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

I primi progettisti di calcolatori, negli anni Cinquanta del secolo scorso, hanno avuto sin da subito l'intuizione di interconnettere una moltitudine di calcolatori tradizionali, al fine di ottenere un sistema di elaborazione sempre più potente.
Questo sogno primordiale ha portato 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 di questo lavoro di tesi, 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 ad esecuzione parallela.

Nel capitolo 3, descriveremo dal punto di vista formale il 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, proporremo un'implementazione parallela dell'algoritmo dietro al metodo di Jacobi, sfruttando le potenzialità fornite dagli *array* globali, utili per aumentare il livello di astrazione di un programma parallelo.

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

Capitolo 1

Calcolo parallelo: sfida o opportunitá?

L'obiettivo di questo capitolo é dare una definizione precisa di calcolo parallelo, un termine spesso impiegato nel mondo del supercalcolo per riferirsi all'uso simultaneo di molteplici risorse di calcolo per la risoluzione di problemi ad elevata intensità computazionale in tempi ragionevolmente brevi.

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

1.1 Introduzione al calcolo parallelo

L'idea alla base del calcolo parallelo é 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 capacitá di calcolo, tipicamente presente nelle architetture dei sistemi *mainframe*, con un insieme di processori piú efficienti dal punto di vista energetico permette di raggiungere migliori prestazioni per unitá di energia, a condizione che i programmi eseguiti siano appositamente progettati per essere eseguiti su hardware parallelo; approfondiremo questi aspetti nei paragrafi 1.2 e 1.3.

Una tendenza introdotta da IBM nel 2001 nell'ambito della progettazione di sistemi paralleli [5] è il raggruppamento di diverse unitá di calcolo all'interno di un singolo circuito integrato; in questo contesto, 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 attivitá simultaneamente e ciascuna attivitá 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 attivitá tra loro indipendenti.

D'altro canto, tutte le applicazioni che richiedono un utilizzo intensivo di risorse computazionali, diffuse non piú 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 potenzialitá 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 dei dati in input tra le CPU del *cluster*, per poi far eseguire la medesima operazione, su sottoinsiemi distinti di dati, a ogni processore.

Una tipica operazione parallelizzabile a livello di dati è la somma vettoriale perché le componenti del vettore risultante vengono ottenute sommando solamente 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 dimensio-

ne N avvalendoci di un sistema *dual-core*, ossia di un sistema di elaborazione dotato di un microprocessore che contiene al suo interno due *core*.
Sarebbe conveniente avviare un thread separato su ogni *core*, specializzato nella somma di due componenti omologhe 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 al contempo il secondo *core* si occuperebbe della somma delle componenti da $\lceil \frac{N}{2} \rceil + 1$ a N.

In realtà, la rigida distinzione proposta tra parallelismo a livello di attivitá e parallelismo a livello di dati non trova un diretto riscontro nella realtá, 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 giá utilizzata, per descrivere la componente hardware e la componente software di un generico calcolatore: l'hardware, riferendoci con questo termine esclusivamente al processore, puó 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 cooperanti la cui esecuzione viene influenzata dagli altri processi presenti nel sistema.
Naturalmente, un programma concorrente puó essere eseguito sia su hardware seriale che su hardware parallelo.
Infine, con il termine programma a esecuzione parallela, o piú 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 comunitá scientifica risale al 1957, anno in cui la Compagnie des Machines Bull (l'odierna Bull SAS) ha annunciato Gamma 60, un computer *mainframe* equipaggiato con la prima architettura della storia in grado di offrire un supporto diretto al parallelismo, mentre l'anno successivo, i ricercatori IBM John Cocke e Daniel Slotnick hanno per la prima volta aperto alla possibilitá di impiegare il *parallel computing* per l'esecuzione di simulazioni numeriche [6].

1.2.1 Alcune applicazioni del calcolo parallelo

Oggi permangono applicazioni in ambito scientifico che possono essere eseguite solo su *cluster* di elaboratori oppure che richiedono lo sviluppo di architetture specifiche di dominio (DSA, *Domain Specific Architecture*), considerate le 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 ha potuto contare su simulazioni numeriche sempre piú 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 é l'intelligenza artificiale (AI, *Artificial Intelligence*) e, in particolare, l'addestramento di modelli di AI mediante tecniche di *Machine Learning*.
I successi e le evoluzioni ottenuti in questo settore, e tangibili in diversi campi di applicazione come il riconoscimento di oggetti o l'industria della traduzione, non sarebbero stati fattibili se non supportati da sistemi di calcolo sufficientemente potenti e in grado di eseguire le operazioni aritmetiche richieste dallo svolgimento di compiti sempre piú complessi.

