
UNIVERSITÀ DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea in Informatica

# Studio delle perfomance dell'implementazione parallela in MATLAB di alcuni algoritmi matriciali

**Relatore:**
Prof. Giuseppe Rodriguez

**Candidato:**
Alessandro Argiolas

Anno Accademico 2016/2017

# Indice

**Introduzione** **1**

**1 Introduzione al calcolo parallelo** **2**
1.1 Classificazione delle architetture parallele . . . . . 3
1.1.1 Accesso alla memoria nelle architetture MIMD . . . . . 4
1.2 Il paradigma dello scambio di messaggi . . . . . 5
1.2.1 L'algoritmo per architetture SISD . . . . . 6
1.2.2 L'algoritmo per architetture MIMD . . . . . 7
1.2.3 L'algoritmo per architetture SIMD . . . . . 8
1.3 I parametri di valutazione del software parallelo . . . . . 9

**2 Calcolo Parallelo e Distribuito con MATLAB** **11**
2.1 Introduzione a MATLAB . . . . . 11
2.1.1 Parallelizzazione implicita: la libreria BLAS . . . . . 12
2.2 PCT - Il toolbox per il calcolo parallelo . . . . . 12
2.2.1 Cicli *for* in parallelo: il costrutto *parfor* . . . . . 13
2.2.2 Single Program, Multiple Data: il costrutto *spmd* . . . . . 14
2.2.3 Valutazione di funzioni in parallelo: il costrutto *parfeval* . . . . . 16
2.2.4 Matrici distribuite e codistribuite . . . . . 17
2.2.5 Sviluppo parallelo interattivo: *pmode* . . . . . 19

**3 Parallelizzazione in MATLAB: prodotto matrice per vettore** **21**
3.1 Calcolo delle prestazioni . . . . . 21

**4 Parallelizzazione del metodo di Jacobi** **23**
4.1 Costruzioni di metodi iterativi lineari . . . . . 23
4.2 Gli algoritmi in MATLAB . . . . . 24
4.2.1 L'algoritmo parallelo . . . . . 25
4.3 Calcolo delle prestazioni . . . . . 25
4.3.1 Prestazioni al variare della dominanza diagonale . . . . . 26
4.3.2 Prestazioni al variare del numero di processori . . . . . 27
4.3.3 Prestazioni al variare della dimensione della matrice . . . . . 28
4.3.4 Prestazioni al variare degli elementi nulli della matrice . . . . . 29

**5 Conclusioni** **30**

**Bibliografia** **31**

**Appendice** **33**

# Introduzione

La scelta di tale argomento come tesi di laurea, è il risultato dell'interesse per il calcolo scientifico, nato grazie alle lezioni di "Calcolo Scientifico e Metodi Numerici" tenute dal Professor Rodriguez.
È rilevante l'importanza del calcolo scientifico per lo sviluppo di nuove applicazioni, basti pensare che all'analisi numerica sono attribuiti successi come la tomografia computerizzata, la risonanza magnetica e la risoluzione di problemi della multimedialità. Algoritmi numerici sono applicati quotidianamente per risolvere problemi scientifici e tecnici; ne sono esempi la progettazione di strutture come ponti e aeroplani, le previsioni meteorologiche, l'analisi di molecole. Essendo il calcolo parallelo una delle metodologie principali del calcolo scientifico, è convenuto spenderci un pò di tempo.

## Contesto generale

Attualmente i computer sono indispensabili: la tecnologia fa ormai parte della vita quotidiana, telefoni cellulari, notebook, tablet, console per videogiochi, registratori di cassa, POS, ATM e quant'altro. Ciò che si esige da queste apparecchiature è che svolgano i loro compiti con efficacia e nel minor tempo possibile. L'introduzione del parallelismo, ovvero del calcolo parallelo, ha reso possibile soddisfare il continuo desiderio di applicazioni sempre più veloci. Tale tecnica, oltre a velocizzare svariati algoritmi, permette di gestire grandi moli di dati. Si pensi, a titolo di esempio, all'enorme quantità di informazioni che devono gestire applicazioni quali Facebook e Google. L'esigenza di macchine più sofisticate ed efficienti non risiede nel solo uso quotidiano, ma anche nel progresso scientifico: oggi le infrastrutture per il calcolo e la comunicazione sono uno strumento indispensabile per l'avanzamento della conoscenza in tutte le scienze, dalla fisica alla biologia, dall'architettura all'economia.

## Struttura dell'elaborato

L'elaborato si compone di tre parti. La prima, compilativa, introduce le caratteristiche principali del calcolo parallelo, come approccio iniziale all'argomento al fine di consentire lo sviluppo della parte pratica. La stesura della prima parte si basa, essenzialmente, sulle teorie descritte da Almerico Murli in [1] e Pasquale De Luca in [2]. La seconda introduce i comandi principali di MATLAB necessari per le sperimentazioni sulla parallelizzazione esplicita del software, descritti da Yair Altman in [4]. Nella terza parte, puramente sperimentale, saranno riportati i test effettuati sul prodotto matrice per vettore e sul metodo di Jacobi.

# Capitolo 1

# Introduzione al calcolo parallelo

Il modo più semplice per spiegare il funzionamento delle architetture parallele, è quello di raffrontarle a una squadra di operai che sovraintende alla costruzione di una casa.[1] Il calcolatore sequenziale corrisponde al singolo operaio cha affronta da solo l'intero lavoro: egli espleta ognuno dei compiti assegnatogli, seguendo un ordine preciso.

L'architettura del calcolatore sequenziale si basa sullo schema funzionale della **macchina di Von Neumann**[2] (Figura 1.1), nella quale una singola **CPU** (*Central Processing Unit*, unità di elaborazione) viene designata per eseguire l'intero set di istruzioni. Una CPU è composta dalla **CU** (*Control Unit*, unità di controllo), che si occupa del trasferimento dei dati e della corretta esecuzione dei programmi; dalla **ALU** (*Aritmetic Logic Unit*, unità logico-aritmetica), preposta all'esecuzione di operazioni logiche e aritmetiche; un accumulatore, ora chiamato **registro** della CPU, che corrisponde alla memoria della CPU usata per le elaborazioni interne.

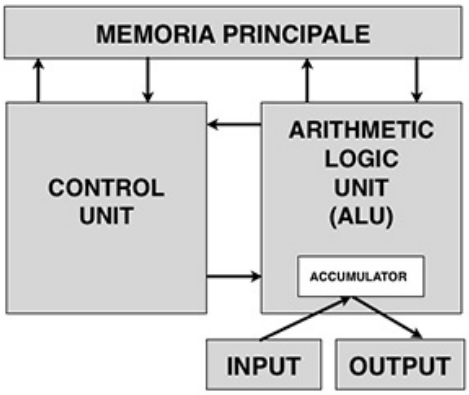



Figura 1.1: Schema funzionale della macchina di Von Neumann. [9]

Il modo di operare di un calcolatore sequenziale è estremamente svantaggioso: molti compiti potrebbero essere assolti più velocemente se fossero ripartiti tra unità diverse. Per costruire una casa il lavoro può essere suddiviso, in modo opportuno, tra gli operai in base alle loro competenze. L'idea più naturale allora è quella di decomporre un problema di dimensione $N$ in $P$ sottoproblemi di dimensione $N/P$ e risolverli contemporaneamente.

[1] G. Fox e P. Messina, *Architetture per i supercalcolatori*, 1987. Citato da Almerico Murli in *Lezioni di Calcolo Parallelo* [1].

[2] Da Wikipedia [7]: John Von Neumann (Budapest, 28 dicembre 1903 - Washington, 8 febbraio 1957), è stato un matematico, fisico e informatico ungherese naturalizzato statunitense.

Per costruire delle architetture parallele si possono, ad esempio, realizzare CPU con più ALU, connettere due o più processori o collegare in rete più calcolatori (*cluster*). Il principio del **Calcolo Parallelo** pone le sue fondamenta proprio sulla base di questa semplice teoria.

## 1.1 Classificazione delle architetture parallele

Una classificazione ormai standard delle architetture parallele è quella ad opera di Flynn[3] (1966)(Figura 1.2). La **tassonomia di Flynn** classifica i sistemi di calcolo a seconda della molteplicità del flusso di "informazione", dove per "informazione" si intende una istruzione o un dato. Il flusso è "singolo" se viene elaborata una sola istruzione o un solo dato, è "multiplo" se vengono elaborate più istruzioni o più dati.

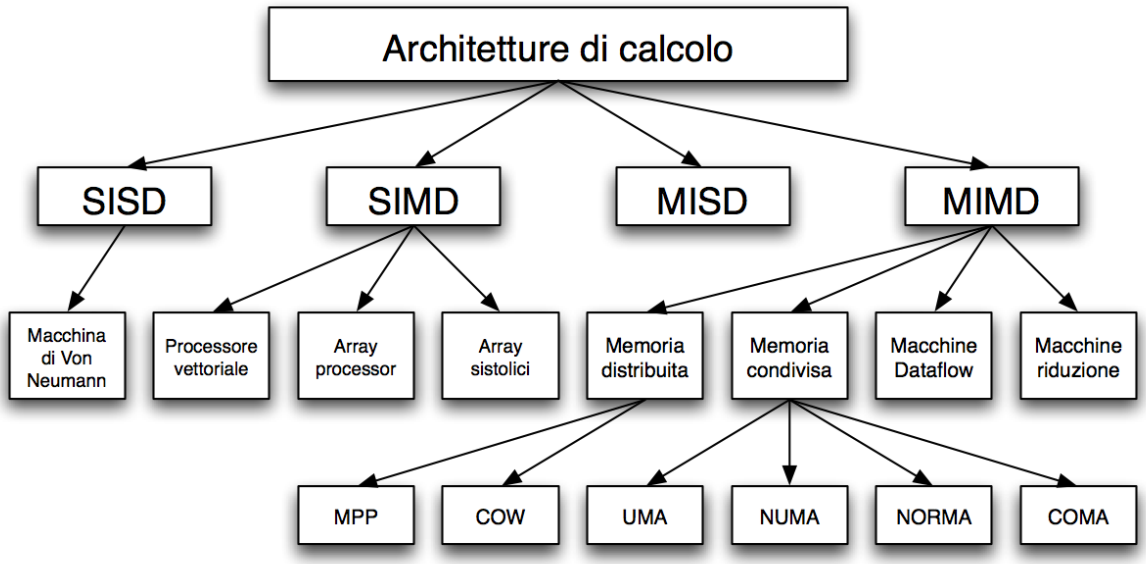



Figura 1.2: Architetture di calcolo. [11]

In base a questa classificazione ogni sistema di calcolo rientra in una di queste categorie:

- **SISD** (*Single Instruction Single Data*): a tale famiglia appartengo le architetture mono processore. Il singolo processore elabora i dati sequenzialmente con un singolo flusso di istruzioni;
- **SIMD** (*Single Instruction Multiple Data*): sono architetture composte da una singola unità di controllo e più ALU che eseguono la stessa istruzione su dati diversi. Tale architettura è utilizzata in alcuni microprocessori moderni;
- **MISD** (*Multiple Instructions Single Data*): si riferiscono a calcolatori in grado di eseguire istruzioni diverse su uno stesso insieme di dati. Vengono considerate architetture anomale e non sono mai state realizzate;[4]

[3]Da Wikipedia [8]: Michael J. Flynn (New York City, 20 maggio 1934) è un ingegnere e informatico statunitense, attualmente professore emerito presso la Stanford University.

