somma parziale, come mostrato in Fig. 4.5.

Calcoliamo speed-up ed efficienza. Il tempo necessario a calcolare la somma totale su di 1 processore è $15 t_{calc}$. Con i dati così distribuiti vengono eseguite in parallelo 6 somme su 15 ed in sequenziale 9 somme su 15, 8 dovute ai numeri in più che il processore P_0 ha avuto assegnati e 1 per sommare le due somme parziali. Risulta quindi :

$$T_2 = 12 t_{calc}$$

effettuando 3 somme in parallelo e 9 in sequenziale,

$$S_2 = \frac{15}{12} = 1,25$$

$$E_2 = \frac{1,25}{2} = 0,62$$

Effettuando la somma di $N = 16$ numeri con 2 processori e distribuzione bilanciata, si ha (si veda Esempio 11):

$$\bullet T_2 = 8 t_{calc}$$

e quindi

$$S_2 = \frac{15}{8} = 1,875$$

$$E_2 = \frac{1,875}{2} = 0,9375$$

Nel caso di distribuzione bilanciata si osserva, quindi, un miglioramento sia dello speed-up che dell'efficienza.

△

In conclusione, la valutazione delle prestazioni di un algoritmo parallelo deve tener conto del:

- numero di processori/processi;
- tempo di calcolo;
- tempo di comunicazione;
- tempo di sincronizzazione dei processori;
- bilanciamento del carico computazionale.

Questo rende necessario l'utilizzo di più parametri di valutazione (T_p , S_p , E_p , OC_p).

Tali fattori dipendono strettamente, oltre che dall'algoritmo, dall'ambiente di elaborazione, in particolare da:

- numero di processori;
- architettura e potenza dei processori;
- tipo e numero di memorie;
- connessione tra i processori;
- connessioni tra processori e memorie;
- software message passing.

Quindi, in generale, una valutazione effettiva delle prestazioni di un algoritmo non può prescindere dalla misura del tempo richiesto dal software che implementa l'algoritmo in uno specifico ambiente di elaborazione.

4.2 Il modello scalato di speed-up ed efficienza

È stato già detto che lo speed-up S_p risulta sempre minore di p , cioè che $S_p \leq p$. Ci chiediamo se sia possibile ottenere speed up prossimi allo speed up ideale.

In generale un algoritmo è costituito da una *parte seriale* (inizializzazioni di variabili, ricostruzione della soluzione del problema a partire da quelle dei sottoproblemi in cui è ripartito) e da una *parte parallela* (insieme di istruzioni che sono eseguite concorrentemente).

♣ Esempio 15

Si consideri la somma di N numeri su p processori. In particolare sia $N = 4$ e $p = 2$.

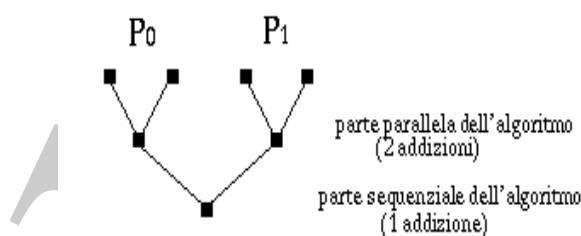


Figura 4.6: Nel grafo ad albero dell'algoritmo della somma di N numeri è evidenziata la parte parallela da quella sequenziale.

Poiché le addizioni sono 3, ($T_1 = 3 t_{com}$) indichiamo con

$$\alpha = \frac{1}{3}$$

la frazione di T_1 addizioni eseguita sequenzialmente e con

$$\alpha' = 1 - \alpha = \frac{2}{3}$$

la frazione di addizioni eseguita in parallelo da 2 processori.

Risulta allora:

$$T_2 = \frac{1}{3}T_1 + \frac{\frac{2}{3}T_1}{2}$$

e

$$S_2 = \frac{T_1}{T_2} = \frac{T_1}{\frac{1}{3}T_1 + \frac{\frac{2}{3}T_1}{2}} = \frac{3}{2}$$

△

In generale, detta α la frazione di T_1 di operazioni dell'algoritmo eseguite in sequenziale ($0 \leq \alpha \leq 1$) e

$$\alpha' = (1 - \alpha)$$

la frazione di T_1 delle operazioni dell'algoritmo eseguite in parallelo ($0 \leq \alpha' \leq 1$), se si assume che nell'esecuzione concorrente della frazione parallela gli overhead di comunicazione siano trascurabili, risulta

$$T_p = \alpha \cdot T_1 + \frac{(1 - \alpha)T_1}{p}$$

Inoltre, sussiste la seguente:

Definizione: Sia T_1 il tempo di esecuzione di un algoritmo su 1 processore e T_p il tempo di esecuzione su p processori. Se α è la frazione sequenziale dell'algoritmo e $\alpha' = (1 - \alpha)$ è la frazione parallela dell'algoritmo, si ha il modello teorico di speed up seguente

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

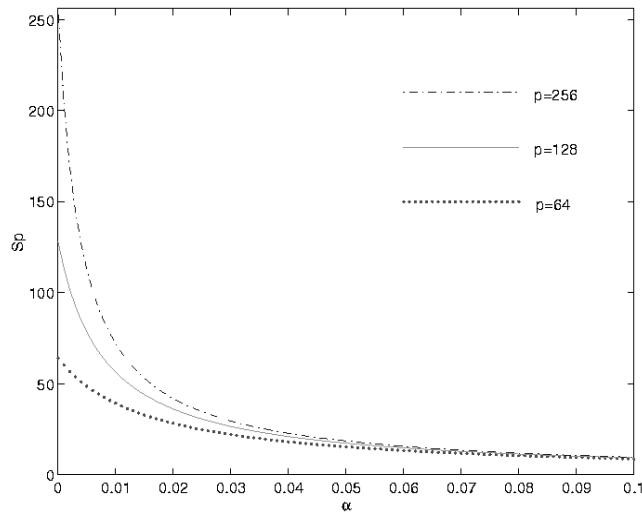


Figura 4.7: In figura è riportato il valore dello speed up in funzione di α per i valori di $p = 64, 128, 256$

noto come legge di Ware [50] o di Amdhal.

