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 Novecento, ebbero l'intuizione di interconnettere una moltitudine di calcolatori tradizionali, al fine di ottenere un sistema di elaborazione sempre più potente.
Quel sogno primordiale portò alla nascita dei *cluster* di elaboratori trent'anni dopo e allo sviluppo delle architetture di microprocessore *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.
Successivamente, proporremo un'implementazione parallela dell'algoritmo codificato dal metodo di Jacobi, sfruttando le potenzialità fornite dall'impiego dagli *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 opportunitá?

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

In seguito, investigheremo le cause che portarono alla nascita del parallelismo e descriveremo le principali difficoltà incontrate dai programmatori di applicazioni durante l'implementazione di programmi a 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 di calcolo *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 è il raggruppamento di diverse unità di calcolo all'interno di una singola CPU (*Central Processing Unit*); per evitare ambiguità nei termini usati, 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 P 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 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 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, hanno bisogno di essere eseguite 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 di telecomunicazione.
In ogni caso, il funzionamento di un sistema di elaborazione parallela 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 di dati da processare tra le unità di lavoro del *cluster*, per poi lanciare in esecuzione la medesima operazione, con sottoinsiemi distinti di dati in ingresso, 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 vettori di partenza.
Possiamo intuire fin da subito che una condizione necessaria per la parallelizzazio-

ne 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 prevede 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

[1]l'odierna Bull SAS, con sede a Les-Clayes-sous-Bois in Francia.

Cocke e Daniel Slotnick aprirono per la prima volta alla possibilità di integrare il *parallel computing* nell'esecuzione di simulazioni numeriche [4].

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, raggruppate sotto l'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 piú accurate.

La rivoluzione introdotta dal calcolo parallelo non si limita esclusivamente al campo scientifico: un dominio applicativo che, negli ultimi due decenni, sta registrando 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 e 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 di tutti i servizi Internet utilizzati ogni giorno da milioni di utenti, tra cui figurano 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 trascura-

bile, influenzata 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 transistore assorbe prevalentemente energia elettrica durante la commutazione alto-basso-alto (o basso-alto-basso) del suo stato di uscita, secondo la formula

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

dove E rappresenta l'energia dissipata nelle due transizioni di stato, V^+ la tensione di alimentazione e C_L la capacità di carico del transistore.
La potenza dissipata P_D, assumendo che la frequenza di commutazione dello stato del transistore 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 é 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 1V.
Al contempo, la diminuzione della tensione di alimentazione ha favorito la crescita delle correnti di dispersione del transistore, tanto che nel 2008 circa il 40% della potenza assorbita era imputabile a queste correnti: 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 apportato da rilevanti 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 genere destinate al largo consumo da parte di Intel e AMD nel 2006.

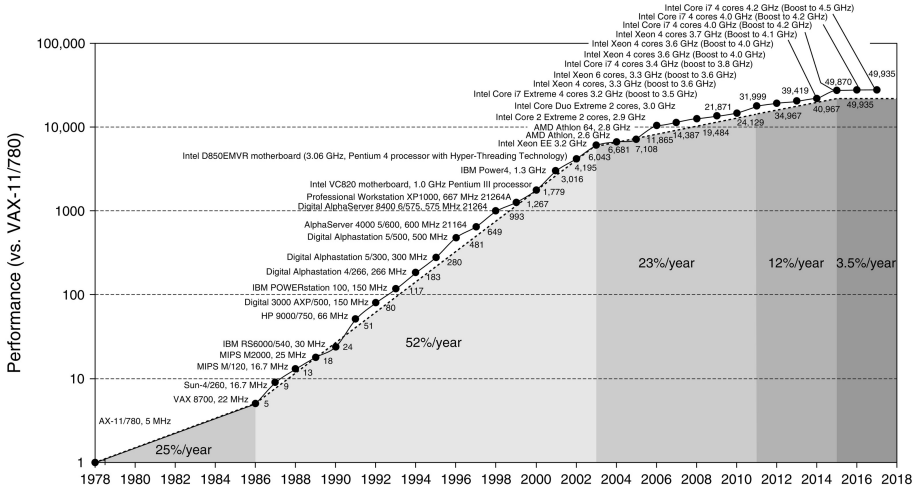

Figura 1.1: Crescita nelle prestazioni dei processori dal 1978 al 2018; il grafico riporta le prestazioni dei processori, paragonandoli al VAX11/780, mediante l'esecuzione dei *benchmark* SPECint *(Da J.L. Hennessy, D.A. Patterson, Computer Architecture: A quantitative Approach. Ed. 6. Waltham, MA:Elsevier, 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 riflesso le risorse di calcolo impiegate [6].
Da un lato, la scalabilità, sfruttando la sinergia tra hardware e software di un 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.

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 monoprocessore, la cui programmazione è di gran lunga più agevole.

Come abbiamo accennato nel paragrafo 1.1, le attività, chiamate *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 consigliato suddividere l'applicazione in maniera tale che ogni processore compi 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, i benefici prestazionali portati dall'impiego di sistemi multiprocessore svanirebbero.

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 compito dato.

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, dedicando idealmente un lasso di tempo irrilevante a questi due aspetti.

Chiaramente, le difficoltá incontrate nella realizzazione di programmi a esecuzione parallela vanno di pari passo con il numero di processori presenti nel sistema.

Un'ulteriore sfida da affrontare durante la progettazione di programmi eseguiti su più processori simultaneamente 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, reputata un'espressione quantitativa dalla legge dei rendimenti decrescenti dell'economista classico David Ricardo.

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 l'entità del miglioramento.
In aggiunta, il concetto di incremento di velocità, o *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.
Questa indicazione dell'incremento di prestazioni viene calcolata 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 misurabili attraverso le metriche prestazionali scelte.