[4]La tassonomia di Flynn venne proposta quando ancora non esistevano vere architetture parallele.

- **MIMD** (*Multiple Instructions Multiple Data*): in questa categoria rientrano le architetture costituite da più CPU che consentono l'esecuzione di istruzioni diverse su dati differenti. La maggior parte dei sistemi paralleli appartengono a questa categoria.

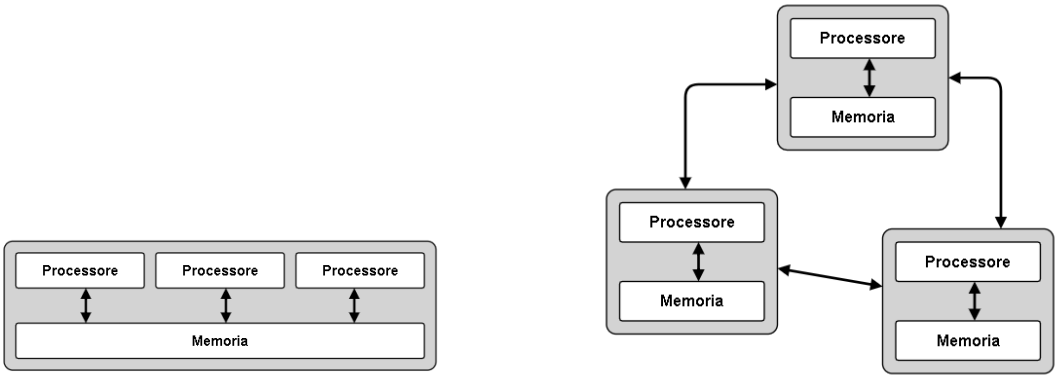
Tornando all'analogia della costruzione di una casa, l'architettura SIMD può essere rapportata a più muratori che eseguono simultaneamente la stessa azione su mattoni diversi (**parallelismo spaziale**), mentre l'architettura MIMD può essere paragonata a più operai che svolgono contemporaneamente azioni diverse su parti diverse (**parallelismo asincrono**).

### 1.1.1 Accesso alla memoria nelle architetture MIMD

Un'ulteriore classificazione per i sistemi MIMD è legata alla modalità di accesso della CPU alla memoria:

- Sistemi a **memoria condivisa** (*shared memory*);
- Sistemi a **memoria distribuita** (*distributed memory*).

Nei sistemi a memoria condivisa, detti anche sistemi **multicore** o **multiprocessore**, le CPU condividono i dati nella stessa memoria (Figura 1.3(a)), mentre nei sistemi a memoria distribuita, detti anche **multicomputer**, ogni processore ha la propria memoria e ognuno di questi può indirizzare solo la propria (Figura 1.3(b)).




(a) (b)

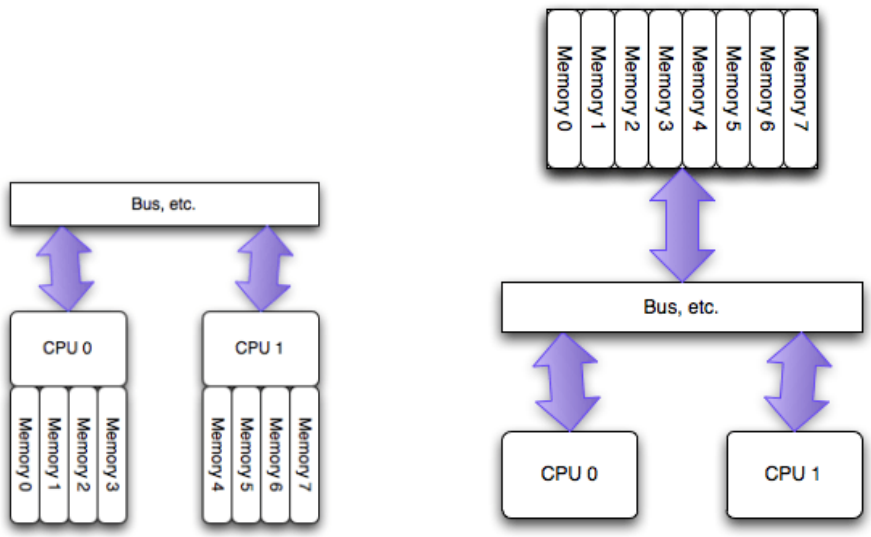
Figura 1.3: (a) Memoria condivisa. (b) Memoria distribuita.

Le architetture a memoria condivisa possono ulteriormente differire per il tempo di accesso in memoria. Nelle architetture con memoria ad accesso non uniforme (**NUMA**, *Non Uniform Memory Access*) un processore può accedere rapidamente alla propria memoria locale e meno velocemente alle memorie degli altri processori o alla memoria condivisa (Figura 1.4(a)). Nelle architetture con memoria ad accesso uniforme (**UMA**, *Uniform Memory Access*) il tempo è equivalente per tutti gli accessi alla memoria (Figura 1.4(b)).

Successivamente alla classificazione di Flynn, si sono realizzate nuove architetture parallele, caratterizzate dall'essere combinazioni delle due tipologie fondamentali, MIMD

e SIMD. Si possono citare le SMP e le DSM, caratterizzate dall'essere un esempio di architetture UMA e NUMA:

- **SMP** (*Symmetric Multi Processor*): sono costituite da più processori che operano indipendentemente accedendo alla stessa memoria globale; ogni processore può avere anche una propria memoria (*cache*), mentre, l'accesso alla memoria condivisa avviene tramite bus o switch. Sono un esempio di architettura shared memory di tipo UMA;
- **DSM** (*Distributed Shared Memory*): la memoria è fisicamente distribuita a livello hardware, ma è globalmente condivisa dal punto di vista software ovvero a livello delle operazioni di accesso (scrittura, lettura) così che all'utente il sistema si presenta con un'unica memoria. Sono un esempio di architettura shared memory di tipo NUMA.




(a) (b)

Figura 1.4: (a) NUMA. (b) UMA. [14]

## 1.2 Il paradigma dello scambio di messaggi

La distribuzione della memoria tra i processori pone il problema di come questi possano sincronizzarsi per risolvere un medesimo problema. Il "***message passing***" è un modello di progettazione che non prevede l'uso di risorse condivise (per questo anche detto *shared nothing*, nessuna condivisione), in cui tutte le comunicazioni avvengano attraverso l'invio di messaggi tra i processori. Il concetto fondamentale alla base del paradigma *message passing* è che i processori, durante l'esecuzione del programma, comunicano tra loro inviando e ricevendo dati.

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

1. Distribuzione dei dati.
2. Esecuzione delle operazioni concorrenti.
3. Combinazione dei risultati.

Per spiegare il funzionamento dello scambio di messaggi è utile esporre la risoluzione di un semplice calcolo matriciale, mettendo in evidenza la strategia alla base dell'introduzione del parallelismo.

## Problema

*Calcolo del prodotto:*

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

*con A matrice quadrata di ordine n ed x ed y vettori di dimensione n.*

Gli algoritmi analizzati nei Paragrafi 1.2.1, 1.2.2, 1.2.3, sono gli stessi algoritmi descritti in [1].

### 1.2.1 L'algoritmo per architetture SISD

L'algoritmo più semplice, ovvero quello sequenziale, prevede il calcolo del vettore $y$ componente per componente:

$$y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j,\ con\ i = 1, ..., n$$

ultimando i prodotti scalari di ciascuna riga di A per il vettore $x$, viene in questo modo calcolata una componente per volta secondo un ordine preciso. Lo pseudo-codice dell'algoritmo è descritto di seguito:

```
procedure matvet(A, x, n, y)
integer n, i
real A(n,n), x(n), y(n)
for i = 0 to n − 1 do
    for j = 0 to n − 1 do
        y_i := y_i + A_ij x_j
    endfor
endfor
return
end matvet
```

Si noti che il calcolo di ciascun prodotto, tra le righe della matrice $A$ e il vettore $x$, può essere portato a termine in modo indipendente.

### 1.2.2 L'algoritmo per architetture MIMD

Per eseguire la stessa operazione sfruttando il parallelismo in un ambiente MIMD a memoria distribuita, è possibile distribuire il calcolo dei prodotti tra i vari processori. Questa distribuzione può essere effettuata in diversi modi.

**Distribuzione per righe.** Le righe della matrice $A$ sono distribuite, a blocchi o ciclicamente, in parti uguali tra tutti i processori: con $p$ processori e $n$ righe a ogni processore saranno assegnate $p/n$ righe, mentre il vettore $x$ viene assegnato a tutti i processori. Ogni processore esegue il prodotto riga per colonna utilizzando i dati residenti nella propria memoria e, terminata la fase di calcolo, i processori si scambieranno i risultati per avere il vettore $y$ risultante nella propria memoria.

**Distribuzione per colonne.** Le colonne della matrice $A$ sono distribuite, a blocchi o ciclicamente, in parti uguali tra tutti i processori: con $p$ processori e $n$ colonne a ogni processore saranno assegnate $p/n$ colonne. Le righe del vettore $x$ vengono distribuite in base alle rispettive colonne assegnate. Tale distribuzione permette di sfruttare al meglio la struttura della matrice $A$ e di rendere gli algoritmi bilanciati. In questo caso i processori, dopo avere effettuato il calcolo, hanno una somma parziale di tutte le componenti del vettore risultato. Per ottenere quindi il risultato finale i due processori devono scambiare tra loro le somme parziali ottenute e sommare le corrispondenti.

Gli algoritmi realizzati con le varie modalità di distribuzione sono molto simili fra loro, perciò ne analizzeremo solo uno nel dettaglio. L'algoritmo in questione è quello a distribuzione per righe con assegnamento a blocchi, con: due processori, una matrice $A$ di ordine 6, $x$ e $y$ vettori di lunghezza 6.

Ogni processore ha il proprio algoritmo e i relativi pseudo-codici sono descritti di seguito:

*Pseudo-codice del processore 0:*

**begin**
% *distribuzione dei dati*
  $A_{loc}$ := **A(:,0:2)**
  $x_{loc}$ := **x(0:5)**
  $y_{loc} := y(0:5)$
% *calcolo delle componenti*
  **for** $i = 0$ **to** $2$ **do**
    $y_{loc}(i) := 0$
    **for** $j = 0$ **to** $5$ **do**
      $y_{loc}(i) := y_{loc}(i) + A_{loc}(i,j)x_{loc}(j)$
    **endfor**
  **endfor**
% *scambio dei risultati*
  **send**$(y_{loc}(0:2), 1)$
  **recv**$(y_{loc}(3:5), 1)$
**end**

*Pseudo-codice del processore 1:*

**begin**
% *distribuzione dei dati*
  $A_{loc}$ := **A(:,3:5)**
  $x_{loc}$ := **x(0:5)**
  $y_{loc} := y(0:5)$
% *calcolo delle componenti*
  **for** $i = 3$ **to** $5$ **do**
    $y_{loc}(i) := 0$
    **for** $j = 0$ **to** $5$ **do**
      $y_{loc}(i) := y_{loc}(i) + A_{loc}(i,j)x_{loc}(j)$
    **endfor**
  **endfor**
% *scambio dei risultati*
  **recv**$(y_{loc}(0:2), 0)$
  **send**$(y_{loc}(3:5), 0)$
**end**

$A_{loc}$, $x_{loc}$, $y_{loc}$ sono variabili locali residenti nella memoria di ciascun processore, *send* e *recv* sono le funzioni utilizzate dai processi per scambiarsi i dati. La funzione *send* ha come argomenti gli elementi da inviare e l'identificativo del processore a cui inviarli. La funzione *recv* ha come argomenti gli elementi da ricevere e l'identificativo del processore da cui riceverli.