αT_1 costituisce la parte seriale del tempo di esecuzione dell'algoritmo su p processori e $\frac{(1-\alpha)T_1}{p}$ la parte parallela. Da tale relazione si osserva che, assumendo che α sia costante rispetto a p , poiché:

$$\lim_{p \rightarrow \infty} \frac{1-\alpha}{p} = 0$$

risulta

$$S_p \leq \frac{1}{\alpha}$$

Ovvero la parte sequenziale può degradare fortemente lo speed up (vedi Fig. 4.7).

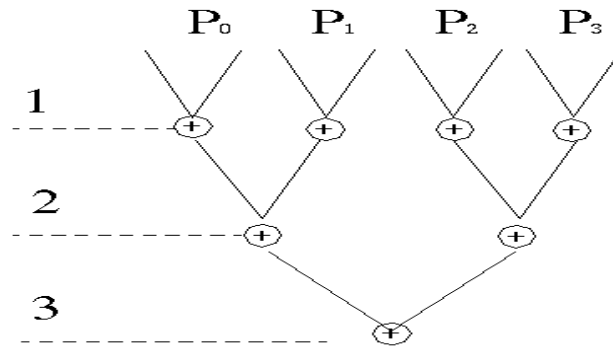


Figura 4.8: Schema delle comunicazioni tra 4 processori che eseguono la somma di N numeri

♣ Esempio 16

Consideriamo la somma di $N = 8$ numeri su $p = 4$ processori (Fig. 4.8).

1. Al passo 1 vengono eseguite 4 addizioni da 4 processori.
2. Al passo 2 vengono eseguite 2 addizioni da 2 processori.
3. Al passo 3 viene eseguita 1 addizione da 1 processore.

Il passo 1 costituisce la parte parallela dell'algoritmo, la parte cioè dove vengono eseguite le operazioni concorrentemente da tutti i processori. Il passo 3 costituisce la parte sequenziale, la parte cioè dove vengono eseguite le operazioni da un solo processore. Cerchiamo di definire il contributo del passo 2 in cui vengono eseguite operazioni solo da un certo numero di processori $\tilde{p} < p = 4$.

Nel nostro caso risulta

$$T_1 = 7 t_{calc}$$

Al passo 1 la frazione di T_1 relativa alle 4 addizioni eseguite dai 4 processori è

$$\alpha_4 = \frac{4}{7}$$

e il contributo a T_4 , tempo di esecuzione su 4 processori, è

$$\alpha_4 \cdot \frac{T_1}{4}$$

Al passo 3 la frazione di T_1 relativa all'addizione eseguita da un processore è

$$\alpha_1 = \frac{1}{7}$$

e il contributo a T_4 è

$$\alpha_1 \cdot T_1$$

Consideriamo quindi le 2 addizioni eseguite dai 2 processori al passo 2. Risulta

$$\alpha_2 = \frac{2}{7}$$

e il contributo a T_4 è

$$\alpha_2 \cdot \frac{T_1}{2}$$

In generale, indicato con i il numero di processori che contribuiscono ad α_i si ha

$$T_p = \sum_{i=1}^p \alpha_i \frac{T_1}{i}$$

dove T_p rappresenta il tempo di esecuzione relativo alle operazioni eseguite da tutti i processori.

Nel caso in cui $p = 4$ e $T_1 = 7$ si ha:

$$T_4 = \alpha_1 T_1 + \alpha_2 \frac{T_1}{2} + \alpha_4 \frac{T_1}{4} = \frac{1}{7} \cdot 7 + \frac{2}{7} \cdot \frac{7}{2} + \frac{4}{7} \cdot \frac{7}{4} = 3$$

△

Detta α_i la frazione di T_1 eseguita da i processori ($1 \leq i \leq p$) (parallelismo medio), la formulazione della legge di Ware più generale è la seguente:

$$S_p = \frac{T_1}{\alpha_1 T_1 + \sum_{k=2}^{p-1} \alpha_k \cdot \frac{T_1}{k} + \alpha_p \cdot \frac{T_1}{p}}$$

♣ Esempio 17

Consideriamo la somma di $N = 32$ numeri su $p = 2$ processori come mostrato in Fig. 4.9/A.

In questo caso entrambi i processori eseguono la somma parziale di 16 numeri impiegando un tempo

$$T = 15 t_{calc}$$

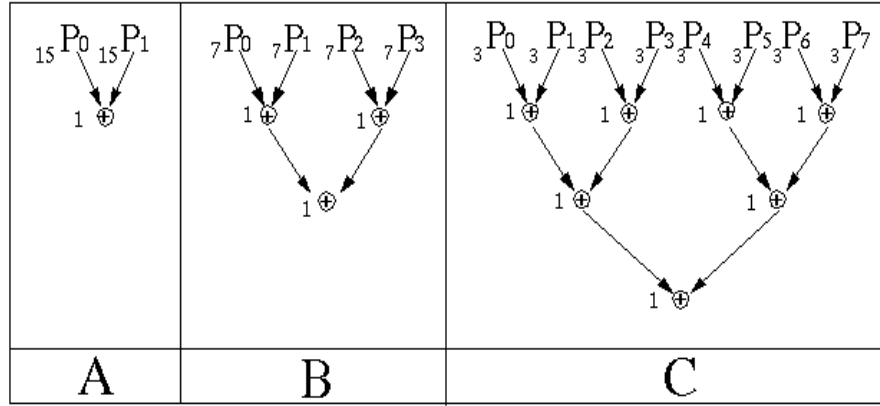


Figura 4.9: Schematizzazione della somma di $N = 32$ numeri rispettivamente su $p = 2$ processori, $p = 4$ processori e $p = 8$ processori. Ad ogni passo è indicato il numero di somme effettuate da ciascun processore.

successivamente le due somme parziali si devono sommare. Risulta:

$$T_1 = N - 1 = 31$$

$$\alpha_1 = \frac{1}{31} = 0,03 \quad \alpha_2 = \frac{30}{31} = 0,96 \implies T_2 = \alpha_1 T_1 + \alpha_2 \frac{T_1}{2} = 16$$

$$S_2 = \frac{T_1}{T_2} = \frac{31}{16} = 1,9$$

$$E_2 = \frac{S_2}{p} = \frac{1,9}{2} = 0,95$$

Consideriamo ora la somma di $N = 32$ numeri su $p = 4$ processori come mostrato in Fig. 4.9/B.