Come ulteriore esempio, possiamo citare i calcolatori dei moderni centri di calcolo, chiamati *Warehouse Scale Computer* (WSC), che costituiscono l'infrastruttura di erogazione dei moderni servizi Internet utilizzati ogni giorno da milioni di utenti, come i motori di ricerca, i *social network* e i servizi di *e-commerce*.
Inoltre, la rivoluzione del *cloud computing*, ovvero l'offerta via Internet di risorse di elaborazione «*as a service*», consente 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 é 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 solo marginalmente della disponibilitá di sistemi di raffreddamento dei microprocessori adatti a dissipare una grande quantitá di energia.

Il consumo di energia elettrica dei microprocessori viene misurato in Joule (J) ed é quasi interamente rappresentato dalla dissipazione di energia dinamica da parte dei transistori CMOS (*Complementary Metal Oxide Semiconductor*), essendo la tecnologia dominante impiegata nella realizzazione dei moderni circuiti integrati. Un transistore assorbe prevalentemente energia elettrica durante la commutazione alto-basso-alto del suo stato di uscita, secondo la formula

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

dove E rappresenta l'energia dissipata nelle due transizioni, V^+ la tensione di alimentazione e C_L la capacitá di carico del transistore.
La potenza dissipata P_D, assumendo che la frequenza di commutazione dello stato del transistore sia pari a f, é quindi data da

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

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

In passato, i progettisti di circuiti integrati hanno tentato di contenere l'assorbimento di energia da parte 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 transistore, tanto che nel 2008 circa il 40% della potenza assorbita da un transistore era imputabile alle correnti di dispersione; ci si é imbattuti in una vera e propria «barriera dell'energia».

In figura 1.1, possiamo notare come fino alla prima metá degli anni Ottanta del secolo scorso, la crescita annua delle prestazioni dei processori si attestava al 25%, per poi passare al 52% grazie a importanti innovazioni nella progettazione e nell'organizzazione dei calcolatori; infine dal 2002, si sta continuando a registrare una crescita delle prestazioni meno evidente, pari al 3.5% annuo, a causa del raggiungimento dei limiti relativi la potenza assorbita.
La presenza di queste limitazioni tecnologiche ha accelerato la ricerca di nuove architetture per microprocessori, culminata con lo sviluppo del primo processore *multicore*, IBM Power4, nel 2001 e il successivo lancio delle prime CPU *multicore* destinate al mercato *consumer*, nel 2006 da parte di Intel e AMD.

In futuro, il miglioramento delle prestazioni dei microprocessori sará verosimilmente

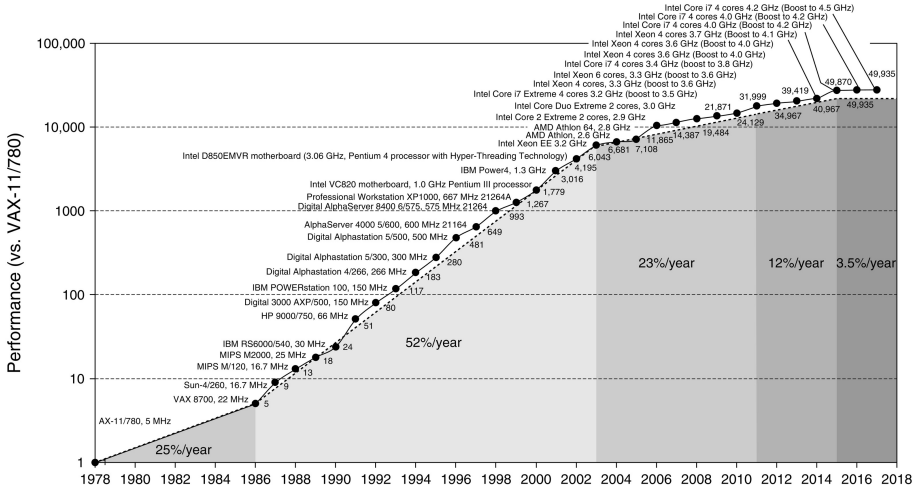

Figura 1.1: Crescita nelle prestazioni dei processori dal 1978 al 2018; il grafico riporta le prestazioni dei processori, paragonandoli al VAX11/780 mediante l'esecuzione dei *benchmark* SPECint *(Da J.L. Hennessy, D.A. Patterson, Computer Architecture: A quantitative Approach. Ed. 6. Waltham, MA:Elsevier,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

apportato 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 sono molteplici e appartengono a diverse aree di intervento.