Questo modo di programmare, ovvero progettando un algoritmo per ogni processore, viene definito come **MPMD** (*Multiple Program Multiple Data*).

### 1.2.3 L'algoritmo per architetture SIMD

Analizzando gli algoritmi, si può notare che l'unica differenza risiede nei dati su cui ogni processore opera, e che le operazioni da seguire sono le stesse per entrambi i processori. Introducendo opportune variabili è possibile scrivere un solo algoritmo che vada bene per tutti i processori. Tale modello di programmazione prende il nome di **SPMD** (*Single Program Multiple Data*).

Le variabili necessarie per progettare l'algoritmo sono due, e identificano:

- il numero di processori
- l'identificativo del processore

L'inizializzazione di queste variabili (*inizializzazione dell'ambiente*) è definita con l'istruzione

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

la quale restituisce in $p$ il numero dei processori e in *myid* l'identificativo del processore che l'ha invocata.
Non conoscendo a priori le dimensioni locali, l'algoritmo calcola il numero di righe $k = n/p$ della matrice $A$ da assegnare ai processori. Le variabili $k_1$ e $k_2$ indicano le righe del vettore $x$ da assegnare ad ogni processore. Per effettuare gli scambi di dati tra i processori, ciascun processore calcola l'identificativo del processore con cui comunicare, *proccom*, e gli indici dei blocchi che deve inviare e ricevere.

Possiamo riscrivere quindi gli algoritmi per l'architettura MIMD in un unico algoritmo per un'architettura SIMD in questo modo:

**begin**
% *inizializzazione dell'ambiente*
    **loc.init(p,myid)**
%*numero di righe da assegnare a ciascun processore*
    $k := n/p$
%*determinazione degli indici delle righe da distribuire*
$k1 := k \times myid$
$k2 := k1 + k - 1$
% *distribuzione dei dati*
    $A_{loc} :=$ **A(k1:k2,:)**
    $x_{loc} :=$ **x(:)**

$y_{loc} := y(0 : n-1)$
% *calcolo delle componenti di* $y_{loc}$ *relative ai dati acquisiti*
**for** $i = 0$ **to** $k-1$ **do**
$y_{loc}(k * myid + i) := 0$
**for** $j = 0$ **to** $n-1$ **do**
$y_{loc}(k * myid + i) := y_{loc}(k * myid + i) + A_{loc}(i,j)x_{loc}(j)$
**endfor**
**endfor**
% *calcolo dell'identificativo del processore con cui comunicare*
% *e degli indici degli elementi da spedire e ricevere*
**if**$(myid = 0)$ **then**
$proccom := myid + 1$
$l1 := 0;\ l2 := n/2 - 1;\ l3 := n/2;\ l4 := n-1$
**else**
$proccom := myid - 1$
$l1 := n/2;\ l2 := n-1;\ l3 := 0;\ l4 := n/2 - 1$
**endif**
%*scambio delle componenti di* $y_{loc}$ *calcolate*
**send**$(y_{loc}(l1 : l2), proccom)$
**recv**$(y_{loc}(l3 : l4), proccom)$
**end**

## 1.3 I parametri di valutazione del software parallelo

Se $T_1$ è il tempo di esecuzione di un algoritmo su singolo processore e $T_p$ è il tempo di esecuzione su $p$ processori, lo ***speed up*** $S_p$ è definito così:

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

Lo *speed up* misura la riduzione del tempo di esecuzione rispetto all'algoritmo su un processore.

Se si hanno a disposizione $p$ processori, l'obiettivo è impiegare $1/p$ volte il tempo necessario a risolverlo con un solo processore. In base a questa considerazione, viene definito ***speed up ideale***:

$$S_{ideale} = p$$

cioè l'algoritmo parallelo risulta migliore quanto più lo *speed up* è vicino al numero di processori. La differenza tra lo *speed up ideale* e lo *speed up effettivo* viene definita ***Overhead***.

Si definisce efficienza$E_p$ il parametro

$$E_p = \frac{S_p}{p}$$

che fornisce un'indicazione di quanto sia stato utilizzato il parallelismo del calcolatore. Dalla definizione di speed up ed efficienza segue che

$$E_p \leq 1$$

Si definisce quindi l'efficienza ideale:

$$E_{ideale} = 1$$

# Capitolo 2

# Calcolo Parallelo e Distribuito con MATLAB

Numerosi linguaggi di programmazione supportano il calcolo parallelo, ne sono esempi *C*, *Occam*, *Fortran* e *Lisp*. Tuttavia, MATLAB permette la risoluzione di numerosi problemi di calcolo tecnico, in particolare quelli caratterizzati da formulazioni di tipo vettoriale e matriciale, attraverso algoritmi molto più semplici e snelli rispetto a quelli che sarebbero necessari in un programma in linguaggio scalare non interattivo, come quelli citati precedentemente.

## 2.1 Introduzione a MATLAB

MATLAB, ***MAT****rix* ***LAB****oratory* (laboratorio di matrice), è un linguaggio ad alto rendimento per la computazione tecnica, in cui i problemi e le soluzioni sono espressi in notazione matematica familiare. L'elemento di base è la matrice: una semplice variabile numerica è espressa come una matrice quadrata di dimensione 1.

Matlab utilizza due librerie famose per eseguire i calcoli, descritte in maniera esaustiva da Murli [1]:

- **Blas** (*Basic Linear Algebra SubPrograms*): è una libreria dedicata alle operazioni di base del calcolo matriciale. L'idea alla base di questa libreria è quella di sostituire le operazioni scalari con le corrispondenti operazioni vettoriali, in modo da aumentare il rapporto tra il numero di operazioni effettuate e gli accessi ai livelli "*lenti*" della memoria.

- **Lapack** (*Linear Algebra PACKage*): è una libreria dedicata alla risoluzione di problemi quali: sistemi lineari, minimi quadrati, autovalori e valori singolari. Si possono fornire in input matrici generiche dense, a banda, simmetriche, Hermitiane (definite positive) o triangolari. Lapack utilizza le routine di BLAS per svolgere i calcoli.

MathWorks presenta il suo prodotto sul sito ufficiale [16]:

> Milioni di ingegneri e scienziati di tutto il mondo usano MATLAB per analizzare e progettare i sistemi e i prodotti che trasformano il nostro mondo. MATLAB è nei sistemi di sicurezza attivi nelle automobili, nei veicoli spaziali interplanetari, nei dispositivi di monitoraggio dello stato di salute, nelle reti elettriche intelligenti e nelle reti cellulari LTE. È utilizzato per l'apprendimento

automatico, l'elaborazione di segnali, l'elaborazione di immagini, la visione artificiale, le comunicazioni, la finanza computazionale, la progettazione di controllo, la robotica e molto altro.

### 2.1.1 Parallelizzazione implicita: la libreria BLAS

Come accennato precedentemente, BLAS è una libreria di software matematico per l'esecuzione di operazioni di base del calcolo matriciale che ottimizza gli accessi alla memoria. Negli anni, tale libreria è stata sviluppata su più livelli:

**BLAS1 (1974).** Opera su coppie di vettori.

**BLAS2 (1984).** Effettua operazioni di tipo matrice-vettore.

**BLAS3 (1987).** Lavora su coppie di matrici.

Per i calcolatori moderni BLAS3 è la più adatta per compiere calcoli matriciali. Si prenda come esempio il calcolo del prodotto $C = A \cdot B$, con $A$ e $B$ matrici quadrate. BLAS3 effettua tale prodotto partizionando le matrici in $N \times N$ blocchi, ciascuno dei quali ha dimensione $p \times p$ con $p = n/N$, dove $p$ ed $n$ sono rispettivamente il numero di colonne e di righe della matrice. La Figura 2.1 rappresenta la distribuzione delle matrici tra due processori.

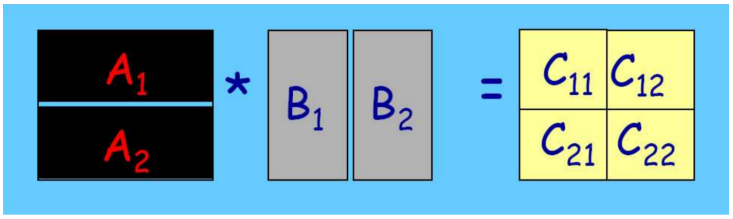



Figura 2.1: Calcolo del prodotto tra matrici quadrate $C = A \cdot B$ in BLAS3. [19]

Si capisce quindi che la parallelizzazione vista nel Capitolo 1.2 è implicitamente realizzata in MATLAB utilizzando la libreria BLAS.

## 2.2 PCT - Il toolbox per il calcolo parallelo

**PCT** (*Parallel Computing Toolbox*) è un insieme di funzioni MATLAB, di alto livello e facile utilizzo, e di costrutti linguistici che consentono di risolvere problemi di calcolo intensivo e grande quantità di dati, utilizzando processori multicore e GPU. Le principali caratteristiche di PCT sono riassunte qui di seguito:

- Cicli *for* in parallelo (***parfor***) per eseguire algoritmi con task in parallelo su più processori.
- Possibilità di eseguire più worker su un sistema multicore.
- Computer cluster e grid support (insieme al toolbox MDCS).
- Esecuzione interattiva e di gruppo di applicazioni in parallelo.
- Array distribuiti e il costrutto ***spmd*** (*single-program multiple-data*) per la gestione di una grande quantità di dati e algoritmi data-parallel.
- Computazione in parallelo sulle GPU usando gli oggetti gpuArray e CUDAKernel.

- Supporta la parallelizzazione automatica per alcune funzioni nei toolbox MATLAB. Infatti, il numero di toolbox e funzioni MATLAB supportate da PCT aumentano ad ogni aggiornamento di MATLAB.

Nei prossimi paragrafi saranno riassunti brevemente i comandi principali del toolbox PCT descritti in [4] e [5].

### 2.2.1 Cicli *for* in parallelo: il costrutto *parfor*

Matlab permette di generare un ***pool*** di unità di lavoro indipendenti, detti ***worker***[1]; in questo modo è possibile eseguire codice in parallelo distribuendo il carico di lavoro tra i worker per velocizzare l'esecuzione. Il modo più semplice per farlo è utilizzando il costrutto ***parfor*** (***par****allel* ***for***) al posto dei classici cicli ***for***. Il numero di cicli che verrà effettuato in parallelo dipende da quanti worker vengono creati, e sono al massimo pari al numero di core (logici) della macchina.