In questo caso, al primo passo 4 processori calcolano le somme parziali impiegando ognuno un tempo

$$T = 7 t_{calc}$$

(la parte parallela). Al secondo passo solo 2 processori eseguono le somme. Al terzo passo solo 1 processore esegue una somma. Risulta:

$$\alpha_1 = \frac{1}{31} \quad \alpha_2 = \frac{2}{31} \quad \alpha_4 = \frac{28}{31}$$

$$T_4 = \alpha_1 T_1 + \alpha_2 \frac{T_1}{2} + \alpha_4 \frac{T_1}{4} = \frac{1}{31} \cdot 31 + \frac{2}{31} \cdot \frac{31}{2} + \frac{28}{31} \cdot \frac{31}{4} = 1 + 1 + 7 = 9$$

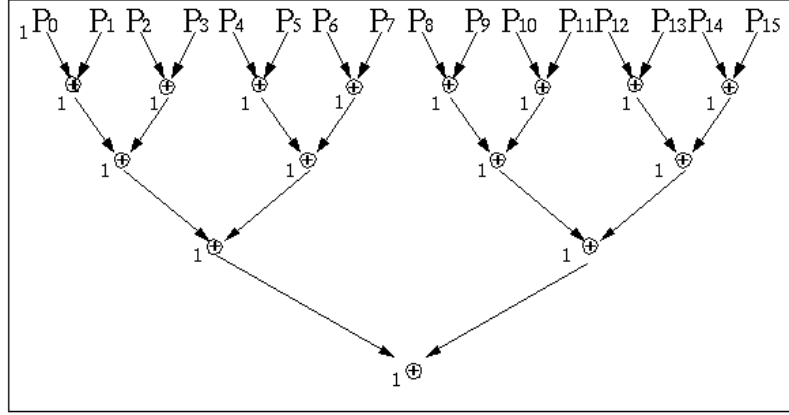


Figura 4.10: Schematizzazione della somma di $N = 32$ numeri su $p = 16$ processori. Ad ogni passo è indicato il numero di somme effettuate da ciascun processore.

$$S_4 = \frac{T_1}{T_4} = \frac{31}{9} = 3,4$$

$$E_4 = \frac{S_4}{p} = \frac{3,4}{4} = 0,85$$

Consideriamo ora la somma di $N = 32$ numeri su $p = 8$ processori come mostrato in Fig. 4.9/C.

In questo caso, al primo passo 8 processori calcolano le somme parziali impiegando ognuno un tempo

$$T = 3 t_{calc}$$

(parte parallela). Al secondo passo 4 processori eseguono 1 somma. Al terzo passo solo 2 processori eseguono 1 somma. Al quarto passo 1 processore esegue 1 somma. Risulta:

$$\alpha_1 = \frac{1}{31} \quad \alpha_2 = \frac{2}{31} \quad \alpha_4 = \frac{4}{31} \quad \alpha_8 = \frac{(31-7)}{31} = \frac{24}{31}$$

$$T_8 = \alpha_1 T_1 + \alpha_2 \frac{T_1}{2} + \alpha_4 \frac{T_1}{4} + \alpha_8 \frac{T_1}{8} =$$

$$= \frac{1}{31} \cdot 31 + \frac{2}{31} \cdot \frac{31}{2} + \frac{4}{31} \cdot \frac{31}{4} + \frac{24}{31} \cdot \frac{31}{8} = 1 + 1 + 1 + 3 = 6$$

$$S_8 = \frac{T_1}{T_8} = \frac{31}{6} = 5,1$$

$$E_8 = \frac{S_8}{p} = \frac{5,1}{8} = 0,6$$

Consideriamo ora la somma di $N = 32$ numeri su $p = 16$ processori, come mostrato in Fig. 4.10.

Al primo passo 16 processori calcolano le somme parziali impiegando ognuno un tempo

$$T = 1 \, t_{calc}$$

(parte parallela). Al secondo passo 8 processori eseguono 1 somma. Al terzo passo 4 processori eseguono 1 somma. Al quarto passo 2 processori eseguono 1 somma. Al quinto passo 1 processore esegue 1 somma. Risulta:

$$\begin{aligned} \alpha_1 &= \frac{1}{31} & \alpha_2 &= \frac{2}{31} & \alpha_4 &= \frac{4}{31} & \alpha_8 &= \frac{8}{31} & \alpha_{16} &= \frac{(31-15)}{31} = \frac{16}{31} \\ T_{16} &= \alpha_1 T_1 + \alpha_2 \frac{T_1}{2} + \alpha_4 \frac{T_1}{4} + \alpha_8 \frac{T_1}{8} + \alpha_{16} \frac{T_1}{16} = \\ &= \frac{1}{31} \cdot 31 + \frac{2}{31} \cdot \frac{31}{2} + \frac{4}{31} \cdot \frac{31}{4} + \frac{8}{31} \cdot \frac{31}{8} + \frac{16}{31} \cdot \frac{31}{16} = 1 + 1 + 1 + 1 + 1 = 5 \\ S_{16} &= \frac{T_1}{T_{16}} = \frac{31}{5} = 6,2 \\ E_{16} &= \frac{S_{16}}{p} = \frac{6,2}{16} = 0,3 \end{aligned}$$

Quindi, se $N = 32$ si ha:

p	parte sequenziale	parte parallela	S_p	E_p
2	0,03	0,96	1,9	0,95
4	0,1	0,9	3,4	0,85
8	0,3	0,7	5,1	0,6
16	0,5	0,5	6,2	0,3

Tabella 4.7: Nelle colonne sono indicati il numero di processori e le corrispondenti parti sequenziale e parallela, oltre che lo speed up e l'efficienza.