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, $T_{nonModificato}$.

$$T_{dopo} = \frac{T_{modificato}}{Entità\ miglioramento} + T_{nonModificato} \tag{1.1}$$

Possiamo riformulare la legge di Amdahl in termini di incremento di velocità rispetto al tempo di esecuzione iniziale.

$$Speedup = \frac{T_{dopo}}{T_{prima} - T_{dopo}} + \frac{T_{dopo}}{Entità\ miglioramento} \tag{1.2}$$

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

La formula precedente viene comunemente riscritta ponendo pari a 1 il tempo di esecuzione prima dell'*enhancement* ed esprimendo il tempo modificato dal miglioramento come frazione del tempo originario di esecuzione, ottenendo

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

Come è intuibile, la legge di Amdahl può essere applicata alla stima quantitativa del miglioramento delle prestazioni solo se il tempo in cui viene sfruttata una certa funzione all'interno del sistema è noto, così come il suo 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 specifici per contraddistinguere classi di problemi da risolvere.
A titolo di esempio, un problema «*embarrassingly parallel*» è un problema che richiede un minimo sforzo per essere suddiviso in un insieme di *task* indipendenti, a causa del loro debole accoppiamento [7], 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 particolare interesse per l'analisi numerica, è il calcolo approssimato di integrali definiti per funzioni di una o più variabili; diversamente, il processo di addestramento di modelli avanzati 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 «*massively parallel*».

Di seguito, effettuiamo una semplice analisi prestazionale di un problema di grandi dimensioni per studiare da vicino le insidie che si nascondono nella distribuzione e nell'esecuzione di software parallelo su sistemi reali.

Esempio 1.1 (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 monoprocessore e,

successivamente, di un sistema multiprocessore con 30 CPU che supporta la parallelizzazione della somma tra matrici.
Vogliamo analizzare la variazione delle prestazioni dei due sistemi quando:

a) il numero di processori del sistema multiprocessore aumenta 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, calcolata applicando le formule (1.1) e (1.2).

Num. processori / Dim. matrice	**30**	**120**
3000 × 3000	0,7770	0,4727
6000 × 6000	0,8739	0,6375

Tabella 1.1: Frazione dello *speedup* potenziale nei casi proposti dall'esempio 1.1.

L'esempio 1.1 evidenzia il problema fondamentale del calcolo parallelo: aumentare la velocità di esecuzione di un programma a esecuzione parallela su un sistema multiprocessore mantenendo fisse le dimensioni del problema è più difficile rispetto a migliorare le prestazioni incrementando le dimensioni del problema proporzionalmente al numero di unità di calcolo montate nel sistema.

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.

Possiamo giustificare il comportamento descritto in precedenza prendendo come modelli un qualsiasi sistema multiprocessore e un programma a esecuzione parallela. Indichiamo con $P > 1$ il numero di processori presenti nel sistema e denotiamo con M la dimensione del problema risolto dal programma[2].
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 me-

[2]Per semplicità possiamo pensare a M come la dimensione dello spazio da allocare in memoria centrale per la risoluzione del problema.

moria disponibile per ogni CPU, ma a seconda del contesto applicativo considerato possiamo individuare validi motivi a supporto di ciascuno dei due approcci.
In linea di massima, problemi di grandi dimensioni richiedono moli di dati in input, rendendo la scalabilità debole più agevole da raggiungere.

Quest'ultimo esempio chiarisce l'importanza del bilanciamento del carico.

Esempio 1.2 (Bilanciamento del carico per un problema di grandi dimensioni)
Per ottenere un aumento di velocità di 75,4 volte per il problema di grandi dimensioni considerato nel caso b) dell'esempio 1.1, abbiamo implicitamente supposto che il carico fosse perfettamente bilanciato tra i 120 processori del sistema.
Ora vogliamo determinare l'impatto sulle prestazioni nel caso in cui a uno dei processori viene assegnato:

a) il 2,5% del carico totale, ovvero 150 addizioni;
b) il 12,5% del carico totale, ovvero 750 addizioni.

La tabella 1.2 riporta lo *speedup* ottenuto nei due scenari ipotizzati e la rispettiva variazione percentuale calcolata con riferimento alla situazione ideale che presenta un carico perfettamente bilanciato.

	150 addizioni	**750 addizioni**
Speedup	33,50	7,73
$\boldsymbol{\Delta\%_{ideale}}$	−55,57	−89,74

Tabella 1.2: *Speedup* e sua variazione percentuale nei casi proposti dall'esempio 1.2.

L'importanza di suddividere equamente il carico di lavoro tra i processori del sistema é evidente dalla variazione in diminuzione dello *speedup* registrata in entrambi gli scenari di utilizzo.
Una distribuzione non corretta delle attività da eseguire porta a una minore efficienza del sistema con intervalli di tempo non trascurabili in cui la maggioranza delle unità di elaborazione è a riposo, attendendo che i processori in sovraccarico terminino il loro lavoro.

Capitolo 2

Un linguaggio per il calcolo parallelo: MATLAB

L'accesso diffuso a sistemi di elaborazione multiprocessore ha aumentato la domanda di mercato per soluzioni software a supporto dello sviluppo di programmi a esecuzione parallela.
Gli ambienti di programmazione per il calcolo scientifico, MATLAB (abbreviazione di *Matrix Laboratory*) incluso, hanno recepito questa tendenza, creando tutte le condizioni necessarie per l'aggiunta di nuove funzionalità dedicate all'elaborazione parallela.

In questo capitolo forniremo una rapida panoramica delle funzionalità di MATLAB per il calcolo parallelo, con un particolare focus sulle motivazioni e sulle scelte di design che hanno guidato l'implementazione di questa nuova parte del linguaggio.

2.1 Gli ingredienti per un MATLAB parallelo

2.1.1 Una breve prospettiva storica

L'approccio seguito dai progettisti di The MathWorks[1] per estendere MATLAB al mondo del calcolo parallelo è stato modificare le caratteristiche del linguaggio stesso, cominciando dall'aggiunta di *routine* comunemente impiegate nella risoluzione di problemi *embarrassingly parallel*.

[1] La *software house* statunitense, con sede in Massachusetts (Stati Uniti), che si occupa dello sviluppo di MATLAB e di altri prodotti per il calcolo scientifico.

A partire dai primi passi compiuti in questa direzione negli anni Ottanta del secolo scorso da Cleve Moler, l'autore del linguaggio, ci si imbatté nel fatto che il modello di memoria globale di MATLAB, secondo il quale le variabili definite dall'utente e importate dall'esterno vengono conservate in un'area di memoria allocata dalla sessione attiva di MATLAB, era in contrasto con il modello di memoria condivisa impiegato dalla maggioranza dei sistemi multiprocessore.