Per "aprire" un pool di worker si utilizza la funzione ***parpool*** (*parallel pool*): il valore restituito è una variabile di tipo *pool* (Figura 2.2). Per "chiuderlo" si utilizza il comando '`delete(gcp)`'.

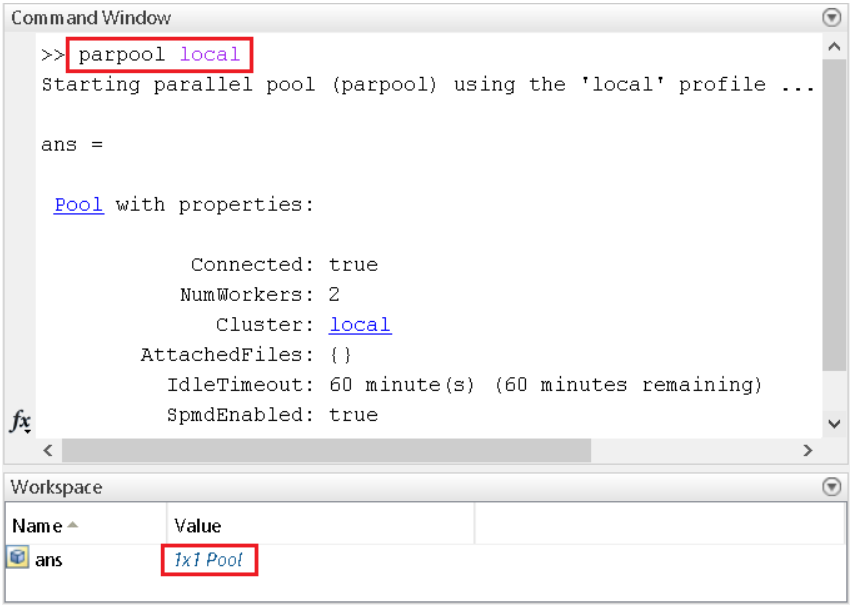



Figura 2.2: Creazione di un pool di worker.

La funzione accetta come parametri in ingresso un profilo e il numero di worker; se non vengono specificati, viene utilizzato il profilo locale (equivalente alla chiamata '`parpool local`') con un numero di worker pari al numero di core (fisici) della macchina. Effettivamente, in ogni comando che prevede un pool di worker, l'allocazione avviene in automatico con il profilo di default se, tale comando, viene eseguito senza l'allocazione di alcun worker in precedenza. Per modificare il profilo si può interagire con la voce del menu **Parallel > Manage Cluster Profiles**.

[1] Si possono definire *worker*, sia *core* fisici che logici. Poichè i core logici non sono argomenti di questa tesi, i termini *worker*,*core* e *processore* assumono lo stesso significato.

Tornando al costrutto *parfor*, la sintassi è la stessa di un normale ciclo *for*

```
parfor [iterazione]
   istruzioni
end
```

Si noti, nella Figura 2.3, come l'utilizzo di un semplice ciclo *parfor* incrementi le prestazioni rispetto al classico ciclo *for*.

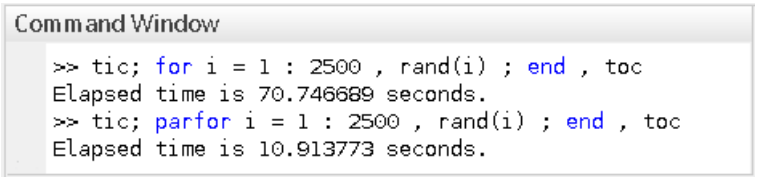



Figura 2.3: Esempio di codice che confronta le prestazioni del *parfor* con quelle del *for*.

Il ciclo *parfor* ha però delle limitazioni:

- le istruzioni eseguite al suo interno devono essere **indipendenti**;
- i valori di iterazione devono essere **interi consecutivi** (Tabella 2.1).

| **Valori di iterazione** | **Validità** |
|---|---|
| `parfor i = 1 :  100` | Valido |
| `parfor i = -20 :  20` | Valido |
| `parfor i = 1 :  2 :  25` | Non valido |
| `parfor i = -7.5 :  7.5` | Non valido |

Tabella 2.1: Esempi di valori iterativi per un ciclo *parfor*.

### 2.2.2 Single Program, Multiple Data: il costrutto *spmd*

L'istruzione ***spmd*** (***s**ingle **p**rogram, **m**ultiple **d**ata*) definisce un blocco di codice da eseguire contemporaneamente su più worker che sono stati riservati tramite il comando *parpool*. L'aspetto "***multiple-data***" (dati multipli) di *spmd* significa che anche se viene eseguito del codice identico su tutti i worker, ogni worker può operare su dati univoci per quel codice, esattamente come spiegato nel Sottoparagrafo 1.2.3. Pertanto, più insiemi di dati possono essere trattati da più worker.

La struttura generale del comando *spmd* è la seguente

```
spmd
   istruzioni
end
```

Facoltativamente, si può anche specificare il numero minimo e massimo di worker da utilizzare per l'esecuzione del codice parallelo, ad esempio "`spmd(3)`" o "`spmd(2,4)`".

In genere, *spmd* viene utilizzato per l'esecuzione simultanea di un programma su più insiemi di dati quando è necessaria la comunicazione o la sincronizzazione tra i worker. Alcuni casi d'uso comuni sono:

- programmi con esecuzione lunga: diversi worker calcolano le soluzioni contemporaneamente;
- programmi che operano su grandi insiemi di dati: i dati vengono distribuiti tra più worker.

A differenza dei cicli *parfor*, ad ogni worker che esegue un'istruzione *spmd* viene assegnato un valore univoco, ***labindex***, che consente di accedere esplicitamente il worker, di specificare il codice da eseguire solo su determinati worker o di accedere a dati unici. Il numero totale di worker che eseguono il blocco in parallelo può essere ottenuto utilizzando il valore di ***numlabs***.
Questo livello di controllo, che non è disponibile in *parfor*, consente di allocare intelligentemente i dati tra i worker, ad esempio per ottenere il bilanciamento del carico. Un esempio di codice che utilizza *spmd* e *labindex* è rappresentato nella Figura 2.4.

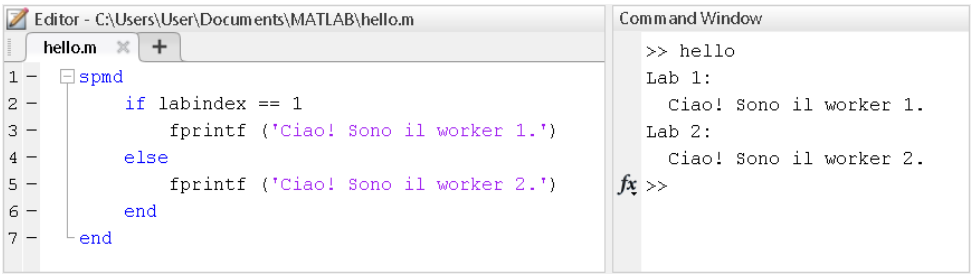



Figura 2.4: Esempio di codiche che utilizza spmd.

I valori calcolati all'interno del corpo dell'istruzione *spmd* vengono restituiti sotto forma di oggetti a struttura mista "*composite*" nel client MATLAB. Un oggetto *composite* contiene riferimenti ai valori memorizzati nei worker remoti di MATLAB e questi valori possono essere recuperati utilizzando l'indice per le matrici a celle, specificando tra `{ }` il numero del worker a cui si vuole accedere, come mostrato nella Figura 2.5.

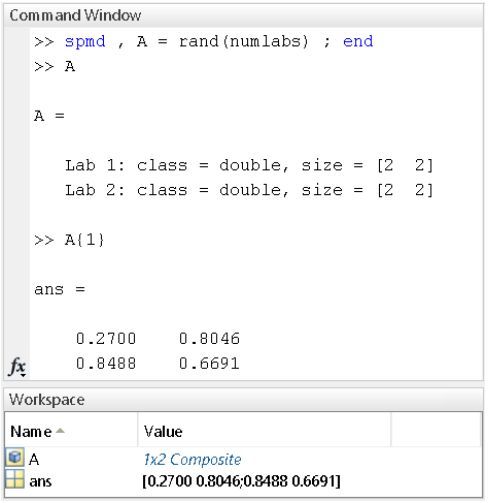



Figura 2.5: Esempio di utilizzo di una variabile *composite*.

Si noti che i worker di un'istruzione *spmd* sono consapevoli degli altri. In altre parole, è possibile controllare direttamente il trasferimento dei dati tra di essi e utilizzare tra loro una matrice codistribuita. Ad esempio, si prenda in considerazione il frammento di codice della Figura 2.6.

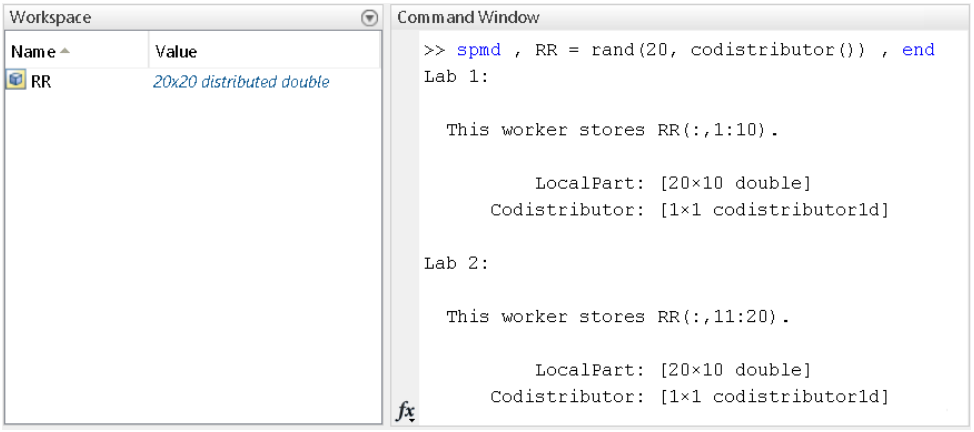



Figura 2.6: Esempio di codice che utilizza una matrice codistribuita all'interno del corpo *spmd*.

Ciò crea una matrice di tipo "***codistributed***" (codistribuita) di dimensione $20 \times 20$, in cui ogni worker ne ottiene un segmento $20 \times 10$. Le matrici codistribuite sono un meccanismo attraverso il quale MATLAB partiziona e distribuisce i dati su più worker.

Oltre alle matrici distribuite, MATLAB fornisce una serie di funzioni di comunicazione per consentire il controllo sull'esecuzione di un processo in parallelo all'interno di blocchi *spmd*. Le funzioni più importanti sono elencate nella Tabella 2.2.

| **Classe** | **Descrizione** |
|---|---|
| labBarrier | Blocca l'esecuzione fino a quando tutti i worker non ricevono questa chiamata. |
| labBroadcast | Invia o riceve dati da tutti i worker. |
| labProve | Verifica se i messaggi sono pronti per essere ricevuti da altri worker. |
| labReceive | Riceve dati da un altro worker. |
| labSend | Invia dati ad un altro worker. |
| labSendReceive | Invia e riceve dati simultaneamente. |
| gcat | Concatenazione globale di una matrice su tutti i worker. La matrice risultante viene duplicata su tutti i worker, a meno che non venga specificato un worker. |

Tabella 2.2: Funzioni per il costrutto spmd.