Innanzitutto, una caratteristica contraddistintiva del software parallelo è la scalabilitá, definibile come la capacitá del sistema software di incrementare le proprie prestazioni in funzione della potenza di calcolo richiesta in un preciso istante, adeguando di conseguenza le risorse computazionali impiegate [2].
Da un lato la scalabilitá, sfruttando la stretta sinergia tra hardware e software di un sistema informatico, consente di ottenere sistemi multiprocessore tolleranti ai guasti e a elevata disponibilitá, 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 quasi completamente riscritto a ogni incremento del numero di *core*.
Chiaramente, la difficoltá di sviluppare programmi a esecuzione parallela corretti ed efficienti 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 esecuzione parallela, radicata sia nella fase di *design* che durante la fase di manu-

tenzione, é necessaria per ottenere le massime prestazioni da un sistema di calcolo parallelo.
A questo proposito, la programmazione parallela é per definizione ad alte prestazioni e richiede una velocitá di esecuzione molto elevata; in caso contrario, sarebbe sufficiente avere a disposizione programmi sequenziali eseguiti su sistemi monoprocessore, la cui programmazione é di gran lunga piú agevole.

Come abbiamo giá accennato nel paragrafo 1.1, le attivitá, chiamate anche *task* in cui é suddiviso un *job* svolto da un programma a esecuzione parallela devono essere indipendenti le une dalle altre per poter essere eseguite su piú processori simultaneamente.
Di conseguenza, é necessario suddividere l'applicazione in maniera tale che ogni processore compi circa lo stesso carico di lavoro nel medesimo intervallo di tempo; se un processore impiegasse un tempo maggiore per terminare le *task* a esso assegnate rispetto agli altri, i benefici prestazionali portati dall'impiego di un sistema multiprocessore svanirebbero rapidamente.

Oltre allo *scheduling* delle attivitá e al bilanciamento del carico di lavoro tra i processori, altri problemi derivano dalla presenza di *overhead* di comunicazione tra le diverse unitá di lavoro e di sincronizzazione, qualora si rendesse necessaria la cooperazione tra diverse *task* per portare a termine il loro compito.
Una regola generale per gestire queste problematiche é 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 é 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 dalla 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}, puó 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 velocitá 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 è il seguente

«Anche le più 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 diversi per contraddistinguere diverse classi di problemi secondo svariati criteri.
Per esempio, un problema «*embarrassingly parallel*» é un problema che richiede uno sforzo minimo 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 tutti quei problemi di grandi dimensioni suddivisibili in un numero enorme di *task* esegubili simultaneamente su migliaia di processori.

Un problema «*embarrassingly parallel*» di interesse nell'ambito dell'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 moltiplicazioni matriciali, inserendolo di diritto all'interno della classe dei problemi «massicciamente paralleli».

Esempio 1 (Analisi prestazionale di un problema «massicciamente parallelo»)**.** Supponiamo di moltiplicare trenta variabili scalari e due matrici quadrate di dimensione 3000×3000 servendoci prima di un tradizionale sistema monoprocessore e poi di un sistema multiprocessore con 30 CPU in grado di parallelizzare solo il prodotto tra matrici.
Vogliamo calcolare l'aumento di velocitá 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 lo *speedup* registrato negli scenari a) e b).

Dim. matrici \ Num. processori	**30**	**120**
3000 x 3000	0.55	0.22
6000 x 6000	0.82	0.51

Tabella 1.1: *Speedup* relativo al problema dell'esempio 1

L'esempio 1 rende evidente un problema fondamentale del calcolo parallelo: aumentare la velocitá di esecuzione di un programma a esecuzione parallela su un sistema multiprocessore mantenendo fisse le dimensioni del problema é piú 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 scalabilitá forte e di scalabilitá debole:

- dato un sistema multiprocessore e un problema da risolvere, definiamo scalabilità forte l'incremento di velocitá ottenuto nel multiprocessore senza aumentare la dimensione del problema;

- dato un sistema multiprocessore con $P > 1$ processori e un problema da risolvere, definiamo scalabilità debole l'incremento di velocit'øttenuto nel multiprocessore aumentando la dimensione del problema proporzionalmente a P.

Possiamo giustificare il comportamento descritto in precedenza tramite la seguente osservazione.

Consideriamo un qualsiasi sistema multiprocessore e un problema ad esecuzione parallela da risolvere; denotiamo P il numero di processori presenti nel sistema ed M la dimensione del problema (per semplicitá supponiamo che M sia pari alla dimensione dello spazio da allocare in memoria centrale per la risoluzione del problema). Sotto queste ipotesi, ogni processore possiederá uno spazio di memoria dedicato pari a M nel caso della scalabilitá debole e pari a $\frac{M}{P}$ nel caso della scalabilitá forte.

Potremmo essere erroneamente indotti a pensare che la scalabilitá debole sia piú facilmente ottenibile rispetto alla scalabilitá forte, 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, andando a favorire la scalabilitá debole in questo confronto.

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.