Si osserva quindi che il peso della frazione sequenziale aumenta all'aumentare del numero di processori. Se N è fissato, al crescere del numero di processori, la parte sequenziale domina quella parallela, facendo così degradare speed-up ed efficienza.

△

Come si può osservare dagli esempi precedenti, se la dimensione del problema è fissata, al crescere del numero di processori non solo non si riescono ad ottenere speed-up vicini a quello ideale, ma le prestazioni peggiorano. Quindi non conviene utilizzare un numero maggiore di processori.

Vediamo allora cosa accade alla parte sequenziale e alla parte parallela, quindi allo speed-up e all'efficienza, se fissiamo il numero di processori e facciamo variare la dimensione N del problema.

♣ Esempio 18

Consideriamo la somma di N numeri su $p = 2$ processori.

Per $N = 8$ risulta $\alpha_1 = \frac{1}{7}$, $S_2 = \frac{7}{4} = 1,75$, $E_2 = \frac{1,75}{2} = 0,875$

Per $N = 16$ risulta $\alpha_1 = \frac{1}{15}$, $S_2 = \frac{15}{8} = 1,8$, $E_2 = \frac{1,8}{2} = 0,9$

Per $N = 32$ risulta $\alpha_1 = \frac{1}{31}$, $S_2 = \frac{31}{16} = 1,9$, $E_2 = \frac{1,9}{2} = 0,96$

Per $N = 64$ risulta $\alpha_1 = \frac{1}{63}$, $S_2 = \frac{63}{32} = 1,96$, $E_2 = \frac{1,96}{2} = 0,99$

N	parte sequenziale	parte parallela	$S_2(N)$	$E_2(N)$
8	0,14	0,86	1,75	0,875
16	0,06	0,93	1,8	0,9
32	0,03	0,96	1,9	0,96
64	0,01	0,98	1,96	0,99

Tabella 4.8: *Nelle colonne sono indicate la dimensione del problema e le parti sequenziale e parallela, lo speed up e l'efficienza corrispondenti.*

Come si osserva dalla Tab. 4.8, aumentando la dimensione del problema, la parte sequenziale α_1 va a zero.

△

♣ Esempio 19

Vediamo cosa accade se aumenta sia la dimensione N del problema sia il numero p di processori, in modo che sia costante N/p .

Per $N = 8$ e $p = 2$ risulta

$$\alpha_1 = \frac{1}{7} = 0,14, \alpha_2 = \frac{6}{7} = 0,85,$$

$$S_2(8) = \frac{7}{4} = 1,75, E_2(8) = \frac{1,75}{2} = 0,875$$

Per $N = 16$ e $p = 4$ risulta

$$\alpha_1 = \frac{1}{15} = 0,06, \alpha_4 = \frac{12}{15} = 0,8,$$

$$S_4(16) = \frac{15}{5} = 3, E_4(16) = \frac{3}{4} = 0,75$$

Per $N = 32$ e $p = 8$ risulta

$$\alpha_1 = \frac{1}{31} = 0,03, \alpha_8 = \frac{24}{31} = 0,77,$$

$$S_8(32) = \frac{31}{6} = 5,1, E_8(32) = \frac{5,1}{8} = 0,64$$

Per $N = 64$ e $p = 16$ risulta

$$\alpha_1 = \frac{1}{63} = 0,01, \alpha_{16} = \frac{48}{63} = 0,76,$$

$$S_{16}(64) = \frac{63}{7} = 9, E_{16}(64) = \frac{9}{16} = 0,56$$

$N/p = 4$	parte sequenziale	parte parallela	$S_p(N)$	$E_p(N)$
$N = 8, p = 2$	0,14	0,86	1,75	0,875
$N = 16, p = 4$	0,06	0,8	3	0,75
$N = 32, p = 8$	0,03	0,77	5,1	0,64
$N = 64, p = 16$	0,01	0,76	9	0,56

Tabella 4.9: Nella prima colonna sono indicati la dimensione del problema ed il numero di processori; nelle colonne successive le parti sequenziale e parallela, lo speed up e l'efficienza corrispondenti.

Dalla Tab. 4.9 risulta che, aumentando la dimensione del problema e il numero di processori, rimane “quasi costante” la parte parallela mentre decresce la parte sequenziale.

△

In base alla legge di Ware, al crescere del numero di processori appare sempre più difficile, quasi impossibile, un uso efficiente di un calcolatore parallelo se si tiene fissa la dimensione del problema. D'altra parte, per una dimensione del problema fissata è quasi naturale aspettarsi che ad un certo punto non ha più senso aumentare il numero di processori in quanto gli overhead di comunicazione sarebbero prevalenti rispetto al tempo di calcolo.

Ciascuna classe di problemi è quindi caratterizzata da una propria *granularità*³, al di sotto della quale gli overhead di comunicazione e di sincronizzazione diventano tali da degradare significativamente lo speed up e l'efficienza.

³Quantità di calcolo tra due punti successivi di sincronizzazione.

Questo significa che la legge di Ware non tiene conto, almeno esplicitamente, della dimensione del problema.

