



Università degli Studi di Bergamo

SCUOLA DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Informatica
Classe n. L-8 Ingegneria dell'Informazione (D.M. 270/04)

Introduzione al calcolo parallelo in MATLAB[®]

Candidato:

Thomas Fabbris

Matricola 1086063

Relatore:

Chiar.mo Prof. Fabio Previdi

Anno Accademico 2024–2025

Indice

Introduzione	1
1 Calcolo parallelo: sfida o opportunità?	3
1.1 Introduzione al calcolo parallelo	3
1.2 Le cause a supporto del parallelismo	5
1.2.1 Alcune applicazioni del calcolo parallelo	6
1.2.2 La barriera dell'energia	6
1.3 Le sfide nella progettazione di software parallelo	8
1.3.1 La legge di Amdahl	9
1.3.2 Verso i problemi «massicciamente paralleli»	11
Bibliografia	13

Introduzione

I primi progettisti di calcolatori, negli anni Cinquanta del XIX secolo, ebbero l'intuizione di interconnettere una moltitudine di calcolatori tradizionali, al fine di ottenere un sistema di elaborazione sempre più potente.

Questo sogno primordiale portò alla nascita dei *cluster* di elaboratori trent'anni dopo e allo sviluppo delle architetture *multicore* a partire dall'inizio del 2000.

Oggi la maggior parte delle applicazioni in ambito scientifico, tra cui quelle impiegate nella risoluzione di problemi di analisi numerica su larga scala, possono funzionare solo disponendo di sistemi di calcolo in grado di fornire una capacità di elaborazione molto elevata.

Nel capitolo 1, ci concentreremo sul concetto di calcolo parallelo e sulle principali sfide da affrontare durante la scrittura di software eseguito su più processori simultaneamente, tra cui spicca una crescita delle prestazioni non proporzionale al miglioramento apportato al sistema di elaborazione, un risultato espresso quantitativamente dalla legge di Ahmdal.

Nel corso del capitolo 2, analizzeremo i principali costrutti di programmazione parallela messi a disposizione dall'ambiente di calcolo numerico e programmazione MATLAB[®], nonché le scelte di progettazione fondamentali che hanno influenzato le attuali caratteristiche del linguaggio dedicate alla scrittura di programmi a esecuzione parallela.

Nel capitolo 3, forniremo un'illustrazione formale del metodo di Jacobi, un metodo iterativo dell'analisi numerica per la risoluzione approssimata di sistemi di equazioni lineari, dopo aver introdotto alcune nozioni di algebra lineare la cui conoscenza è necessaria per un'adeguato sviluppo dell'argomento.

Successivamente, proporranno un'implementazione parallela dell'algoritmo dietro al metodo di Jacobi, sfruttando le potenzialità fornite dall'impiego degli *array* globali per aumentare il livello di astrazione del programma a elaborazione parallela.

Infine, ci occuperemo dell'analisi dei risultati ottenuti dall'esecuzione dell'algoritmo su problemi di grandi dimensioni.

Capitolo 1

Calcolo parallelo: sfida o opportunità?

L'obiettivo di questo capitolo è esibire una definizione puntuale di calcolo parallelo, un termine spesso impiegato nel mondo del HPC (*High Performance Computing*) per riferirsi all'uso simultaneo di molteplici risorse di calcolo, consentendo la risoluzione di problemi a elevata intensità computazionale in tempi ragionevolmente brevi.

In seguito, investigheremo le cause che hanno portato alla nascita del parallelismo e descriveremo le principali difficoltà incontrate dai programmatori durante l'implementazione di programmi ad esecuzione parallela.

1.1 Introduzione al calcolo parallelo

L'idea alla base del calcolo parallelo è che gli utenti di un qualsiasi sistema di elaborazione possono avere a disposizione tanti processori quanti ne desiderano, per poi interconnetterli a formare un sistema multiprocessore, le cui prestazioni sono, con buona approssimazione, proporzionali al numero di processori impiegati.

La sostituzione di un singolo processore caratterizzato da un'elevata capacità di calcolo, tipicamente presente nelle architetture dei sistemi *mainframe*, con un insieme di processori più efficienti dal punto di vista energetico permette di raggiungere migliori prestazioni per unità di energia, a condizione che i programmi eseguiti siano stati appositamente progettati per lavorare su hardware parallelo; approfondiremo questi aspetti nel paragrafo 1.2.