Questa incompatibilità causò dei rallentamenti al progetto di parallelizzazione di MATLAB, ma le pressioni esterne di coloro che auspicavano a un suo completamento erano troppo insistenti per essere ignorate.
La crescente disponibilità di sistemi multiprocessore aveva reso il calcolo parallelo un argomento presente sulla bocca di tutti gli specialisti del settore; la comparsa delle prime architetture *multicore* e l'ascesa dei *cluster* Beowulf[2] permisero una diffusione massiccia dei sistemi di calcolo ad alte prestazioni, che precedette la «democratizzazione» dei WSC portata dal *cloud computing*.
Inoltre, MATLAB era già allora un ambiente di programmazione affermato all'interno della comunità scientifica e quindi doveva fornire ai propri utenti un prodotto completo e funzionale in tutti gli scenari applicativi, inclusi i progetti a elevata intensità computazionale.

Ecco che nel novembre del 2004 vennero rilasciati al pubblico i primi risultati di questo progetto, sotto le vesti di due pacchetti software addizionali (chiamati *toolbox* nel vocabolario tecnico del linguaggio): il Distributed Computing Toolbox™ e il MATLAB Distributed Computing Engine™[3].

2.1.2 Gli aspetti imprescindibili dell'implementazione

L'adattamento di MATLAB al calcolo parallelo non fu condotto in modo casuale, bensí le aggiunte al linguaggio furono ponderate attentamente a partire dalle informazioni ricavate dai sondaggi condotti nelle fasi preliminari del progetto.
Per questo motivo, il modello di programmazione proposto da MATLAB è adatto all'esecuzione di programmi paralleli su sistemi *multicore* e su *cluster* di elaboratori, trattandosi delle architetture di calcolo parallelo più comuni in ambito industriale.

Di seguito elenchiamo, in ordine decrescente di importanza, gli obiettivi di pro-

[2] I *cluster* Beowulf sono *cluster* costituiti dall'interconessione di componenti hardware commerciali, ad esempio PC al termine della loro vita utile, mediante una tradizionale rete LAN.

[3] I due nomi commerciali sono i corrispettivi degli odierni Parallel Computing Toolbox™ e MATLAB Parallel Server™.

gettazione che ispirarono e continuano a ispirare il processo di parallelizzazione di MATLAB;

- la programmabilità, cioé la capacità di creare programmi che soddisfino i requisiti degli utenti e che siano facili da mantenere per gli sviluppatori;
- l'esecuzione di codice arbitrario sui sistemi multiprocessore supportati;
- l'astrazione da dettagli irrilevanti durante l'implementazione; di conseguenza, lo sviluppatore medio non deve più preoccuparsi di aspetti quali lo *scheduling* delle *task* e la distribuzione dei dati alle unità di lavoro, in quanto vengono gestiti automaticamente dal linguaggio;
- l'indipendenza del programma dall'allocazione delle risorse computazionali: un software parallelo scritto in MATLAB deve funzionare correttamente sia quando eseguito su un sistema multiprocessore che su un sistema monoprocessore, adattando il suo comportamento alle risorse di calcolo disponibili;
- l'accesso a costrutti di programmazione di prima classe[4].

Il percorso di trasformazione di MATLAB al fine di renderlo appetibile al calcolo parallelo non è ancora giunto al termine.
La destinazione finale fissata dagli addetti ai lavori è la realizzazione del modello di linguaggio ideato dal direttore tecnico di TheMathWorks, Roy Lurie [10], secondo cui gli esperti di dominio inseriscono annotazioni minimali al codice sorgente per esprimere l'intenzione di eseguire il programma su più processori simultaneamente.

2.2 Parallel Computing Toolbox

Il Parallel Computing Toolbox, abbreviato in PCT, permette di risolvere problemi *compute-intesive* e *data-intensive* sfruttando la capacità di calcolo offerta dai più recenti sistemi *multicore* e *cluster* di elaboratori.
Costrutti di programmazione di alto livello, come gli *array* distribuiti, consentono di sviluppare applicazioni MATLAB scalabili senza ricorrere alla programmazione

[4]Secondo la classificazione proposta dall'informatico britannico Christopher Strachey [9], i costrutti di programmazione di prima classe possono essere manipolati liberamente nelle istruzioni del linguaggio; in pratica, devono poter essere passati come parametri attuali durante l'invocazione di una procedura, restituiti come valore di ritorno di una funzione e assegnati a variabili o a strutture dati.

MPI[5].
L'applicazione può essere eseguita su *cluster* o su server in *cloud* senza dover apportare alcuna modifica al codice grazie a MATLAB Parallel Server, così da concentrarsi esclusivamente sullo sviluppo dell'algoritmo migliore per il caso d'uso in esame.

Incominciamo il nostro studio del Parallel Computing Toolbox riportando alcune definizioni, tratte dalla documentazione ufficiale di MATLAB [13], riguardanti il modello di programmazione parallela messo a disposizione dal linguaggio.

- *Client*: termine impiegato per identificare la sessione di MATLAB con cui l'utente sta interagendo; tipicamente coincide con il computer usato dallo sviluppatore durante la prototipazione del programma a esecuzione parallela. Attraverso le funzioni che compongono il PCT, il *client* suddivide la computazione in *task* atomiche e le assegna ai *worker*.

- *Parallel Pool*: spesso abbreviato in parpool, è definito come un insieme di *worker* comunicanti che possono eseguire codice interattivamente.

- *Worker*: corrisponde a un'istanza di MATLAB priva di interfaccia grafica, in grado di fornire la potenza del motore di calcolo del linguaggio.

Una prima distinzione da rimarcare è quella fra l'infrastruttura e i componenti del linguaggio inclusi negli strumenti di calcolo parallelo di MATLAB.
Il linguaggio comprende costrutti di programmazione parallela e funzioni con supporto automatico al parallelismo, mentre l'infrastruttura riguarda i meccanismi che coaudivano il linguaggio, come il protocollo di trasferimento del codice e dei dati alle unità di lavoro.
Nelle prossime sezioni, esamineremo da vicino alcuni costrutti paralleli proprio del linguaggio, astraendo dall'infrastruttura sottostante, nonostante entrambe le componenti siano imprescindibili per il corretto funzionamento del PCT.

2.2.1 L'architettura di riferimento

L'architettura di un cluster di elaboratori impiegata per l'analisi del Parallel Computing Toolbox è schematizzata in figura 2.1.
MATLAB Parallel Server riunisce un insieme di *worker*, in esecuzione sui nodi del *cluster*, che ricevono le *task* computazionali assegnate dal *client* attraverso specifiche

[5]La *Message Passing Interface* (MPI) rappresenta lo standard per il modello di comunicazione interprocesso basato sullo scambio di messaggi nei sistemi distribuiti [12], stabilendo la sintassi e la semantica di funzioni di libreria impiegate nella scrittura di software parallelo in C, C++ e Fortran.

funzioni del Parallel Computing Toolbox.
I *worker* leggono il codice da eseguire e i dati su cui lavorare da una memoria di massa condivisa popolata dall'*head node* (non rappresentato in figura), un nodo del sistema responsabile della schedulazione delle attività.
Una volta terminata l'elaborazione, i risultati sono raccolti dall'*head node* e posti all'interno dello spazio di lavoro del *client* mediante i canali di comunicazione instaurati tra quest'ultimo e i *worker*.