Dall'esempio precedente si osserva però che al crescere della dimensione di un problema la frazione sequenziale di un algoritmo decresce significativamente all'aumentare del numero di processori⁴. Dunque, se p è fissato ed $N \rightarrow \infty$, la parte sequenziale α tende a zero (vedi Tab. 4.9):

$$\alpha + \frac{1 - \alpha}{p} \rightarrow \frac{1}{p} \quad e \quad S_p \rightarrow p$$

In effetti, la parte sequenziale α di un algoritmo, è funzione sia della dimensione del problema sia del numero di processori, cioè

$$\alpha = \alpha(N, p)$$

con

$$\alpha = \alpha(N, p) \rightarrow 0 \quad se \quad \begin{cases} N \rightarrow \infty \\ p = p_0 \end{cases}$$

$$\alpha = \alpha(N, p) \rightarrow \infty \quad se \quad \begin{cases} p \rightarrow \infty \\ N = N_0 \end{cases}$$

Pertanto, questo modello prevede speed-up prossimi a quello ideale solo se si fissa p e si fa crescere N ; d'altro canto però non è possibile aumentare N in maniera arbitraria perché le risorse (hardware) disponibili sono limitate. In altre parole, la legge di Amadhal, sembra fornire un modello attendibile solo se $p \leq p_0$ e $N \leq N_0$, con N_0 e p_0

⁴A tal proposito si ricorda la definizione data da C. Moler, che considera un algoritmo "effettivamente parallelo" quando la sua frazione sequenziale α è tale che $\alpha \rightarrow 0$ per $n \rightarrow \infty$, cioè quando la parte parallela è predominante [44] .

opportunamente prefissati.

Cosa succede, allora, se, aumentando il numero di processori p , utilizziamo lo stesso algoritmo per risolvere lo stesso problema di dimensione maggiore?

♣ Esempio 20

Si supponga di risolvere un problema di dimensione $N = 2$ su $p = 1$ processori in un tempo t . Ci si aspetta quindi di risolvere nello stesso tempo t :

- su $p = 2$ processori un problema di dimensione $N = 4 = 2 * 2 = N_{loc} \times p$;
- su $p = 4$ processori un problema di dimensione $N = 8 = 2 * 4 = N_{loc} \times p$;
- su $p = 8$ processori un problema di dimensione $N = 16 = 2 * 8 = N_{loc} \times p$,

avendo indicato con N_{loc} la dimensione del problema su 1 processore, in particolare $N_{loc} = 2$.

△

In generale, se

$$T_p(N) = t$$

ci si aspetta

$$T_{2p}(2N) = t, T_{4p}(4N) = t \dots$$

ovvero che

$$T_p(N) = T_{2p}(2N) = T_{4p}(4N) = \dots$$

cioè che all'aumentare del numero di processori, nello stesso tempo riesco a risolvere un problema di dimensioni più grandi.

Nasce quindi l'esigenza di elaborare un nuovo modello alla base del quale ci sia la considerazione che la “*dimensione*” di un proble-

ma debba aumentare al crescere della potenza computazionale a disposizione⁵.

Nel 1988 Gustafson ha proposto un nuovo modello per la stima delle prestazioni di un algoritmo parallelo [29]. Alla base del nuovo modello c'è la considerazione che, avendo a disposizione elevate potenze di calcolo, la complessità computazionale dell'algoritmo sul singolo processore, “*scala*” con la potenza di calcolo. La quantità fissa non è la dimensione del problema, come nella formulazione classica dello speed up, ma piuttosto il tempo che è necessario alla risoluzione del problema (*fixed - time model*)⁶.

Sia α' la frazione sequenziale di un algoritmo eseguito su p processori e $1 - \alpha'$ la frazione parallela dell'algoritmo eseguito su p processori. Si trascurino gli overhead di comunicazione e di sincronizzazione. L'esecuzione di tale algoritmo su un solo processore richiede allora un tempo

$$T_1 = \alpha' T_p + p(1 - \alpha') T_p$$

Definizione: si definisce *speed up scalato* la quantità:

$$SS_p = \frac{T_1}{T_p} = \frac{\alpha' T_p + p(1 - \alpha') T_p}{T_p} = \alpha' + p(1 - \alpha') = p + (1 - p)\alpha'$$

Tale modello di *speed-up* è dovuto a Gustafson [28, 29].

In contrasto alla curva rappresentata in Fig. 4.7, il grafico dello speed up scalato in funzione di α' , mostrato in Fig. 4.11, è una retta con pendenza $1 - p$. Quando lo speed-up viene misurato scalando la

⁵In effetti, è la complessità computazionale del problema che deve aumentare se aumenta il numero di processori. Avendo qui trattato il problema della somma con una complessità computazionale lineare rispetto alla dimensione del problema, abbiamo assimilato complessità di calcolo con dimensione del problema.

⁶Poiché il modello dovuto ad Amadhal è stato sviluppato mantenendo fissa la dimensione del problema, lo speed up classico è detto anche fixed-sized e quello scalato fixed-time.

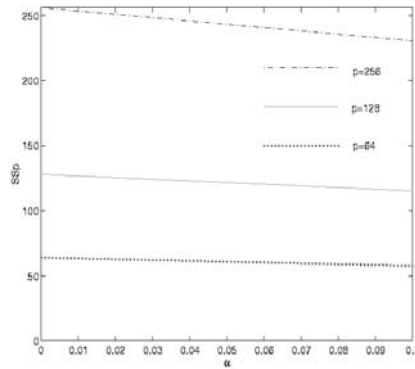


Figura 4.11: In figura è riportato il valore dello speed up scalato in relazione ad α per diversi valori di p

dimensione del problema, la frazione α' tende a zero all'aumentare del numero di processori utilizzati. È quindi auspicabile ottenere prestazioni elevate.

È probabile che aumentando la dimensione del problema, questo non possa essere risolto da un solo processore. Come misurare quindi lo speed-up scalato $\frac{T_1}{T_p}$?