Una tendenza introdotta da IBM nel 2001 nell'ambito della progettazione di sistemi paralleli [5] è il raggruppamento di diverse unità di calcolo all'interno di una singola CPU (*Central Processing Unit*); per evitare ambiguità nei termini impiegati, i processori montati su un singolo *chip* di silicio vengono chiamati *core*.

Il microprocessore *multicore* risultante appare al sistema operativo in esecuzione sull'elaboratore come l'insieme di N processori, ciascuno dotato di un set di registri e di una memoria *cache* dedicati; solitamente i microprocessori *multicore* sono impiegati in sistemi a memoria condivisa, in cui i *core* condividono lo stesso spazio di indirizzamento fisico.

Il funzionamento di questa categoria di sistemi multiprocessore si basa sul parallelismo a livello di attività (o a livello di processo): più processori sono impiegati per svolgere diverse attività simultaneamente e ciascuna attività corrisponde a un'applicazione a singolo *thread*.

In generale, ogni *thread* esegue un'operazione ben definita e *thread* differenti possono agire sugli stessi dati o su insiemi di dati diversi, garantendo un elevato *throughput* per attività tra loro indipendenti.

D'altro canto, tutte le applicazioni che richiedono un utilizzo intensivo di risorse di calcolo, diffuse non solamente in ambito scientifico, possono essere eseguite solo su *cluster* di elaboratori, una tipologia di sistemi multiprocessore che si differenzia dai microprocessori *multicore* per il fatto di essere costituita da un insieme di calcolatori completi, chiamati nodi, collegati tra loro per mezzo di una rete LAN (*Local Area Network*).

In ogni caso, il funzionamento di un sistema di elaborazione parallelo si basa sull'uso congiunto di processori distinti.

Per sfruttare al meglio le potenzialità offerte dai *cluster* di elaboratori, i programmatori di applicazioni devono sviluppare programmi a esecuzione parallela efficienti e scalabili a seconda del numero di processori disponibili durante l'esecuzione; risulta necessario applicare un parallelismo a livello di dati, che prevede la distribuzione dell'insieme di dati in input tra le CPU del *cluster*, per poi lanciare in esecuzione la medesima operazione, su sottoinsiemi distinti di dati, su ogni processore.

Una tipica operazione parallelizzabile a livello di dati è la somma vettoriale perché le componenti del vettore risultante sono ottenute semplicemente sommando le componenti omologhe dei due vettori di partenza.

Possiamo intuire fin da subito che una condizione necessaria per la parallelizzazione di un qualsiasi algoritmo è l'indipendenza tra le operazioni eseguite ad un certo

passo dell'esecuzione.

Per esempio, supponiamo di dover sommare due vettori di numeri reali di dimensione N avvalendoci di un sistema *dual-core*, ossia di un sistema di elaborazione dotato di un microprocessore che contiene al suo interno due *core*.

Un'approccio di risoluzione prevederebbe l'avvio di un thread separato su ogni *core*, specializzato nella somma di due componenti corrispondenti dei vettori operandi; attraverso un'attenta distribuzione dei dati in input, il *thread* in esecuzione sul primo *core* sommerebbe le componenti da 1 a $\lceil \frac{N}{2} \rceil$ dei vettori di partenza e, contemporaneamente, il secondo *core* si occuperebbe della somma delle componenti da $\lceil \frac{N}{2} \rceil + 1$ a N .

A dire il vero, la rigida distinzione proposta tra parallelismo a livello di attività e parallelismo a livello di dati non trova un diretto riscontro nella realtà, in quanto sono comuni programmi applicativi che sfruttano entrambi gli approcci al fine di massimizzare le prestazioni.

Cogliamo l'occasione per precisare la terminologia, in parte già impiegata, per descrivere la componente hardware e la componente software di un calcolatore: l'hardware, riferendoci con questo termine esclusivamente al processore, può essere seriale, come nel caso di un processore *single core*, o parallelo, come nel caso di un processore *multicore*, mentre il software viene detto sequenziale o concorrente, a seconda della presenza di processi la cui esecuzione viene influenzata dagli altri processi presenti nel sistema.

Naturalmente, un programma concorrente può essere eseguito sia su hardware seriale che su hardware parallelo, con ovvie differenze in termini di prestazioni.

Infine, con il termine programma a esecuzione parallela, o semplicemente software parallelo, indichiamo un programma, sequenziale o concorrente, eseguito su hardware parallelo.