### 2.2.3 Valutazione di funzioni in parallelo: il costrutto *parfeval*

Come per i cicli *parfor*, i worker di *spmd* sono sessioni MATLAB *headless* (senza interfaccia grafica). Di conseguenza, non possono creare grafici o altre uscite grafiche sul desktop e quindi i comandi MATLAB che aggiornano la GUI non sono consentiti all'interno di istruzioni *spmd*. R2013b (MATLAB 8.2) ha introdotto la funzione ***parfeval*** (**par**allel **f**unction **eval**uation), che esegue funzioni MATLAB su un pool di worker in parallelo, senza bloccare il processo genitore (Desktop). Questo consente di eseguire programmi paralleli asincroni (non bloccanti), a differenza di quelli che utilizzano *parfor* o *spmd* che bloccano l'elaborazione. Inoltre, *parfeval* può utilizzare MATLAB GUI/graphics(plots,waitbar, ecc.), cosa impossibile nei cicli *parfor* e in *spmd*.

La sintassi del costrutto *parfeval* è la seguente:

```
F = parfeval(p,fcn,numout,in1,in2,...)
```

Il comando richiede l'esecuzione asincrona della funzione *fcn* sopra un worker contenuto nel pool *p* (se omesso utilizza il pool corrente), e si aspetta un numero di argomenti di output pari a *numout* e come argomenti di input *in1*, *in2* e così via. La valutazione asincrona di *fcn* non blocca MATLAB. *F* è un oggetto ***parallel.FevalFuture***, i cui risultati vengono ottenuti tramite ***fetchNext(F)***, quando un qualunque worker ha completato la valutazione di *fcn*. La valutazione di *fcn* procede fino alla fine o viene cancellata esplicitamente chiamando ***cancel(F)***. La Figura 2.7 mostra un esempio di codice che utilizza il costrutto.

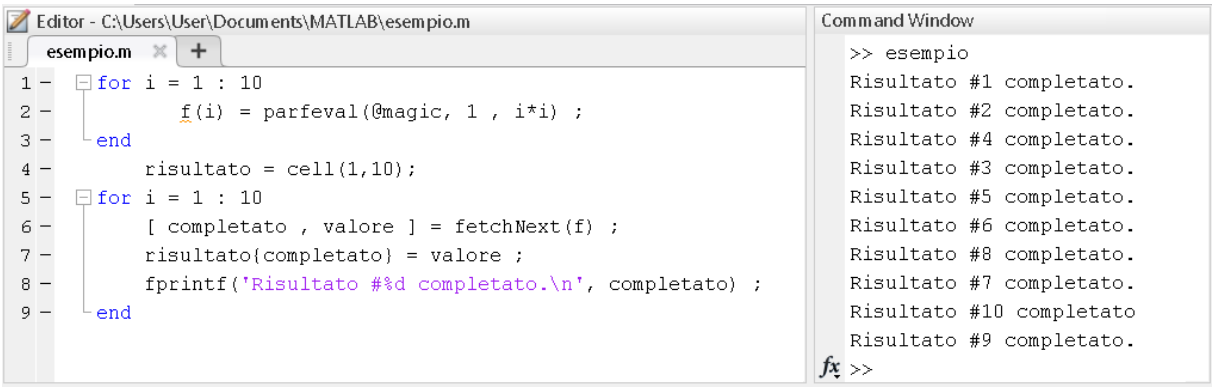



Figura 2.7: Esempio di codice che utilizza *parfeval*.

### 2.2.4 Matrici distribuite e codistribuite

Un'idea generale dietro le matrici distribuite è quella di fornire un livello d'astrazione sui dati distribuiti tra tutti i worker, in modo tale che i dati possano essere visualizzati come una singola matrice nell'area di lavoro di MATLAB. Una matrice distribuita somiglia a una matrice normale nel modo in cui indichiamo e manipoliamo gli elementi, ma nessuno degli elementi esiste sul client. Quindi utilizzando tali matrici, MATLAB esegue in automatico lo scambio di messaggi visto nel Paragrafo 1.2, rendendolo invisibile agli occhi dell'utente.

Per eliminare eventuali confusioni, la differenza tra le matrici codistribuite, già rilevate nella sezione precedente, e le matrici distribuite è solo una questione di prospettiva: una matrice codistributa che esiste sui worker è accessibile dal client (la sessione principale di MATLAB) come un matrice distribuita e viceversa.

Le matrici distribuite vengono create utilizzando la funzione ***distributed*** per distribuire una matrice esistente nell'area di lavoro del client ai worker appartenenti a un *pool* di MATLAB aperto. Si può quindi accedere ai dati come una matrice codistribuita all'interno di un blocco *spmd*:

```
A = distributed ( A ) ;
```

Inoltre, è possibile costruire una matrice distribuita direttamente sui worker senza una matrice preesistente sul lato client utilizzando delle apposite funzioni. Queste funzioni operano allo stesso modo delle loro controparti non distribuite nel linguaggio MATLAB. Vengono descritte brevemente nella Tabella 2.3:

Quando distribuiamo una matrice su un certo numero di worker, MATLAB partiziona la matrice in segmenti e ne assegna uno ad ogni worker. Ogni worker ha accesso a tutti i

| Metodo distribuito | Descrizione |
|---|---|
| `distributed.cell(m,n,...)` | Crea una matrice a celle distribuita. |
| `eye(m,...,class,'distributed')` | Crea una matrice identità distribuita di tipo specifico. |
| `distributed.spalloc(m,n,nzmax)` | Alloca lo spazio per una matrice sparsa distribuita. |
| `distributed.speye(m,n)` | Crea una matrice identità sparsa distribuita. |
| `ones(m,n,...,'distributed')` | Crea una matrice distribuita di uno. |
| `zeros(m,n,...,'distributed')` | Crea una matrice distribuita di zeri. |
| `rand(m,n,...,'distributed')` | Genera una matrice distribuita di numeri pseudo-casuali distribuiti uniformemente. |
| `randn(m,n,...,'distributed')` | Genera una matrice distribuita di numeri pseudo-casuali distribuiti normalmente. |
| `randi(m,n,...,'distributed')` | Genera una matrice distribuita di numeri interi pseudo-casuali distribuiti. |
| `true(m,n,...,class,'distributed')` | Crea una matrice logica distribuita di uno. |
| `false(m,n,...,class,'distributed')` | Crea una matrice logica distribuita di zeri. |

Tabella 2.3: Funzioni per la generazione di matrici distribuite.

segmenti di una matrice codistribuita. Tuttavia, l'accesso al segmento locale è più veloce rispetto ai segmenti remoti (i segmenti che risiedono nella memoria di altri worker), poiché quest'ultimo richiede l'invio e la ricezione dei dati tra i worker (in particolare quando si esegue in un cluster parallelo). È possibile esplicitamente accedere a una porzione locale della matrice codistribuita usando la funzione ***getLocalPart***. L'esempio nella Figura 2.8 crea una matrice codistribuita tra quattro worker e stampa le parti locali di ciascuno.

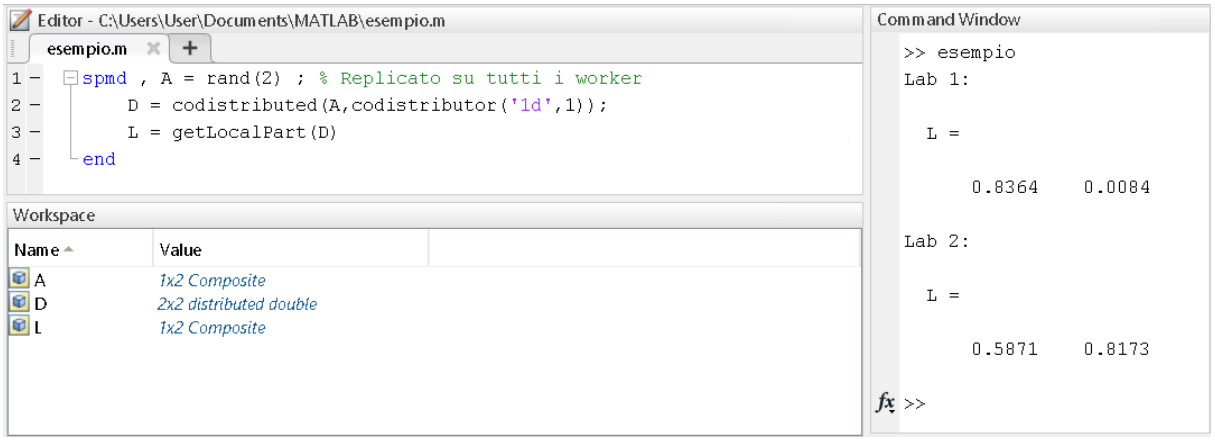



Figura 2.8: Codistribuzione di una matrice.

Nel codice, "`codistributor('1d', 1)`" distribuisce la matrice $A$ lungo la sua prima dimensione (1 per le righe, 2 per le colonne), come visto nel Paragrafo 1.2.

Una matrice distribuita può essere riportata alla sua forma non distribuita attraverso la funzione ***gather***. Utilizzata all'interno del corpo *spmd* raccoglie i segmenti di una matrice, che risiedono su diversi worker, e li unisce in una matrice replicata su tutti i worker o in una singola matrice su un worker; utilizzata all'esterno del corpo *spmd* consolida i risultati dei calcoli paralleli sul lato client.

È possibile creare degli schemi di distribuzione su una dimensione utilizzando il comando "***codistributor1d***", che possono essere riutilizzati nel codice, in questo modo:

| Metodo distribuito | Descrizione |
|---|---|
| `codistributed.cell(m,n,...,codist)` | Crea una matrice a celle codistribuita con uno schema di distribuzione specifico. |
| `eye(m,...,class,'codistributed')` | Crea una matrice identità codistribuita di tipo specifico. |
| `sparse(m,n,codist)` | Crea una matrice sparsa distribuita con uno schema di distribuzione specifico. |
| `codistributed.speye(m,...,codist)` | Crea una matrice identità sparsa codistribuita con uno schema di distribuzione specifico. |
| `ones(m,n,...,'codistributed')` | Crea una matrice codistribuita di uno. |
| `zeros(m,n,...,'codistributed')` | Crea una matrice codistribuita di zeri. |
| `rand(m,n,...,'codistributed')` | Genera una matrice codistribuita di numeri pseudo-casuali distribuiti uniformemente. |
| `randn(m,n,...,'codistributed')` | Genera una matrice codistribuita di numeri pseudo-casuali distribuiti normalmente. |
| `randi(m,n,...,'codistributed')` | Genera una matrice codistribuita di numeri interi pseudo-casuali distribuiti. |
| `true(m,n,...,class,'codistributed')` | Crea una matrice logica codistribuita di uno. |
| `false(m,n,...,class,'codistributed')` | Crea una matrice logica codistribuita di zeri. |

Tabella 2.4: Funzioni per la generazione di matrici codistribuite.

```
schema=codistributor1d(distribuzione,codistributor1d.unsetPartion,dimensione);
matriceDistribuita=codistributed(matrice,schema);
```

Il primo parametro del comando "`codistributor1d`" specifica se la distribuzione deve avvenire per righe (1), come in questo caso, o per colonne (2); il secondo specifica la partizione di distribuzione da utilizzare, in questo caso utilizza quella predefinita; infine, il terzo specifica la dimensione della matrice.