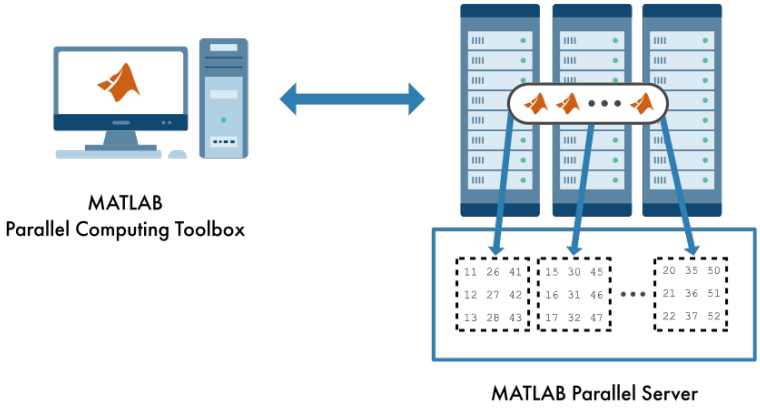

Figura 2.1: Architettura di riferimento per gli strumenti di calcolo parallelo di MATLAB. (Da `https://it.mathworks.com/products/matlab-parallel-server.html`)

Giunti a questo punto, introduciamo i paradigmi di programmazione parallela supportati da MATLAB:

- parallelizzazione implicita: alcune funzioni, quando richiamate dal codice sorgente del programma, sfruttano le librerie di *runtime* del linguaggio in modo da essere eseguite su *thread* distinti all'interno della stessa sessione;

- parallelizzazione esplicita: il carico di lavoro del programma è automaticamente suddiviso in *task* elementari, ciascuna delle quali viene attribuita a un *worker* per l'esecuzione.

2.2.2 Il paradigma di programmazione parallela implicita

I *toolbox* di MATLAB sono dotati di un crescente numero di funzioni con supporto automatico al parallelismo, al fine di beneficiare di tutti i vantaggi dati dall'elaborazione parallela senza modificare il codice eventualmente scritto per la versione

seriale di un programma, in accordo con i principi di design elencati nella sezione 2.1.2.

Alcune funzioni, come `mldivide` per la risoluzione di sistemi di equazioni lineari, sono eseguite automaticamente su *thread* distinti, se richiamate dalla sessione attiva di MATLAB.

Ragionando sulla nostra architettura di riferimento, il parallelismo implicito viene attivato solo quando la funzione è eseguita direttamente dal *client*, mentre è sconsigliato quando l'esecuzione è a carico dei nodi del *cluster*, allo scopo di evitare un parallelismo «annidato» che degraderebbe le prestazioni dell'intero sistema.
In quest'ottica, possiamo notare come i progettisti del linguaggio abbiano ideato i *worker* come unità di elaborazione a singolo *thread*.

Quando il *client* incontra una funzione con supporto automatico al parallelismo nel codice sorgente del programma, avvia un *parallel pool* per la sua esecuzione in parallelo.
Un apposito profilo di configurazione determina le caratteristiche dell'ambiente di elaborazione parallela; nello specifico, il Parallel Computing Toolbox permette di scegliere tra i seguenti profili preimpostati:

- *Processes*: i *worker* vengono attivati come processi in esecuzione sui *core* fisici del calcolatore che ospita la sessione principale di MATLAB.

- *Threads*: i *worker* sono in esecuzione su dei *thread* e non più su processi veri e propri.

I vantaggi portati dall'ambiente parallelo *Threads* sono un minor consumo di memoria, un basso costo di comunicazione tra i *worker* e una schedulazione delle attività particolarmente performante, a scapito della disponibilità di un'ampia gamma di funzioni con supporto al parallelismo implicito su *thread*.

Relativamente alla scelta del numero di *worker* nell'ambiente *Processes* é consigliato riservare un motore di calcolo per ogni *core* fisico disponibile, ignorando la presenza di eventuali *core* virtuali; infatti, questi ultimi condividono alcune risorse di calcolo appartenenti allo stesso processore, tra cui la *Floating Point Unit* (FPU), e poiché la maggior parte delle elaborazioni in MATLAB richiede l'esecuzione di operazioni aritmetiche in virgola mobile, limitare il numero di *worker* per CPU a uno migliora la stabilità del sistema.
L'unica eccezione è rappresentata dalle applicazioni *data-intensive*, per le quali

potrebbe essere conveniente innalzare il numero di *worker* per *core* a due.

Indipendentemente dall'ambiente di esecuzione selezionato, un singolo parpool a supporto della parallelizzazione implicita può contenere fino a 512 *worker*, a prescindere dalle specifiche del calcolatore utilizzato.

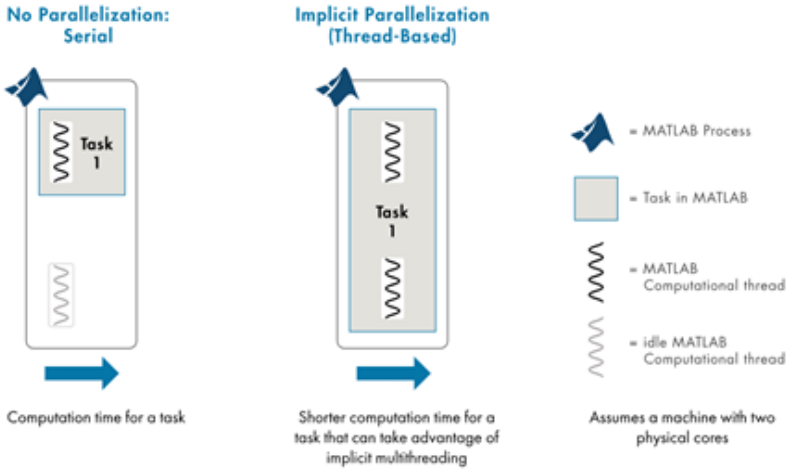

Figura 2.2: Rappresentazione del modello di parallelizzazione implicita di MATLAB su un sistema *dual-core* (Da `https://it.mathworks.com/discovery/matlab-multicore.html`)

Se una funzione non include il supporto automatico al parallelimo, possiamo trasferire l'esecuzione del programma su una *workstation*, in modo da beneficiare dello *speedup* garantito da un sistema con maggiore capacità di calcolo, oppure possiamo utilizzare il paradigma di programmazione parallela esplicita offerto da MATLAB.