1.2 Le cause a supporto del parallelismo

L'attenzione riservata all'elaborazione parallela da parte della comunità scientifica risale al 1957, anno in cui la Compagnie des Machines Bull ¹ annunciò Gamma 60, un computer *mainframe* equipaggiato con la prima architettura della storia con supporto diretto al parallelismo, mentre l'anno successivo, i ricercatori di IBM John Cocke e Daniel Slotnick aprirono per la prima volta alla possibilità di integrare il *parallel computing* nell'esecuzione di simulazioni numeriche [6].

¹l'odierna Bull SAS, con sede a Les-Clayes-sous-Bois

1.2.1 Alcune applicazioni del calcolo parallelo

Oggi sopravvivono in ambito scientifico alcune applicazioni che possono essere eseguite solo su *cluster* di elaboratori oppure che richiedono lo sviluppo di speciali architetture parallele, individuate dall'acronimo DSA (*Domain Specific Architecture*), per via delle loro caratteristiche *compute-intensive*.

Esempi di settori che hanno beneficiato dello sviluppo di architetture innovative per il calcolo parallelo sono la bioinformatica, l'elaborazione di immagini e video e il settore aerospaziale, che si è potuto affidare a simulazioni numeriche sempre più accurate.

La rivoluzione introdotta dal calcolo parallelo non si limita esclusivamente al campo scientifico: un dominio applicativo che negli ultimi due decenni ha registrato uno sviluppo senza precedenti è l'intelligenza artificiale (AI, *Artificial Intelligence*) e, in particolare, l'addestramento di modelli di AI mediante tecniche di *machine learning*. I successi ottenuti in questo settore, tangibili in contesti applicativi distanti tra loro come il riconoscimento di oggetti o l'industria della traduzione, non sarebbero stati fattibili se non supportati da sistemi di calcolo sufficientemente potenti in grado di eseguire le operazioni aritmetiche richieste dall'allenamento di modelli sempre più complessi.

Come ulteriori evidenze di questo processo, possiamo citare i calcolatori dei moderni centri di calcolo, i cosiddetti *Warehouse Scale Computer* (WSC), che costituiscono l'infrastruttura di erogazione dei moderni servizi Internet utilizzati ogni giorno da milioni di utenti, di cui fanno parte i motori di ricerca, i *social network* e i servizi di commercio elettronico.

Inoltre, l'avvento del *cloud computing*, ovvero l'offerta via Internet di risorse di elaborazione «*as a service*», ha recentemente consentito l'accesso ai WSC a chiunque sia dotato di una carta di credito.

1.2.2 La barriera dell'energia

Il fattore fondamentale dietro all'adozione di massa delle architetture multiprocessore è la riduzione del consumo di energia elettrica offerta dai sistemi di calcolo paralleli; infatti, l'alimentazione e il raffreddamento delle centinaia di server presenti in un centro di calcolo moderno costituiscono una componente di costo non trascurabile, che risente marginalmente della disponibilità di sistemi di raffreddamento dei microprocessori atti a dissipare una grande quantità di energia.

Il consumo di energia elettrica dei microprocessori viene misurato in Joule (J) ed è quasi interamente rappresentato dalla dissipazione di energia dinamica da parte dei transistori CMOS (*Complementary Metal Oxide Semiconductor*), essendo quest'ultima la tecnologia dominante nella realizzazione dei moderni circuiti integrati.

Un transistor assorbe prevalentemente energia elettrica durante la commutazione alto-basso-alto del suo stato di uscita, secondo la formula

$$E = C_L \cdot V^{+2}$$

dove E rappresenta l'energia dissipata nelle due transizioni, V^+ la tensione di alimentazione e C_L la capacità di carico del transistor.

La potenza dissipata P_D , assumendo che la frequenza di commutazione dello stato del transistor sia pari a f , è quindi data da

$$P_D = f \cdot E = f \cdot C_L \cdot V^{+2} \propto f_C$$

dove f_C è la frequenza di *clock* del circuito, esprimibile in funzione di f .

In passato, i progettisti di calcolatori hanno tentato di contenere l'assorbimento di energia dei microprocessori riducendo la tensione di alimentazione V^+ di circa il 15% ad ogni nuova generazione di CPU, fino al raggiungimento del limite inferiore di 1 V.

Al contempo, la diminuzione della tensione di alimentazione ha favorito la crescita delle correnti di dispersioni interne al transistor, tanto che nel 2008 circa il 40% della potenza assorbita da un transistor era imputabile alle correnti di dispersione: ci eravamo imbattuti in una vera e propria «barriera dell'energia».