Dall'esempio precedente risulta che

$$T_1(N) = T_1(N_{loc} \times p) ,$$

si assume allora che, almeno nel caso dell'algoritmo della somma

$$T_1(N) = p \times T_1(N_{loc})$$

cioè si assume $T_1(N)$ uguale al tempo che si ottiene moltiplicando per p il tempo per risolvere su 1 processore il problema di dimensione N_{loc} . Più precisamente, se T_p misura il tempo di esecuzione dell'algoritmo su p processori necessario a risolvere un problema di

dimensione

$$N = p \times N_{loc} ,$$

si assume T_1 uguale a quello che si ottiene moltiplicando per p il tempo necessario per risolvere su 1 processore il problema di dimensione $N_{loc} = \frac{N}{p}$. Si ha cioè la seguente definizione operativa per il calcolo dello speed up scalato:

$$SS_p = \frac{pT_1(N_{loc})}{T_p(pN_{loc})}$$

♣ Esempio 21

Consideriamo la somma di $N = np$ numeri su p processori. Sia $N = 32$ e $p = 4$. Calcoliamo

$$S_p = \frac{T_1(np)}{T_p(np)}$$

Dalla definizione si ha:

$$T_1(np) = \alpha'_p(np) + pf'_p(np)T_p$$

avendo denotato con f'_p la quantità $1 - \alpha'_p$. Quindi:

$$T_1(32) = \alpha'_4(32) + 4f'_4(32)T_4(32) \cong 4[\alpha'_4(8) + 4f'_4(8)T_4(8)] = 4T_1(8)$$

Cioè

$$T_1(32) \cong 4T_1(8)$$

e

$$S_p = \frac{T_1(32)}{T_4(32)} \cong \frac{4T_1(8)}{T_4(4 \cdot 8)} = SS_p$$

△

Se si rapporta lo speed-up scalato al numero di processori p si ottiene l'efficienza scalata

$$ES_p = \frac{SS_p(n)}{p} = \frac{pT_1(n)}{pT_p(pn)} = \frac{T_1(n)}{T_p(pn)}$$

Un Algoritmo si dice *scalabile* se l'efficienza scalata ES_p rimane costante al crescere del numero p dei processori e della dimensione N del problema.

Problema Fissati il numero di processori p_0 , la dimensione iniziale del problema n_0 e quindi $T_1(n_0)$, calcoliamo n_1 e il tempo $T_1(n_1)$, affinché nel passare da p_0 a p_1 processori sia

$$E_{p_0}(n_0) = E_{p_1}(n_1) \quad (4.2)$$

Definizione Definiamo *isoefficienza* di un algoritmo la funzione

$$I : p, n \longrightarrow T_1(n, p)$$

tale che

$$E_{p_0}(n_0) = E_{p_1}(n_1)$$

Riprendendo la definizione di Overhead dell'esempio 13 data dalla (4.1):

$$O_h = pT_p - T_1 \implies T_p = \frac{O_h + T_1}{p}$$

si ricava che lo speed-up e l'efficienza per un problema di dimensione n , eseguito con p processori, sono

$$S_p(n) = \frac{T_1}{T_p} = \frac{T_1}{(O_h + T_1)/p} = \frac{pT_1}{O_h + T_1} = \frac{p}{\frac{O_h}{T_1} + 1}$$

$$E_p(n) = \frac{S_p(n)}{p} = \frac{1}{\frac{O_h}{T_1} + 1}$$

Volendo verificare la (4.2) otteniamo

$$\begin{cases} E_{p_0}(n_0) = \frac{1}{\frac{O_h(n_0, p_0)}{T_1(n_0)} + 1} \\ E_{p_1}(n_1) = \frac{1}{\frac{O_h(n_1, p_1)}{T_1(n_1)} + 1} \end{cases} \implies \frac{O_h(n_0, p_0)}{T_1(n_0)} = \frac{O_h(n_1, p_1)}{T_1(n_1)}$$

da cui

$$T_1(n_1) = T_1(n_0) \cdot \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)}$$

♣ Esempio 22

Consideriamo la somma di n numeri su p processori.

I tempi di esecuzione con 1 e con p processori sono rispettivamente

$$T_1 = n - 1 = O(n)$$

$$T_p = \frac{n}{p} - 1 + \log_2 p = O\left(\frac{n}{p} + \log_2 p\right)$$

da cui si ricava:

$$O_h = pT_p - T_1 = p \cdot \left(\frac{n}{p} + \log_2 p\right) - n = O(p \log_2 p)$$

Fissata la dimensione $n = n_0$ ed il numero di processori $p = p_0$ abbiamo:

$$O_h(p_0) = p_0 \log_2 p_0$$

$$T_1(n_0) = n_0$$

Nel passare poi da p_0 a p_1 processori (con $p_1 > p_0$) affinché l'efficienza sia costante si

deve avere

$$n_1 = n_0 \cdot \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0}$$

ovvero la dimensione n_1 deve aumentare, rispetto alla dimensione iniziale n_0 , di un fattore

$$\frac{p_1 \log_2 p_1}{p_0 \log_2 p_0}$$

△

♣ Esempio 23

Consideriamo l'algoritmo parallelo sviluppato nel paragrafo 3.4 che esegue il prodotto matrice per matrice

$$A \cdot B = C$$

secondo lo schema BMR (V Strategia).

Supponiamo che le matrici A, B e $C \in \mathbb{R}^{N \times N}$ e che i P processori siano distribuiti su di una griglia $q \times q$. Ogni processore ha dunque i blocchi A_{loc}, B_{loc} e C_{loc} di dimensione $nb \times nb$ con⁷

$$nb = \frac{N}{q} \quad (4.3)$$

Supponiamo inoltre che:

- t_{calc} sia tempo di esecuzione di un'operazione floating point;
- il tempo di invio di n dati f.p. da un processo ad un altro sia

$$\alpha + n \cdot t_{com}$$

dove α è il tempo di latenza e t_{com} è il tempo necessario a comunicare un dato f.p..

Per stimare l'efficienza, definita da

$$E = \frac{T_1}{P \cdot T_P} = \frac{N^3 t_{calc}}{P \cdot T_P}$$

valutiamo T_P procedendo secondo lo schema dell'algoritmo BMR.