2.2.3 Il paradigma di programmazione parallela esplicita

Il modello di programmazione parallela esplicita esposto dal Parallel Computing Toolbox è un'applicazione del parallelismo a livello dei dati, che si fonda sull'esistenza di costrutti di programmazione parallela di cui i programmatori possono avvalersi durante il processo di sviluppo.
Il meccanismo a bassissimo livello per la scrittura di programmi a elaborazione parallela è la comunicazione basata su scambio di messaggi tra *worker* appertenenti al medesimo *parallel pool*, ma questo approccio viene spesso criticato essendo considerato l'equivalente del linguaggio Assembly per il calcolo parallelo.
Con l'obiettivo di agevolare la scrittura di software parallelo, alcuni costrutti di programmazione di alto livello sono stati introdotti nel linguaggio, aumentando il

livello di astrazione del codice e consentendo la scrittura di algoritmi paralleli simili alle loro controparti seriali.

Un esempio di costrutto di alto livello per la programmazione parallela è incarnato dagli *array* [6] distribuiti, strutture dati il cui contenuto viene partizionato tra i *worker* di uno stesso *cluster*.
L'impiego degli *array* distribuiti permette di memorizzare strutture dati di dimensioni tali da non poter essere contenute nella memoria centrale di un singolo calcolatore, sfruttando la capacità di memorizzazione offerta dalla combinazione di tutti i nodi del *cluster*.

Gli *array* distribuiti possono contenere dati di qualsiasi tipo e supportano la distribuzione degli elementi lungo una dimensione, per riga oppure per colonna.
A questo proposito, dobbiamo precisare che esiste la possibilità di definire distribuzioni dei dati alternative e che, molto spesso, il partizionamento degli elementi viene manipolato implicitamente dall'esecuzione di certe operazioni come `gather` (una funzione utile a riunire un *array* distribuito all'interno dello spazio di lavoro del *client*).

Un ulteriore vantaggio derivante dall'uso degli *array* distribuiti è l'assenza di differenze sintattiche per l'accesso agli elementi rispetto agli *array* tradizionali; l'infrastruttura sottostante garantisce in ogni momento una distribuzione dei dati idonea all'esecuzione delle operazioni richieste dall'utente.
Questo ampio raggio di manovra consentito al programmatore potrebbe introdurre consistenti *overhead* di comunicazione tra i *worker*, ma sappiamo che la programmabilità rappresenta l'obiettivo di design prioritario nel processo di parallelizzazione di MATLAB, surclassando persino le prestazioni.

Centinaia di funzioni native, e altrettante presenti nei *toolbox* sviluppati dalla *community*, sono state progettate per lavorare sinergicamente con *array* distribuiti.
Ad esempio, MATLAB prevede un enorme insieme di funzioni parallele per l'algebra lineare che operano su *array* distribuiti, implementate a partire dalle procedure definite dalla libreria di algebra lineare numerica ScaLAPACK.

[6] Secondo la terminologia di MATLAB, la parola *array* è un termine universale per riferirsi a strutture dati ospitanti vettori riga, vettori colonna o matrici.

Capitolo 3

Il metodo di Jacobi parallelo

Innumerevoli modelli matematici, come quello che descrive il processo di diffusione di un soluto in un solvente di una soluzione chimica, si formulano per mezzo di equazioni differenziali.
Non è una novità il fatto che per la maggior parte delle equazioni differenziali l'integrale generale non sia esprimibile in forma esplicita, per cui è necessario ricorrere a metodi di integrazione numerica per la loro risoluzione.
Tra questi, spicca il metodo degli elementi finiti, un metodo per l'approssimazione di equazioni differenziali attraverso sistemi di equazioni lineari.

I problemi reali, in campo ingegneristico e fisico, dipendono da migliaia di variabili per cui lo sviluppo di metodi efficienti per risolvere sistemi lineari di grandi dimensioni non è un capriccio squisitamente teorico, ma trova fondamentali applicazioni in tutti i settori in cui la modellistica è impiegata.

L'obiettivo di questo capitolo è la presentazione del metodo di Jacobi, un metodo iterativo per la risoluzione di sistemi di equazioni lineari, e la sua applicazione a problemi di grandi dimensioni.
I concetti esposti nei capitoli 1 e 2 torneranno utili nell'affrontare questo problema, matematicamente elementare ma computazionalmente intrigante, sfruttando l'elevata capacità di calcolo messa a disposizione dai sistemi a elaborazione parallela.

Di seguito, faremo riferimento a svariati risultati propri dell'algebra lineare e dell'analisi numerica, indispensabili per una trattazione accurata degli argomenti in questione, per le cui dimostrazioni si rimanda a [15] e a [16].

3.1 Il contesto

Siano $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ una matrice quadrata di ordine $n \geq 1$ a coefficienti reali, $\boldsymbol{b} = (b_i) \in \mathbb{R}^n$ e $\boldsymbol{x} = (x_i) \in \mathbb{R}^n$ dei vettori colonna di numeri reali.

Consideriamo il sistema di equazioni lineari scritto in forma matriciale

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

dove A è la matrice dei coefficienti del sistema, $\boldsymbol{b}$ il vettore dei termini noti e $\boldsymbol{x}$ il vettore delle incognite.
Il sistema (3.1) rappresenta un insieme di n relazioni algebriche in n incognite del tipo

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, n \tag{3.2}$$

per il quale siamo interessati a determinarne le soluzioni, ovvero trovare delle n-uple di valori x_i che soddisfino la (3.2).

Ricordiamo che l'esistenza e l'unicità della soluzione di (3.1) è garantita se e solo se sono soddisfatte le seguenti condizioni, equivalenti tra di loro:

1. $\det(A) \neq 0$, dove $\det(A)$ denota il determinante della matrice A;
2. A è invertibile;
3. $\text{car}(A) = n$, dove $\text{car}(A)$ denota la caratteristica (o rango) di A, corrispondente al massimo numero di colonne (o righe) linearmente indipendenti della matrice;
4. il sistema omogeneo associato $A\boldsymbol{x} = \boldsymbol{0}$ ammette come unica soluzione il vettore nullo.