In figura 1.1 possiamo notare come fino alla prima metà degli anni Ottanta del secolo scorso la crescita annua delle prestazioni dei processori si attestava al 25%, per poi passare al 52% grazie al contributo di importanti innovazioni nella progettazione e nell'organizzazione dei calcolatori; dal 2002 in avanti, si sta registrando una crescita delle prestazioni meno evidente, pari al 3.5% annuo, a causa del raggiungimento dei limiti relativi alla potenza assorbita.

La presenza di queste limitazioni tecnologiche ha certamente accelerato la ricerca di nuove architetture per microprocessori, culminata con lo sviluppo del primo processore *multicore*, IBM Power4, nel 2001 e la successiva introduzione delle prime CPU di questo tipo destinate al mercato *consumer* nel 2006 da parte di Intel e AMD.

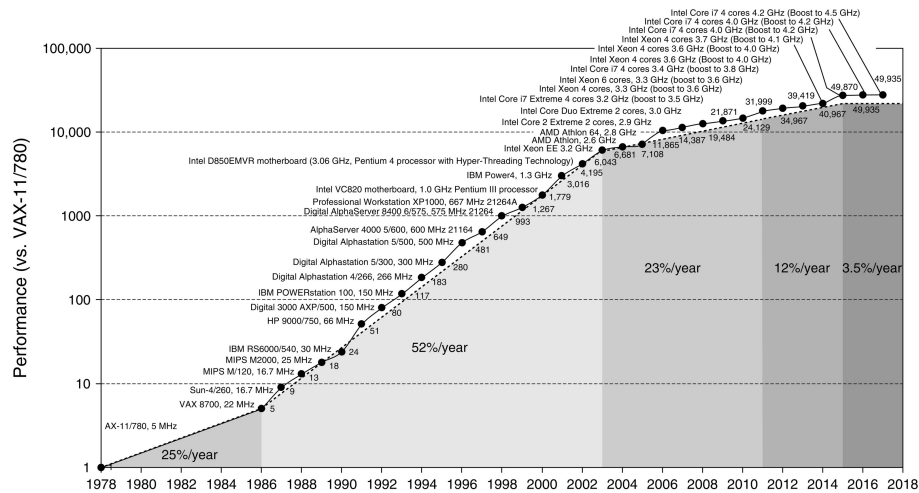


Figura 1.1: Crescita nelle prestazioni dei processori dal 1978 al 2018; il grafico riporta le prestazioni dei processori, paragonandoli al VAX11/780, mediante l'esecuzione dei benchmark SPECint (Da J.L. Hennessy, D.A. Patterson, *Computer Architecture: A quantitative Approach*. Ed. 6. Waltham, MA:Elsevier, 2017)

In futuro, il miglioramento delle prestazioni dei microprocessori sarà verosimilmente determinato dall'aumento del numero di *core* montati su un singolo *chip* piuttosto che dalla crescita della frequenza di *clock* dei singoli processori.

1.3 Le sfide nella progettazione di software parallelo

Le motivazioni che rendono lo sviluppo di programmi a esecuzione parallela una vera e propria sfida per i programmatori di applicazioni sono molteplici e appartengono a diverse aree di intervento.

Innanzitutto, una caratteristica contraddistintiva del software parallelo è la scalabilità, ovvero la capacità del sistema software di incrementare le proprie prestazioni in funzione della potenza di calcolo richiesta in un preciso istante e di adeguare di conseguenza le risorse computazionali impiegate [2].

Da un lato la scalabilità, sfruttando la sinergia tra hardware e software di ogni sistema informatico, consente di ottenere sistemi multiprocessore tolleranti ai guasti e a elevata disponibilità, ma dall'altro richiede che il software venga progettato in maniera tale da sfruttare al meglio i diversi processori e che il codice sorgente sia riscritto a ogni incremento del numero di unità di elaborazione.

Chiaramente, la difficoltà incontrata nella realizzazione di programmi a esecuzione parallela va di pari passo al numero di processori presenti nel sistema.

La profonda ristrutturazione richiesta durante il ciclo di vita di tutti i programmi

a elaborazione parallela, radicata sia nella fase di *design* che durante la fase di manutenzione, è necessaria per il raggiungimento delle massime prestazioni, nonostante rallenti l'introduzione di nuove funzionalità.

A questo proposito, la programmazione parallela è per definizione ad alte prestazioni ed esige una velocità di esecuzione elevata; in caso contrario, sarebbe sufficiente disporre di programmi sequenziali eseguiti su sistemi monoprocesso, la cui programmazione è di gran lunga più agevole.