Similmente a quelle distribuite, MATLAB fornisce diverse funzioni per costruire matrici codistribuite su specifici valori, dimensioni e classi "*on-the-fly*". Queste funzioni distribuiscono la matrice risultante tra i lavoratori utilizzando uno schema di distribuzione desiderato; la Tabella 2.4 ne mostra alcuni.

### 2.2.5 Sviluppo parallelo interattivo: *pmode*

La funzione ***pmode*** è molto simile a *spmd* e non ci dovrebbe essere una grande differenza tra l'esecuzione del codice in *pmode* o *spmd*. Il comando *pmode* consente di lavorare in modo interattivo su un processo parallelo che viene eseguito simultaneamente su più worker. I comandi scritti nel prompt di *pmode* nella **Parallel Command Window** vengono eseguiti su tutti i worker contemporaneamente (Figura 2.9).

Ogni worker esegue i comandi nel proprio spazio di lavoro e sulle proprie variabili. Le variabili possono essere trasferite dal client MATLAB ai worker e viceversa. Tutte le funzioni di comunicazione supportate in blocchi *spmd* (*labindex*, *labSend*, *labReceive*, ecc.) possono essere utilizzate nelle sessioni *pmode*. Tuttavia, la differenza principale tra *spmd* e *pmode* è che la seconda non permette di avere un pieno controllo sull'esecuzione parallela, come invece accade con *spmd*. Quando si esce dalla sessione *pmode*, il processo aperto viene cancellato efficacemente, i processi dei worker vengono chiusi e tutte le informazioni

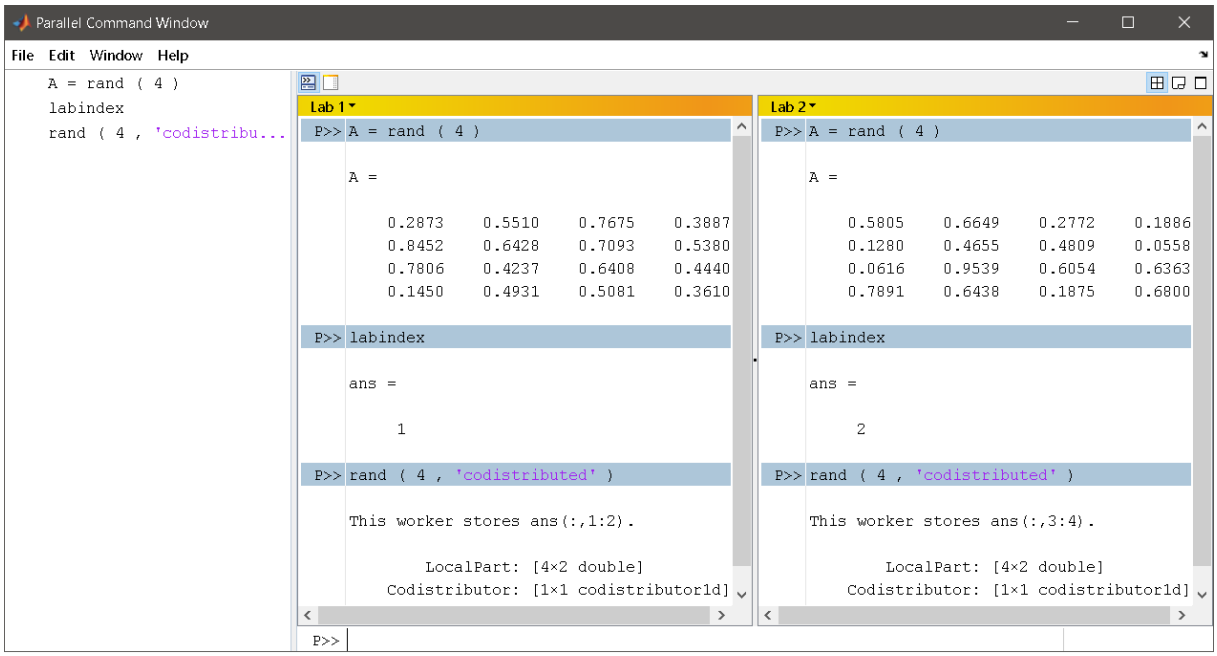



Figura 2.9: Esempio di comandi nella Parallel Command Window.

e i dati sui worker vengono persi. L'avvio di una sessione *pmode* inizia sempre da uno stato pulito.

L'elenco dei comandi principali di pmode è:

- `pmode start [profilo] [numeroworker]`
  Avvia la sessione *pmode*, specificando eventualmente il profilo PCT da utilizzare e il numero di worker.

- `pmode exit`
  Termina la sessione *pmode*.

- `pmode client2lab clientvar workers [workervar]`
  Copia la variabile `clientvar` dal client MATLAB nella variabile `workervar` sui worker specificati da `workers`.

- `pmode lab2client workervar idworker [clientvar]` Copia la variabile `workervar` dal worker specificato in `idworker` nella variabile `clientvar` sul client MATLAB.

- `pmode cleanup profilo`
  Elimina tutti i processi paralleli creati con *pmode* per l'utente corrente, inclusi i processi attualmente in esecuzione.

# Capitolo 3

# Parallelizzazione in MATLAB: prodotto matrice per vettore

In questo capitolo verrà analizzata l'implementazione in MATLAB dell'algoritmo di tipo SPMD illustrato nel Sottoparagrafo 1.2.3. L'algoritmo è illustrato nella Figura 3.1.

Ax.m

```
1   function y = Ax ( A , x )
2   n = length ( x ) ;
3   spmd
4       % Determinazione del numero di righe da assegnare a ciascun processore
5       % e degli indici delle righe da distribuire
6       k = int64 ( n / numlabs ) ;
7       k1 = 1 + ( k * ( labindex - 1 ) ) ;
8       if not ( labindex == numlabs ) , k2 = k1 + k - 1 ; else , k2 = n ; end
9       % Distribuzione dei dati
10      Aloc = A ( k1 : k2 , : ) ;
11      xloc = x ;
12      yloc = zeros ( n , 1 ) ;
13      % Calcolo delle componenti
14      yloc = Aloc * xloc ;
15      % Scambio delle componenti
16      yloc = gcat ( yloc , 1 ) ;
17  end
18  y = yloc{1} ;
```

Figura 3.1: Implementazione in MATLAB della parallelizzazione del prodotto $y = Ax$.

L'algoritmo risulta molto più semplice da implementare in MATLAB, perché la maggior parte delle operazioni vengono svolte in automatico; non sono necessari né i cicli *for*, né lo scambio di messaggi esplicito.

## 3.1 Calcolo delle prestazioni

Per calcolare le prestazioni dell'algoritmo implementato, è stato eseguito un test al variare della dimensione della matrice $A$. I risultati sono evidenziati nella Figura 3.2.
Come si può notare le prestazioni della versione parallela sono molto più basse rispetto alle aspettative.

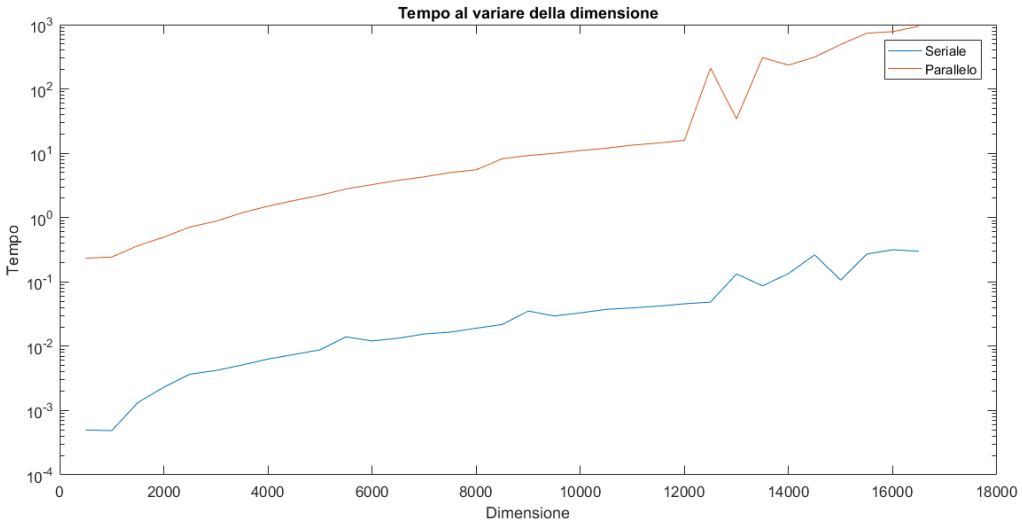



Figura 3.2: Test al variare della dimensione.

# Capitolo 4

# Parallelizzazione del metodo di Jacobi

Il **metodo di Jacobi** è un metodo iterativo per la risoluzione di sistemi lineari, un metodo cioè che genera, a partire da un vettore iniziale $x^{(0)}$, una successione di vettori $x^{(k)}$, $k = 0, 1, ...$, che , sotto opportune ipotesi, converge alla soluzione del problema.
I metodi iterativi risultano convenienti per matrici di grandi dimensioni, specialmente se strutturate o sparse. Infatti, mentre un metodo diretto opera modificando la matrice del sistema, spesso alterandone la struttura e aumentando il numero di elementi non nulli, un metodo iterativo non richiede una modifica della matrice del sistema e, in taluni casi, nemmeno la sua effettiva memorizzazione.[1]

## 4.1 Costruzioni di metodi iterativi lineari

Scrivendo la matrice $A$ come differenza $A = P - N$, dove $P$ è una matrice invertibile (ovvero non singolare, con determinante non nullo), allora la soluzione $x$ di $Ax = b$ risolve anche le equazioni

$$Px = Nx + b$$

$$x = P^{-1}(Nx + b).$$

Partendo da un qualunque vettore $x^{(0)}$, si può allora costruire una successione di vettori $x^{(k)}$ come

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b = Bx^{(k)} + f, \tag{4.1}$$

con $B = P^{-1}N$ e $f = P^{-1}b$. Se questa successione converge ad un vettore $x$, allora $Ax = b$. Una condizione sufficiente per la convergenza è data da $\|P^{-1}N\| < 1$, dove $\|\cdot\|$ è una qualsiasi norma matriciale consistente, mentre una condizione necessaria e sufficiente è $\rho(P^{-1}N) < 1$. Naturalmente, perché il metodo risulti operativo è necessario che la matrice $P$ sia più semplice da invertire di $A$. Per questo viene utilizzato il seguente *splitting additivo*

$$A = D - E - F,$$

dove

$$D_{ij} = \begin{cases} a_{ii}, & i = j \\ 0 & i \neq j \end{cases}, \quad E_{ij} = \begin{cases} -a_{ij}, & i > j \\ 0 & i \leq j \end{cases}, \quad F_{ij} = \begin{cases} -a_{ij}, & i < j \\ 0 & i \geq j \end{cases},$$

[1] Giuseppe Rodriguez, *Algoritmi Numerici* [3, cap. 5]

ovvero

$$A = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & & \\ & \ddots & \\ -a_{ij} & & 0 \end{bmatrix} - \begin{bmatrix} 0 & & -a_{ij} \\ & \ddots & \\ & & 0 \end{bmatrix}$$

dove il metodo di Jacobi corrisponde alla scelta

$$P = D,\ N = E + F. \tag{4.2}$$

Possiamo quindi riscrivere il metodo come

$$Dx^{(k+1)} = b + Ex^{(k)} + Fx^{(k+1)},$$

o, in coordinate,

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

per il calcolo di $x^{(k+1)}$ a partire da quelle di $x^{(k)}$ in qualsiasi ordine e indipendentemente l'una dall'altra. In altri termini, il metodo è **parallelizzabile**. Una volta calcolata la matrice $B$, per ogni iterazione il calcolo di ognuna delle $n$ componenti richiede $n - 1$ moltiplicazioni e $n - 1$ somme. La Figura 4.1[2] mostra, in modo schematico, quali componenti dei vettori $x^{(k)}$ e $x^{(k+1)}$ vengono utilizzate per il calcolo di $x_i^{(k)}$.