La soluzione del sistema $A\boldsymbol{x} = \boldsymbol{b}$ può essere espressa in forma chiusa tramite la regola di Cramer

$$x_j = \frac{\Delta_j}{\det(A)}, \quad j = 1, \ldots, n \tag{3.3}$$

con Δ_j il determinante della matrice ottenuta sostituendo la j-esima colonna di A con il vettore dei termini noti $\boldsymbol{b}$.

Pur rappresentando un risultato fondamentale dell'algebra lineare, la regola di Cramer trova scarsa applicazione in ambito numerico per via del suo elevato costo computazionale.

Denotando con Δ_{ij} il determinante della matrice di ordine $n-1$ ottenuta da A eliminando la i-esima riga e la j-esima colonna e con $A_{ij} = (-1)^{i+j}\Delta_{ij}$ il complemento algebrico dell'elemento a_{ij}, possiamo sfruttare la regola di Laplace per il calcolo effettivo del determinante di A

$$\det(A) = \begin{cases} a_{11} & \text{se } n = 1, \\ \displaystyle\sum_{j=1}^{n} A_{ij} a_{ij} & \text{per } n > 1. \end{cases} \tag{3.4}$$

Supponendo di calcolare i determinanti tramite la (3.4), il costo computazionale della regola di Cramer è dell'ordine di $(n+1)!$ flops (*floating point operations per second*), un costo non accettabile anche per problemi di piccole dimensioni.
Ad esempio, il *supercomputer* attualmente più potente al mondo, soprannominato «El Capitan» e ospitato dal Lawrence Livermore National Laboratory in California (Stati Uniti) [18], è caratterizzato da una velocità pari a $1{,}742 \times 10^{18}$ flops, ma impiegherebbe circa $2{,}82 \times 10^{40}$ anni[1]a risolvere un sistema lineare di 50 equazioni con il metodo di Cramer, mentre un normale PC è in grado di risolvere modelli matematici con migliaia di vincoli in meno di un secondo, sfruttando algoritmi a elevata efficienza.

Alla luce di queste osservazioni, la necessità di sviluppare metodi numerici alternativi per la risoluzione di sistemi di equazioni lineari è evidente. Tali metodi vengono tradizionalmente distinti in metodi diretti se permettono la risoluzione del sistema in un numero finito di passi oppure iterativi se richiedono un numero di passi teoricamente infinito.
Preferire un metodo iterativo al posto di un metodo diretto, o viceversa, non dipende esclusivamente dall'efficienza dell'algoritmo in sè, ma anche dalle proprietà dei vettori e delle matrici coinvolte nel problema nonchè dalle specifiche del sistema di elaborazione relativamente alla capacità di memoria e all'architettura adottata.

Nei paragrafi successivi, ci addentreremo nell'analisi dei metodi iterativi, soffermandoci in particolar modo sul metodo di Jacobi per la risoluzione di sistemi di equazioni lineari.

[1]Per $n = 50$, il costo computazionale è dell'ordine di $(50+1)! \simeq 1{,}55 \times 10^{66}$ flops.
Disponendo di una capacità di calcolo di $1{,}74 \times 10^{18}$ flops, eseguiremmo un operazione in $5{,}74 \times 10^{-19}$ s, richiedendo comunque un tempo di risoluzione del sistema pari a $8{,}90 \times 10^{47}$ s, ovvero all'incirca $2{,}82 \times 10^{40}$ anni.

3.2 Risoluzione di un sistema lineare con metodi iterativi

Impiegare un metodo numerico per la risoluzione di un sistema lineare introduce necessariamente degli errori di arrotondamento, dovuti alla rappresentazione dei numeri reali sul calcolatore con un numero finito di cifre, che fortunatamente non si ripercuotono sull'accuratezza della soluzione finale nel caso di metodi numerici stabili, come quelli che analizzeremo.

Nello scenario peggiore in cui la matrice dei coefficienti A sia piena, ovvero i suoi elementi siano tendenzialmente non nulli, il costo computazionale di un metodo iterativo è dell'ordine di n^2 operazioni per ogni iterazione: una magnitudine inferiore, di diversi ordini, al costo associato al metodo di Cramer.

Inoltre, per una corretta quantificazione degli errori introdotti, ci avvarremo dei concetti di norma vettoriale e di norma matriciale e del legame esistente tra quest'ultima e il raggio spettrale di una matrice.
Presupponiamo, fin da subito, che questi argomenti siano famigliari, così come le principali proprietà relative alle successioni di vettori e di matrici.

3.2.1 Costruzione di un metodo iterativo lineare

I metodi iterativi si fondano sull'idea di calcolare una successione di vettori $\{\boldsymbol{x}^{(k)}\}$, i cui elementi godano della proprietà di convergenza

$$\lim_{k\to\infty} \boldsymbol{x}^{(k)} = \boldsymbol{x}, \tag{3.5}$$

dove $\boldsymbol{x}$ è la soluzione di (3.1).
Ovviamente, ci vogliamo fermare al minimo m per cui vale

$$\|\boldsymbol{x}^{(m)} - \boldsymbol{x}\| < \varepsilon$$

con ε una tolleranza a piacimento e $\|\cdot\|$ un'opportuna norma vettoriale.

Poiché la soluzione esatta del sistema (3.1) non è nota a priori, introdurremo degli adeguati criteri di arresto basati su altre grandezze nella sezione 3.2.2.

Una strategia largamente impiegata nella costruzione della successione $\{\boldsymbol{x}^{(k)}\}$ consiste nella decomposizione additiva della matrice dei coefficienti A in $A = P - N$, dove $P, N \in \mathbb{R}^{n\times n}$ e P è non singolare.