Come abbiamo già accennato nel paragrafo 1.1, le attività, chiamate anche *task* in cui è ripartito un *job* svolto da un programma a esecuzione parallela devono essere indipendenti le une dalle altre per poter essere eseguite su più processori simultaneamente.

Di conseguenza, è indicato suddividere l'applicazione in maniera tale che ogni processore compia circa lo stesso carico di lavoro in intervalli di tempo di durata comparabile; se un processore impiegasse un tempo maggiore per terminare le *task* a esso assegnate rispetto agli altri, farebbe svanire i benefici prestazionali portati dall'impiego di un sistema multiprocessore.

Oltre allo *scheduling* delle attività e al bilanciamento del carico di lavoro tra i processori, altri problemi derivano dalla presenza di *overhead* di comunicazione e di sincronizzazione tra le diverse unità di lavoro, qualora si rendesse necessaria la cooperazione tra le *task* per portare a termine il *job*.

Una regola generale per gestire queste problematiche è evitare di sprecare la maggior parte del tempo di esecuzione di un software parallelo per la comunicazione e la sincronizzazione tra i processori, idealmente dedicando un lasso di tempo irrilevante a questi due aspetti.

Un'ulteriore sfida da affrontare durante la progettazione di software parallelo è descritta dalla legge di Amdahl, che limita il miglioramento prestazionale complessivamente ottenuto dall'ottimizzazione di una singola parte di un sistema di elaborazione.

1.3.1 La legge di Amdahl

La legge di Amdahl, esposta per la prima volta dall'ingegnere statunitense Gene Myron Amdahl al AFIPS *Spring Joint Computer Conference* del 1967, è una legge empirica, considerata un'espressione quantitativa della legge dei rendimenti decrescenti, enunciata dall'economista classico David Ricardo nel XIX secolo.

Amdahl utilizza il termine *enhancement* per indicare un qualsiasi miglioramento introdotto in un sistema di elaborazione.

Il beneficio, in termini di prestazioni, attribuibile a esso dipende da due fattori: la frazione del tempo di esecuzione iniziale, che diminuisce a seguito dell'*enhancement*, e dall'entità del miglioramento.

In aggiunta, il concetto di incremento di velocità, detto anche *speedup*, ricopre un ruolo centrale nell'intero impianto teorico.

Dato un generico programma e un calcolatore a cui viene apportato un *enhancement*, denominato calcolatore migliorato, lo *speedup* è definito come il fattore secondo il quale il calcolatore migliorato riesce ad eseguire più velocemente il programma rispetto al calcolatore originale, calcolato secondo la seguente formula

$$Speedup = \frac{Performance\ programma\ con\ miglioramento}{Performance\ programma\ senza\ miglioramento}$$

sotto l'ipotesi in cui le prestazioni del calcolatore migliorato siano effettivamente rilevabili attraverso le metriche prestazionali scelte.

A questo punto, il tempo di esecuzione per il calcolatore migliorato, denotato T_{dopo} , può essere espresso come somma del tempo di esecuzione modificato dal miglioramento $T_{modificato}$ e di quello non interessato dal cambiamento $T_{nonModificato}$.

$$T_{dopo} = \frac{T_{modificato}}{Entità\ miglioramento} + T_{nonModificato}$$

A questo punto, possiamo riformulare la legge di Amdahl in termini di incremento di velocità rispetto al tempo di esecuzione iniziale

$$Speedup = \frac{T_{dopo}}{T_{prima} - T_{dopo}} + \frac{T_{dopo}}{Entità\ miglioramento}$$

con T_{prima} tempo di esecuzione prima del miglioramento.

La formula precedente viene comunemente riscritta ponendo pari a 1 il tempo di esecuzione prima del miglioramento e supponendo che il tempo modificato dal miglioramento sia espresso in termini del tempo originario di esecuzione attraverso una frazione, ottenendo

$$Speedup = \frac{1}{1 - Frazione\ tempo\ modificato + \frac{Frazione\ tempo\ modificato}{Entità\ miglioramento}}$$

Come é intuibile dalla spiegazione precedente, la legge di Amdahl puó essere applicata alla stima quantitativa del miglioramento delle prestazioni solo se il tempo in cui viene sfruttata una certa funzione all'interno del sistema é noto, cosí come il potenziale *speedup*.