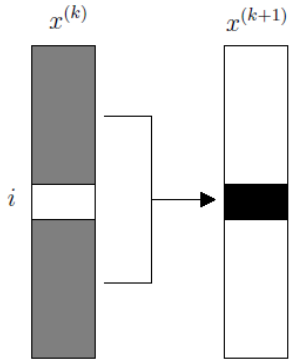



Figura 4.1: Schema dell'iterazione del metodo di Jacobi.

## 4.2 Gli algoritmi in MATLAB

Per valutare le prestazioni del metodo di Jacobi sono state realizzate due versioni. La **versione seriale** è rappresentata nella Figura 4.2, mentre la **versione parallela** è rappresentata nella Figura 4.3.

Si noti che gli algoritmi divergono solamente nell'utilizzo del costrutto *spmd*, presente nella versione parallela dell'algoritmo.

[2]La figura è stata creata su Paint, ma è stata realizzata secondo la Figura 5.1 di [3, cap. 5, pag. 122]

jacobi_ser.m

```
function [ xkp1 , k ] = jacobi_ser ( A , b , x0 , tau , MAX )
% Controlli
if nargin < 2 , error ( 'Dati insufficienti.' ) , end
[ m , n ] = size ( A ) ; [ r , s ] = size ( b ) ;
if not ( m == n ) , error ( 'La matrice deve essere quadrata.' ) , end
if not ( n == r ) , error ( 'Le dimensioni non corrispondono.' ) , end
if not ( s == 1 ) , error ( 'Le dimensioni di b non sono adatte.' ) , end
% Valori predefiniti
if nargin < 3 || isempty ( x0 ) , x0 = zeros ( n , 1 ) ; end
if nargin < 4 || isempty ( tau ) , tau = 1e-6 ; end
if nargin < 5 || isempty ( MAX ) , MAX = 200 ; end
% Calcolo delle matrice B e della vettori necessari per le iterazioni
p = diag ( A ) ; N = diag ( p ) - A ; B = N ./ p ; f = b ./ p ;
% Inizializzazione variabili per il ciclo
k = 0 ; flag = 1 ; xkp1 = x0 ;
% Inizio delle iterazioni
while flag
    k = k + 1 ;
    xk = xkp1 ;
    xkp1 = B * xk + f ;
    flag = ( norm ( xk - xkp1 ) >= tau * norm( xkp1 ) ) && ( k < MAX ) ;
end
if k >= MAX , warning ( 'Superato il numero massimo di iterazioni' ) , end
```

Figura 4.2: Versione seriale del metodo di Jacobi.

### 4.2.1 L'algoritmo parallelo

La versione parallela del metodo di Jacobi utilizza il costrutto *spmd* e le matrici codistribuite.

Per effettuare il prodotto della matrice $B$ con il vettore $x^k$, risultava scomodo utilizzare la funzione "$Ax.m$" vista nel Capitolo 3. Ogni volta che tale funzione viene chiamata, la matrice $B$ viene nuovamente distribuita tra i processori.
Risulta molto più semplice in questo caso utilizzare le matrici codistribuite, che eseguono in automatico la distribuzione delle righe della matrice e, grazie alla funzione *gather*, permettono di memorizzare facilmente il vettore $x^{(k+1)}$ in ogni processore.

## 4.3 Calcolo delle prestazioni

Per calcolare le prestazioni del software parallelo sono stati effettuati diversi test, variando:

- la dominanza diagonale della matrice $A$;
- il numero di processori;
- la dimensione della matrice $A$;
- la sparsità della matrice $A$.

I test sono stati effettuati con matrici random, generate tramite la funzione *rand* di MATLAB.

jacobi_par.m

```
function [ xkp1 , k ] = jacobi_par ( A , b , x0 , tau , MAX )
% Controlli
if nargin < 2 , error ( 'Dati insufficienti.' ) , end
[ m , n ] = size ( A ) ; [ r , s ] = size ( b ) ;
if not ( m == n ) , error ( 'La matrice deve essere quadrata.' ) , end
if not ( n == r ) , error ( 'Le dimensioni non corrispondono.' ) , end
if not ( s == 1 ) , error ( 'Le dimensioni di b non sono adatte.' ) , end
% Valori predefiniti
if nargin < 3 || isempty ( x0 ) , x0 = zeros ( n , 1 ) ; end
if nargin < 4 || isempty ( tau ) , tau = 1e-6 ; end
if nargin < 5 || isempty ( MAX ) , MAX = 200 ; end
% Calcolo delle matrici e dei vettori necessari per le iterazioni
P = diag ( A ) ; N = diag ( P ) - A ; B = N ./ P ; f = b ./ P ;
% Inizializzazione variabili.
k = 0 ; flag = 1 ;
% Distribuzione dei dati.
spmd
    % Creazione schemi di distribuzione
    co_array = codistributor1d (1,codistributor1d.unsetPartition,[m,1]) ;
    co_matrix = codistributor1d (1,codistributor1d.unsetPartition,[m,n]) ;
    % Le matrici e i vettori vengono distribuiti secondo gli schemi creati
    B = codistributed ( B , co_matrix ) ;
    f = codistributed ( f , co_array ) ;
    xkp1 = codistributed ( x0 , co_array ) ;
    % Inizio delle iterazioni
    while flag
        k = k + 1 ;
        xk = gather ( xkp1 ) ;
        xkp1 = B * xk  + f ;
        flag = ( norm (xk - xkp1) >= tau * norm (xkp1) ) && (k < MAX) ;
    end
end
k = k{1} ; %Converte l'oggetto composite in una normale variabile
if k >= MAX , warning ( 'Superato il numero massimo di iterazioni' ) , end
xkp1 = gather ( xkp1 ) ; %Converte array codistribuito in array normale
```

Figura 4.3: Versione parallela del metodo di Jacobi.

### 4.3.1 Prestazioni al variare della dominanza diagonale

Per assicurare la convergenza del metodo di Jacobi, la matrice $A$ del sistema deve risultare a predominanza diagonale stretta. L'obbiettivo del test è comprendere come la dominanza diagonale di una matrice possa influire sulla velocità di calcolo, e di poter scegliere un valore specifico per tutti i test successivi. Nella Figura 4.4 è rappresentato il codice utilizzato nei test per generare una matrice diagonalmente dominante.[3]

esempio.m

```
n = 10 ;
A = rand(n);
A = A - diag(diag(A));      % Matrice senza diagonale
s = abs(A) * ones(n,1);     % Somme riga
k = 2 ;                     % Fattore di predominanza
A = A + k * diag(s);        % Matrice diag. dominante
```

Figura 4.4: Codice per creare una matrice test diagonalmente dominante in MATLAB.

[3]Il codice è descritto in [3].

Quello che si evince dalla Figura 4.5 è che al variare del fattore di predominanza le prestazioni degli algoritmi rimangono, sostanzialmente, invariate. Per effettuare i test successivi è quindi irrilevante generare matrici con un fattore di predominanza maggiore di 2.

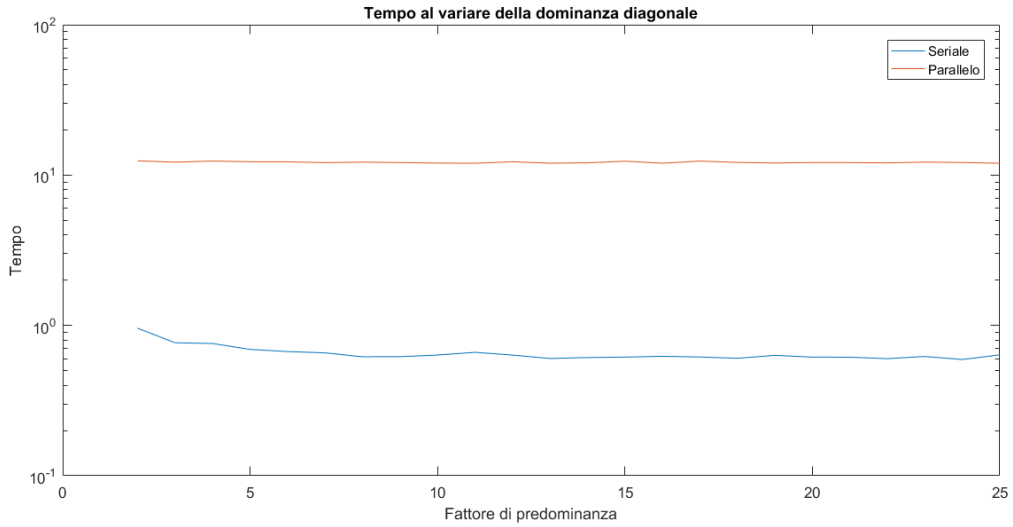



Figura 4.5: Test sulla dominanza diagonale. Il test è stato effettuato su una matrice di dimensione 10.000 con 12 processori.

### 4.3.2 Prestazioni al variare del numero di processori

Dalla Figura 4.6 si comprende che aumentando il numero di processori le prestazioni, effettivamente, calano. Questo perché i processori perdono più tempo a distribuire i dati (funzione *codistributed*) e successivamente a ricomporli (funzione *gather*), piuttosto che a eseguire i calcoli.
Dopo aver effettuato il calcolo sui segmenti del vettore $x^{(k+1)}$, tali segmenti vengono poi concatenati in un vettore unico, che viene memorizzato su ogni processore tramite la funzione *gather*. Più processori implicano più messaggi da scambiare.

Calcolando lo *speed up* come nell'Equazione 1.1, si ottengono dei risultati negativi. Tali risultati sono mostrati nella Figura 4.7.