In particolare, assegnato il vettore iniziale $\boldsymbol{x}^{(0)}$, otteniamo $\boldsymbol{x}^{(k)}$ per $k \geq 1$ risolvendo due nuovi sistemi di n equazioni in n incognite

$$P\boldsymbol{x}^{(k)} = N\boldsymbol{x}^{(k-1)} + \boldsymbol{b}, \quad k \geq 1 \tag{3.6}$$

Possiamo riscrivere, in maniera del tutto equivalente, la (3.6) come

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k-1)} + P^{-1}\boldsymbol{r}^{(k-1)}, \quad k \geq 1, \tag{3.7}$$

avendo indicato

$$\boldsymbol{r}^{(k-1)} = \boldsymbol{b} - A\boldsymbol{x}^{(k-1)}$$

il vettore residuo alla k-esima iterazione.
Per il calcolo iterativo della soluzione approssimata $\boldsymbol{x}^{(k)}$, dobbiamo determinare il residuo del sistema e risolvere un nuovo sistema lineare di matrice P.
Questo procedimento risulta conveniente nell'ipotesi in cui P sia invertibile con un basso costo computazionale.
Definendo

$$\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x} \tag{3.8}$$

come l'errore al passo k e osservando che, dalla decomposizione di A, si ricava $P\boldsymbol{x} = N\boldsymbol{x} + \boldsymbol{b}$ otteniamo la seguente relazione sull'errore

$$\boldsymbol{e}^{(k)} = B\boldsymbol{e}^{(k-1)} \quad \text{con} \quad B = P^{-1}N \tag{3.9}$$

dove $B \in \mathbb{R}^{n\times n}$ è chiamata matrice di iterazione associata allo *splitting* $A = P - N$.
Pertanto, applicando ricorsivamente la (3.9), arriviamo a

$$\boldsymbol{e}^{(k)} = B^k\boldsymbol{e}^{(0)}, \quad k = 0, 1, \ldots. \tag{3.10}$$

La condizione di convergenza (3.5) può essere riformulata in funzione dell'errore come $\boldsymbol{e}^{(k)} \to \boldsymbol{0}$ per $k \to \infty$, soddisfatta per ogni scelta del vettore $\boldsymbol{x}^{(0)}$.
In virtù della (3.10), la nuova proprietà di convergenza risulta verificata se e solo se $B^k \to 0$ per $k \to \infty$.

Relativamente alla convergenza alla soluzione esatta di un metodo iterativo, esponiamo i seguenti risultati senza dimostrarli.

Teorema 3.1 *Il metodo iterativo* (3.6) *converge alla soluzione di* (3.1) *per ogni scelta del vettore iniziale* $\boldsymbol{x}^{(0)}$ *se e solo se* $\rho(B) < 1$.

Corollario 3.1 *Una condizione sufficiente per la convergenza del metodo* (3.6) *è* $\|B\| < 1$ *per qualche norma matriciale* $\|\cdot\|$ *consistente.*

Teorema 3.2 *Sia* $A = P - N$*, con* A *e* P *simmetriche e definite positive. Se la matrice* $2P - A$ *è definita positiva, allora il metodo iterativo* (3.6) *converge per ogni valore del vettore iniziale* $\boldsymbol{x}^{(0)}$ *e si ha*

$$\rho(B) = \|B\|_A = \|B\|_P < 1.$$

Inoltre, la convergenza del metodo è monotona rispetto alle norme $\|\cdots\|_A$ *e* $\|\cdots\|_P$*, ovvero per ogni* $k \geq 1$ *valgono le seguenti relazioni*

$$\|\boldsymbol{e}^{(k+1)}\|_A < \|\boldsymbol{e}^{(k)}\|_A$$
$$\|\boldsymbol{e}^{(k+1)}\|_P < \|\boldsymbol{e}^{(k)}\|_P.$$

3.2.2 Criteri di arresto per metodi iterativi

Un importante aspetto ancora da esaminare è la ricerca dei criteri di arresto, vale a dire le condizioni da soddisfare per decidere quando fermare l'esecuzione di un metodo iterativo.

Un primo criterio si basa sul controllo dell'incremento: data una tolleranza ε fissata, ci fermiamo al primo valore di k per il quale si abbia

$$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\| < \varepsilon,$$

stimando il corrispondente errore $\|\boldsymbol{e}^{(k+1)}\|$ all'ultima iterazione.

Siano B la matrice di iterazione del metodo iterativo considerato, dalla relazione ricorsiva sull'errore $\boldsymbol{e}^{(k+1)} = B\boldsymbol{e}^{(k)}$ otteniamo

$$\|\boldsymbol{e}^{(k+1)}\| \leq \|B\| \, \|\boldsymbol{e}^{(k)}\|. \tag{3.11}$$

Sfruttando la disuguaglianza triangolare e il fatto che $\boldsymbol{e}^{(k+1)} - \boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}$, giungiamo a

$$\|\boldsymbol{e}^{(k+1)}\| \leq \|B\|\Big(\|\boldsymbol{e}^{(k+1)}\| + \|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|\Big)$$

e quindi se $\|B\| < 1$ (come richiesto dal Corollario 3.1)

$$\|\boldsymbol{x} - \boldsymbol{x}^{(k+1)}\| \leq \frac{\|B\|}{1 - \|B\|}\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\| \leq \frac{\|B\|}{1 - \|B\|}\varepsilon. \tag{3.12}$$

Pertanto, l'errore $\|\boldsymbol{e}^{(k+1)}\|$ è contenuto purchè $\|B\| \simeq 1$.

Un altro test d'arresto, più pratico dal punto di vista computazionale, si basa sul controllo del residuo normalizzato: ci fermiamo al primo valore di k per il quale si ottiene $\|\boldsymbol{r}^{(k)}\|/\|\boldsymbol{r}^0\| \leq \varepsilon$, con ε una tolleranza scelta a priori.
Nel caso particolare in cui $\boldsymbol{x}^{(0)} = \mathbf{0}$, la condizione equivale a richiedere

$$\frac{\|\boldsymbol{r}^{(k)}\|}{\|\boldsymbol{b}\|} \leq \varepsilon.$$

Inoltre, possiamo quantificare l'errore relativo commesso come

$$\frac{\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|}{\|\boldsymbol{x}\|} = \frac{\|A^{-1}\boldsymbol{r}^{(k)}\|}{\|\boldsymbol{x}\|} = \frac{\|A^{-1}\boldsymbol{r}^{(k)}\|}{\|\boldsymbol{x}\|} \leq K(A)\frac{\|\boldsymbol{r}^{(k)}\|}{\|\boldsymbol{b}\|} \leq K(A)\varepsilon,$$

dove $K(A) = \|A\|\,\|A^{-1}\|$ è detto numero di condizionamento della matrice A, un indicatore circa la stabilità della soluzione del sistema (3.1) rispetto alle perturbazioni applicate ad A e a $\boldsymbol{b}$.
In definitiva, questo criterio d'arresto è consigliato quando $K(A) \simeq 1$, ovvero quando A è ben condizionata: piccole perturbazioni su A e $\boldsymbol{b}$ causano piccole variazioni su $\boldsymbol{x}$.