Un adattamento della legge di Amdahl al calcolo parallelo é il seguente

«Anche le piú piccole parti di un programma devono essere rese parallele se si vuole eseguire il programma in modo efficiente su un sistema multiprocessore»

1.3.2 Verso i problemi «massicciamente paralleli»

Nel contesto del calcolo parallelo, vengono usati termini specifici per contraddistinguere classi di problemi da risolvere.

Per esempio, un problema «*embarrassingly parallel*» é un problema che richiede un minimo sforzo per essere suddiviso in un insieme di *task*, a causa del loro debole accoppiamento [1], mentre il termine «*massively parallel*», in italiano «massicciamente parallelo», descrive i problemi di grandi dimensioni suddivisibili in un numero elevato di *task*, eseguite simultaneamente su migliaia di processori.

Un problema «*embarrassingly parallel*» di interesse per l'analisi numerica é il calcolo approssimato di integrali definiti per funzioni di una o piú variabili; diversamente, il processo di addestramento di modelli complessi di *machine learning*, come le *deep neural network*, richiede l'esecuzione di migliaia di operazioni aritmetiche, inserendolo di diritto all'interno della classe dei problemi «massicciamente paralleli».

Esempio 1 (Analisi prestazionale di un problema di grandi dimensioni)

Supponiamo di sommare trenta variabili scalari e due matrici quadrate di dimensione 3000×3000 servendoci dapprima di un tradizionale sistema monoprocesso e, successivamente, di un sistema multiprocessore con 30 CPU che supporta la parallelizzazione della somma tra matrici.

Vogliamo analizzare la variazione delle prestazioni del problema quando:

- a) il numero di processori del sistema multiprocessore passa da 30 a 120;
- b) le matrici diventano di dimensione 6000×6000 .

La tabella 1.1 riporta la frazione dello *speedup* potenziale raggiunta nei quattro possibili scenari di esecuzione.

L'esempio 1 evidenzia un problema fondamentale del calcolo parallelo: aumenta-

	Num. processori		
Dim. matrice		30	120
3000 x 3000		0,7770	0,4727
6000 x 6000		0,8739	0,6375

Tabella 1.1: Frazione dello *speedup* potenziale nei casi proposti nell'esempio 1

re la velocità di esecuzione di un programma a esecuzione parallela su un sistema multiprocessore mantenendo fisse le dimensioni del problema è più difficile rispetto che migliorare le prestazioni incrementando le dimensioni del problema proporzionalmente al numero di processori.

Questo particolare comportamento porta alla definizione dei concetti di scalabilità forte e di scalabilità debole.

La prima si riferisce all'incremento della velocità di esecuzione che si ottiene in un sistema multiprocessore senza aumentare la dimensione del problema da risolvere, mentre la seconda descrive l'incremento di velocità ottenuto quando la dimensione del problema viene aumentata proporzionalmente al numero di processori impiegati nel sistema.

Possiamo giustificare il comportamento descritto in precedenza prendendo come esempio un qualsiasi sistema multiprocessore e un programma ad esecuzione parallela.

Denotiamo con P il numero di processori presenti nel sistema e con M la dimensione del problema risolto dal programma (per semplicità supponiamo che M sia pari alla dimensione dello spazio da allocare in memoria centrale per la risoluzione del problema).

Sotto queste ipotesi, ogni processore possiederà uno spazio di memoria dedicato pari a M nel caso della scalabilità debole e pari a $\frac{M}{P}$ nel caso della scalabilità forte.

Potremmo essere erroneamente indotti a pensare che la scalabilità debole sia più facilmente ottenibile rispetto alla scalabilità forte, data la maggiore quantità di memoria disponibile per ogni processore, ma a seconda del contesto applicativo considerato possiamo individuare validi motivi a supporto di ciascuno dei due approcci; vale la pena notare che problemi di grandi dimensioni solitamente richiedono molti dati in input, rendendo la scalabilità debole più agevole da raggiungere.

Bibliografia

- [1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 1st ed. Elsevier, 2011.
- [2] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski, “Scale-up x Scale-out: A Case Study using Nutch/Lucene,” in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.
- [3] D. A. Patterson and J. L. Hennessy, *Struttura e progetto dei calcolatori*, 5th ed. Bologna: Zanichelli, 2022, A cura di Alberto Borghese.
- [4] P. Spirito, *Elettronica Digitale*, 3rd ed. McGraw-Hill Education, 2021.
- [5] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, “POWER4 system microarchitecture,” *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, dec 2001.
- [6] G. V. Wilson, “The History of the Development of Parallel Computing,” Virginia Tech/Norfolk State University, Interactive Learning with a Digital Library in Computer Science, 1994.