⁷ Per semplicità di calcoli si è considerato N esattamente divisibile per q .

Per la spedizione in broadcast⁸ del blocco A_{loc} lungo le righe della griglia, da parte dei processori appartenenti alla diagonale principale, abbiamo:

$$T_{com}^1 = \log_2(q) \cdot (\alpha + nb^2 \cdot t_{com})$$

e per il primo prodotto parziale

$$A_{loc} \cdot B_{loc} = C_{loc}$$

abbiamo

$$T_{calc}^1 = nb^3 \cdot t_{calc}$$

Ad ogni passo $k = 1, \dots, q - 1$

- I processori della k -esima diagonale della griglia effettuano il broadcast di A_{loc} lungo le righe della griglia:

$$T_{com}^2 = \log_2(q) \cdot (\alpha + nb^2 \cdot t_{com})$$

- Nei sottocontesti colonna della griglia, ogni processore spedisce al proprio nord il blocco B_{loc} :

$$T_{com}^3 = \alpha + nb^2 \cdot t_{com}$$

- Ogni processore calcola il prodotto parziale

$$A_{loc} \cdot B_{loc} = C_{loc}$$

per cui

$$T_{calc}^2 = nb^3 \cdot t_{calc}$$

endfor

Sommando tutti i contributi relativi alle spedizioni ed alle operazioni f.p. effettuate otteniamo:

$$T_P = (\alpha + nb^2 \cdot t_{com})(q + q \cdot \log_2(q) - 1) + q \cdot nb^3 \cdot t_{calc}$$

⁸Per effettuare il broadcast ad albero di un dato f.p., con P processori, occorrono $\log_2(P)$ passi.

da cui

$$E = \frac{T_1}{P \cdot T_P} = \frac{N^3 t_{calc}}{P \cdot T_P} = \frac{N^3 t_{calc}}{(\alpha + nb^2 \cdot t_{com})(q + q \cdot \log_2(q) - 1) \cdot q^2 + q^3 \cdot nb^3 \cdot t_{calc}}$$

e sostituendo la (4.3) otteniamo

$$\begin{aligned} E &= \left[1 + \left(\frac{q}{N} \right)^3 \left(1 + \log_2(q) - \frac{1}{q} \right) \frac{\alpha}{t_{calc}} + \frac{q}{N} \left(1 + \log_2(q) - \frac{1}{q} \right) \frac{t_{com}}{t_{calc}} \right]^{-1} \\ &= \left[1 + \left(\frac{q}{N} \right)^3 (c_1 + c_2 \cdot \log_2(q)) \frac{\alpha}{t_{calc}} + \frac{q}{N} \cdot (c_1 + c_2 \cdot \log_2(q)) \frac{t_{com}}{t_{calc}} \right]^{-1} \end{aligned}$$

Si deduce quindi che $q = \sqrt{P}$ deve crescere con N per garantire che l'efficienza non tenda rapidamente a zero.

△

Bibliografia

- [1] M. Baker, R. Buyya - *Cluster Computing at a Glance* - <http://citeseer.nj.nec.com/article/baker99cluster.html>
- [2] M.W. Berry, S.T. Dumais, G.W. O'Brien - *Using Linear Algebra for Intelligent Information Retrieval* - SIAM Review, Vol. 37, No. 4, pp.573-595, December 1995
- [3] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R. C. Whaley - *ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory, Computers - Design Issues and Performance* - UT, CS-95-283, March 1995 - LAPACK Working Note 95, <http://www.netlib.org>
- [4] *Computer Architecture* - Computational Science Education Projects, <http://csep1.phy.ornl.gov/CSEP/CA/CA.html>.
- [5] S. Deerwester, S.T. Dumais, R.A. Harshman - *Indexing by latent semantic analysis* - Journal of the Society for Information Science, 41(6), 391-407, 1990
- [6] J. Demmel - *Lucidi del corso Applications of Parallel Computers* - Computer Science division Berkeley 1999 - http://cs.berkeley.edu/~demmel/cs267_Spr99/.
- [7] P. Messina, A. Murli *et al.* - *Some Perspectives on High Performance Mathematical Software* - High Performance Algorithms

and Software in Nonlinear Optimization, R. de Leone, A. Murli, P.M. Pardalos G. Toraldo, eds., Kluwer Academic Publishers, Boston , 1998.

- [8] J.J. Dongarra, H.W. Meuer, E. Strohmaier - *Top500 Supercomputer Sites* - <http://www.top500.org/> novembre 1999.
- [9] J.J. Dongarra, H.W. Meuer, H.D. Simon, E. Strohmaier - *The Marketplace of High Performance Computing* - Parallel Computing 1999.
- [10] J.J. Dongarra, H.W. Meuer, H.D. Simon, E. Strohmaier - *Changing Technologies of HPC* - Future Generation Computer Systems, Volume 12, Number 5, April 1997, p 461-474.
- [11] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson - *An extended Set of Fortran Basic Linear Algebra Subprograms* - ACM Transactions on Mathematical Software, 14(1):1-17, March 1988, <http://www.netlib.org/blas/index.html>
- [12] J.J. Dongarra, J. Du Croz, I. Duff, S. Hammarling - *A set of Level 3 Basic Linear Algebra Subprograms* - ACM Transactions on Mathematical Software, 16(1):1-17, 1990, <http://www.netlib.org/blas/index.html>
- [13] J.J. Dongarra, V. Eijkhout - *Numerical Linear Algebra Algorithms and Software* - Journal CAM (Numerical) Linear Algebra. Volume 31(4), 28 October 1999.
- [14] J.J. Dongarra, G.E. Fagg, R. Hempel, D.W. Walker - *Chapter in Wiley Encyclopedia of Electrical and Electronics Engineering* - <http://www.netlib.org/utk/people/JackDongarra/papers.htm>
- [15] J.J. Dongarra, A.R. Hinds - *Unrolling Loops in FORTRAN* - Software - Practice and Experience, Vol. 9, 219-226 (1979)