3.3 Il metodo di Jacobi

Se gli elementi sulla diagonale principale di A sono non nulli, possiamo mettere in evidenza in ogni equazione di (3.2) la corrispondente incognita, ottenendo il sistema lineare equivalente

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij}x_j\right), \quad i = 1, \ldots, n. \tag{3.13}$$

Dato un vettore iniziale $\boldsymbol{x}^{(0)}$ scelto arbitrariamente, il metodo di Jacobi calcola $\boldsymbol{x}^{(k+1)}$ come segue

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij}x_j{}^{(k)}\right), \quad i = 1, \ldots, n. \tag{3.14}$$

La (3.14) è un caso particolare della decomposizione additivia $A = P - N$, con

$$P = D, \qquad N = D - A = E + F,$$

dove $D = diag(a_{ii}) \in \mathbb{R}^{n\times n}$ è la matrice diagonale contenente gli elementi diagonali di A, $E = (e_{ij}) \in \mathbb{R}^{n\times n}$ è la matrice triangolare inferiore con $e_{ij} = -a_{ij}$ se $i > j$ ed $e_{ij} = 0$ se $i \leq j$, mentre $F = (f_{ij}) \in \mathbb{R}^{n\times n}$ è la matrice triangolare superiore con $f_{ij} = -a_{ij}$ se $j > i$ e $f_{ij} = 0$ se $j \leq i$. Pertanto $A = D - (E + F)$.

La corrispondente matrice di iterazione B_J è data da

$$B_J = P^{-1}N = D^{-1}(E + F) = I - D^{-1}A. \tag{3.15}$$

3.3.1 Convergenza del metodo di Jacobi

Esistono particolari classi di matrici per le quali è possibile stabilire a priori la convergenza del metodo di Jacobi.

Iniziamo con l'introdurre la definizione di matrice a dominanza diagonale per righe, una proprietà fondamentale per garantire la convergenza del metodo.

Definizione 3.1 *Sia $M = (m_{ij}) \in \mathbb{R}^{n\times n}$ una matrice quadrata di ordine $n \geq 1$ a coefficienti reali, allora M è detta a dominanza diagonale per righe se*

$$|m_{ij}| \geq \sum_{j=1,\, j\neq i}^{n} |m_{ij}|, \quad i = 1, \cdots, n$$

Se la disuguaglianza precedente è verificata in senso stretto, M è detta a dominanza diagonale stretta per righe.

Ora possiamo esporre i risultati di convergenza validi per il metodo di Jacobi.

Teorema 3.3 *Se A è una matrice a dominanza diagonale stretta per righe, allora il metodo di Jacobi converge.*

Teorema 3.4 *Se A e $2D - A$ sono matrici simmetriche definite positive, allora il metodo di Jacobi converge e $\rho(B_J) = \|B_J\|_A = \|B_J\|_D$.*

Bibliografia

[1] D. A. Patterson e J. L. Hennessy, *Struttura e progetto dei calcolatori*, 5ª ed. Bologna: Zanichelli, 2022, A cura di Alberto Borghese.

[2] A. Silberschatz, P. B. Galvin e G. Gagne, *Sistemi operativi: Concetti ed esempi*, 9ª ed. Milano: Pearson, 2014, A cura di Riccardo Melen.

[3] J. L. Hennessy e D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6ª ed. Waltham: Elsevier, 2019.

[4] P. Lu, *A Partial History of Parallel Computing*, set. 2007. visitato il giorno 30 lug. 2025. indirizzo: `https://webdocs.cs.ualberta.ca/~paullu/C681/parallel.timeline.html`.

[5] P. Spirito, *Elettronica Digitale*, 3ª ed. Milano: McGraw-Hill Education, 2021.

[6] M. Michael, J. E. Moreira, D. Shiloach e 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.

[7] M. Herlihy e N. Shavit, *The Art of Multiprocessor Programming*, 1ª ed. Elsevier, 2011.

[8] G. Sharma e J. Martin, «MATLAB®: A Language for Parallel Computing,» *International Journal of Parallel Programming*, vol. 37, n. 1, pp. 3–36, 2009. DOI: `10.1007/s10766-008-0082-5`. indirizzo: `https://doi.org/10.1007/s10766-008-0082-5`.

[9] H. Abelson e G. J. S. with Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd. Cambridge, MA: The MIT Press, 1996.

[10] R. Lurie. «Language Design for an Uncertain Hardware Future,» visitato il giorno 28 lug. 2025. indirizzo: `https://www.hpcwire.com/2008/08/28/ecosystems_are_messy-1/`.

[11] The MathWorks, Inc. «Get Started with Parallel Computing Toolbox,» MathWorks, visitato il giorno 30 lug. 2025. indirizzo: https://it.mathworks.com/help/parallel-computing/getting-started-with-parallel-computing-toolbox.html.

[12] New Mexico State University High Performance Computing. «Introduction to MPI,» New Mexico State University, visitato il giorno 30 lug. 2025. indirizzo: https://hpc.nmsu.edu/discovery/mpi/introduction/.

[13] The MathWorks, Inc. «What Is Parallel Computing?» MathWorks, visitato il giorno 30 lug. 2025. indirizzo: https://it.mathworks.com/help/parallel-computing/what-is-parallel-computing.html.

[14] The MathWorks, Inc. «Quick Start Parallel Computing in MATLAB,» MathWorks, visitato il giorno 30 lug. 2025. indirizzo: https://uk.mathworks.com/help/parallel-computing/quick-start-parallel-computing-in-matlab.html.

[15] R. Betti, *Elementi di Geometria e Algebra Lineare*, 1ª ed. Bologna: Società editrice Esculapio, 2000.

[16] A. Quarteroni, R. Sacco e F. Saleri, *Matematica Numerica*, 2ª ed. Milano: Springer-Verlag, 2000.

[17] A. Quarteroni, *Elementi di Calcolo Numerico*, 2ª ed. Bologna: Società editrice Esculapio, 1997.

[18] T. Jeremy. «Lawrence Livermore National Laboratory's El Capitan verified as world's fastest supercomputer,» visitato il giorno 31 lug. 2025. indirizzo: https://www.llnl.gov/article/52061/lawrence-livermore-national-laboratorys-el-capitan-verified-worlds-fastest-supercomputer.