- [16] J.J. Dongarra, R. van de Geijn, D.W. Walker - *A Look at Scalable Dense Linear Algebra Libraries* - UT, CS-92-155, April, 1992- LAPACK Working Note 43, <http://www.netlib.org>
- [17] J.J. Dongarra, D.W. Walker - *The Design of Linear Algebra Libraries for High Performance Computers* - UT, CS-93-188, June 1993 - LAPACK Working Note 58, <http://www.netlib.org>
- [18] J.J. Dongarra, S. Hammarling, D.W. Walker - *Key Concepts for Parallel Out-of-Core LU Factorization* - UT, CS-95-292, May 1995 - LAPACK Working Note 100, <http://www.netlib.org>
- [19] <http://netlib.org/utk/people/JackDongarra/home.html>.
- [20] M.J. Flynn - *Some computer organisations and their effectiveness* - IEEE Trans. on Comp., C-21:948-960, 1972
- [21] L.Fosdick, E.Jessup - *An Overview of Scientific Computing*.
- [22] I. Foster, C. Kesselman, S. Tuecke - *The anatomy of the grid* - Intl. J. Supercomputer Applications, 15(3), 2001.
- [23] G. C. Fox, P. Messina - *Architetture per i supercalcolatori*.
- [24] E. Gallopoulos - *CSE: Content and Product* - IEEE Computational Science and Engineering, april-june 1997
- [25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam - *PVM: a users'guide and tutorial for networked parallel computing* - Scientific and engineering computation series, MIT Press, Cambridge, MA, 1994.
- [26] G. Giunta, G. Laccetti, A. Murli - *Software matematico e nuove architetture* - Rivista di Informatica Vol. XXI n. 2 aprile-giugno 1991.
- [27] G. Golub, C. Van Loan - *Matrix Computations* - Second ed., Johns-Hopkins University Press, Baltimore, 1989.

- [28] J.L. Gustafson, - Re-evaluating Amadahl's Law - Communications of the ACM, 31, 532-533, 1988
- [29] J.L. Gustafson, G.R. Montry, R.E. Benner - Development of Parallel Methods for a 1024 Processor Hypercube - SIAM Journal of Scientific and Statistical Computing, 9, 609-638, 1988
- [30] Hamacher, Vranesic, Zaky - *Introduzione all'architettura dei calcolatori* - McGraw Hill 1997.
- [31] R.W. Hockney, C.R. Jessope - *Parallel Computers - Architecture, Programming and Algorithms* - Adam Hilger Ltd, Bristol, 1981.
- [32] *The IBM NUMA-Q enterprise server architecture* - http://www.sequent.com/whitepapers/numa_arch.html.
- [33] J. Laudon, D. Lenoski - *System overview of the SGI-Origin2000 Product line* - www.sgi.com.
- [34] C. Lawson, R. Hanson, D. Kincaid, F. Krogh - *Basic Linear Algebra Subprograms for Fortran usage* - ACM Trans. Math. Softw., 5:308-323, 1979
- [35] D. Marshall - Programming in C Unix System Calls and Subroutines using C - <http://www.cs.cf.ac.uk/Dave/C/CE.html>
- [36] P. Messina - *High Performance Computers: The Next Generation (Part I)* - Computers in Physics, Vol. 11, n. 5 Sep/Oct 1997.
- [37] P. Messina - *High Performance Computers: The Next Generation (Part II)* - Computers in Physics, Vol. 11, n.6 Sep/Oct 1997.
- [38] G.E. Moore - *Cramming more components onto integrated circuits* - Electronics, Volume 38, Number 8, April 19, 1965

- [39] *MPI 1.1 - A Message Passing Interface Standard* - <http://www-unix.mcs.anl.gov/mpi/>.
- [40] A. Murli - *Lucidi delle lezioni del corso di Teoria e applicazione delle macchine calcolatrici* - Universita' degli Studi di Napoli "Federico II" Dipartimento di Matematica e Applicazioni "R. Caccioppoli" 1997.
- [41] A. Murli - *Lucidi delle lezioni del corso di Calcolo Parallelo e Distribuito, a.a. 200-2001* - Universita' degli Studi di Napoli "Federico II".
- [42] A. Murli, P.D'Ambra, L.D'Amore - *Parallel Computation and problem solving methodologies: a view from some experience* -in Recent Trends in Numerical Analysis, in Advances in Computation: Theory and Practice, Edito da D. Trigiante, 2000.
- [43] M.S. Paterson, L.J. Stockmeyer - *On the number of nonscalar multiplications necessary to evaluate polynomials* - SIAM J. Comp. 2,60-66, 1973.
- [44] P.C. Patton - *Performance Limits for Parallel Processor* - in "Parallel Supercomputing: Methods, Algorithms and Application", G.F. Carey ed., John Wiley & Sons, 1989
- [45] *Performance of the Cray T3E Multiprocessor* - <http://www.cray.com/products/systems/crayt3e/paper1.html>.
- [46] C.J.C. Schauble - *The Connection Machine An Introduction* - HPSC Group, Department of Computer Science, University of Colorado, Boulder 1993
- [47] V. Strassen - *Gaussian elimination is not optimal* - Numerische Mathematik 13, 354-356, 1969.

- [48] A. S. Tanenbaum - *Architettura del computer: un approccio strutturale*, Ed. Jackson libri, 1996, terza edizione.
- [49] C.W. Ueberhuber - *Numerical Computation 1* - Springer, 1997
- [50] W. Ware - *The ultimate computer* - IEEE Spectrum, March 1983
- [51] R.C. Whaley. J.J. Dongarra - *Automatically Tuned Linear Algebra Software* - SC '89 Proceedings (electronic Publication), IEEE Publication, 1998, <http://netlib.org/utk/people/JackDongarra/papers.html>.