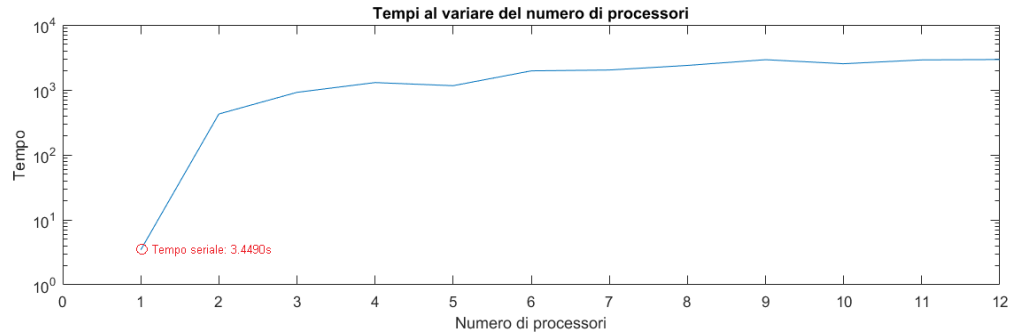



Figura 4.6: Test sul numero di processori. I test sono stati effettuati su una matrice di dimensione 25.000.

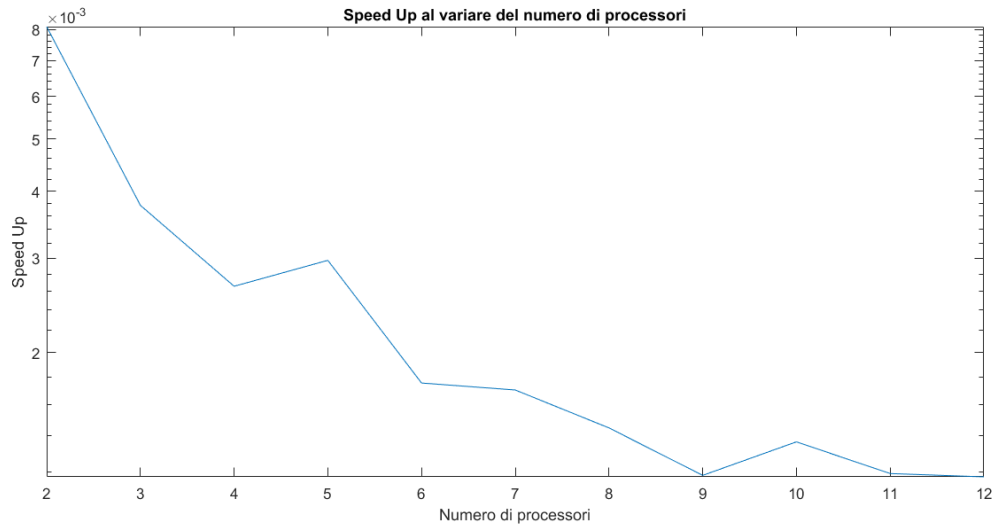



Figura 4.7: Speed up negativo. Le prestazioni calano all'aumentare dei processori.

### 4.3.3 Prestazioni al variare della dimensione della matrice

Nuovamente i risultati del test, mostrati nella Figura 4.8, evidenziano come all'aumentare dei processori, anche se di poco, le prestazioni calano. Inoltre è possibile notare come la differenza di tempo tra i due algoritmi, che inizialmente è costante, peggiora all'aumentare della dimensione della matrice. Più sono i processori, prima peggiora l'andamento del tempo.

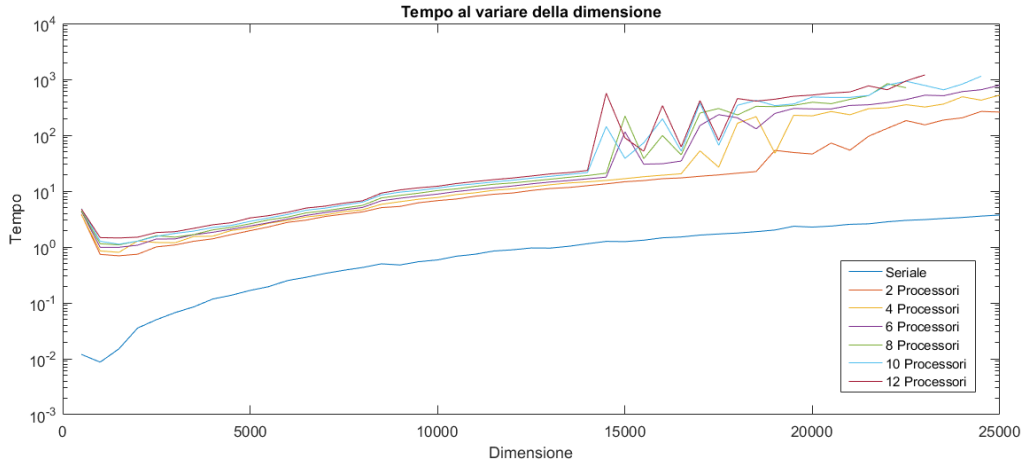



Figura 4.8: Test sulla dimensione della matrice.

I risultati sono molto simili a quelli ottenuti nel Paragrafo 3.1.

### 4.3.4 Prestazioni al variare degli elementi nulli della matrice

Una matrice sparsa è una matrice in cui la maggior parte degli elementi è uguale a zero. Tale caratteristica permette di occupare meno spazio in memoria e di rendere i calcoli più veloci.

I metodi iterativi risultano convenienti, rispetto ai metodi diretti, per matrici sparse; per questo è stato fatto un test al variare della sparsità della matrice. Per generare una matrice sparsa random è stato utilizzato il comando *sprand*, in cui viene specificata una densità della matrice che varia da 0 a 1. I risultati del test sono mostrati nella Figura 4.9.

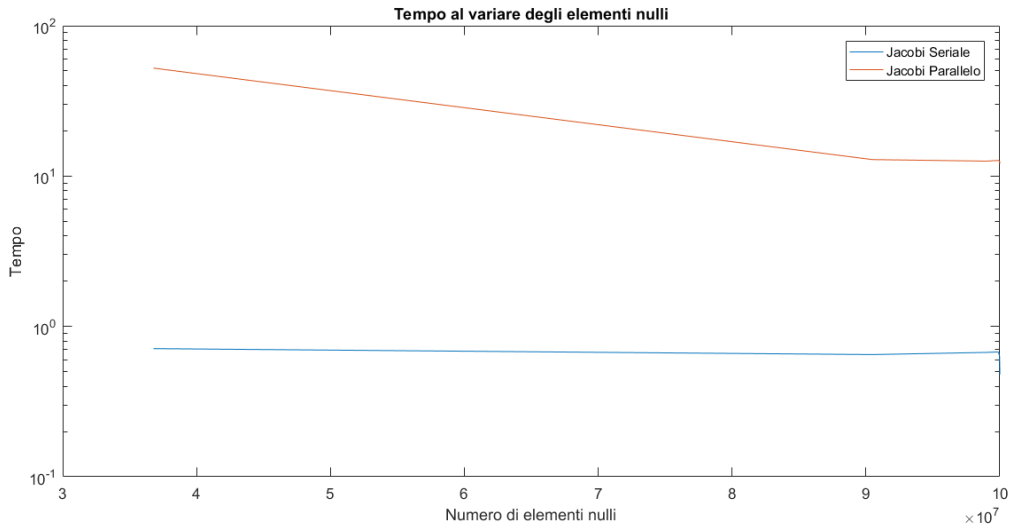



Figura 4.9: Test al variare della sparsità della matrice.

Neanche in questo caso le prestazioni della parallelizzazione risultano positive.

# Capitolo 5

# Conclusioni

Dai test effettuati si può dire che parallelizzando algoritmi matriciali in MATLAB, tramite l'utilizzo del toolbox PCT, si perde efficienza. Lavorando con matrici e vettori si ottengono prestazioni migliori lasciando che sia il software a gestire la parallelizzazione. Si può notare infatti che l'unico caso in cui la parallelizzazione ha avuto successo, è quello della Figura 2.3. Questo perché all'interno del ciclo non sono stati gestiti calcoli matriciali.
In definitiva, si può notare come MATLAB sia uno strumento di calcolo molto potente ed efficiente, con una forte parallelizzazione implicita, che trae vantaggio dalle architetture parallele utilizzando un modello a memoria condivisa, quindi senza perdere tempo a trasferire i dati.

# Bibliografia


[1] Almerico Murli, *Lezioni di Calcolo Parallelo*, Liguori, 2006.

[2] Pasquale De Luca, *Il Calcolo Parallelo*, 23 Giugno 2015.

[3] Giuseppe Rodriguez, *Algoritmi Numerici*, Pitagora Editrice Bologna, 2008.

[4] Y.M. Altman, *Accelerating MATLAB® Performance: 1001 tips to speed up MATLAB programs*, CRC Press, 2014.

[5] MathWorks, *Parallel Computing Toolbox™ User's Guide*, 2016.

[6] Wikipedia, *Analisi numerica*, `https://it.wikipedia.org/wiki/Analisi_numerica`, 27-06-2017

[7] Wikipedia, *John von Neumann*, `https://it.wikipedia.org/wiki/John_von_Neumann`, 08-07-2017

[8] Wikipedia, *Michael J. Flynn*, `https://it.wikipedia.org/wiki/Michael_J._Flynn`, 08-07-2017

[9] Pronto Intervento Apple , *L'informatica semplice: cosa fa funzionare un calcolatore*, `http://www.prontointerventoapple.it/informatica_semplice.html` , 08-07-2017

[10] daCREMa, *Struttura di una CPU*, `http://www.dacrema.com/Informatica/CPU_struttura.htm`, 08-07-2017

[11] Wikipedia, *Tassonomia di Flynn*, `https://it.wikipedia.org/wiki/Tassonomia_di_Flynn`, 08-07-2017

[12] Mauro Ennas, *Breve introduzione al calcolo parallelo*, 1998/2013, `http://www.mauroennas.eu/ita/phocadownload/report/004_Note_sul_calcolo_parallelo.pdf`, 08-07-2017

[13] Valeria Cardellini, *Classificazione delle Architetture Parallele*, 2009/2010, `http://www.ce.uniroma2.it/courses/sd0910/lucidi/ClassArchPar.pdf`, 27-06-2017

[14] Luigi Rosa, *Architettura UMA/NUMA*, 29-09-2010, `https://siamogeek.com/2010/09/architettura-umanuma/`, 08-07-2017

[15] Giuseppe Ciaburro, *MANUALE MATLAB*, `httphttps://siamogeek.com/2010/09/architettura-umanuma/://www.ciaburro.it.`, 06-07-2017

[16] MathWorks, *Presentazione*, https://it.mathworks.com/products/matlab.html, 08-07-2017

[17] MathWork, *Documentation*, https://it.mathworks.com/help/matlab/index.html, 08-07-2017

[18] Wikipedia, *Metodo di Jacobi*, https://it.wikipedia.org/wiki/Metodo_di_Jacobi, 09-07-2017

[19] Maria Rosaria Russo, *Panoramica sulle librerie BLAS, LAPACK e ATLAS*, http://www.math.unipd.it/~mrrusso/Didattica/ESE_CN_Informatica/LibrerieALN.pdf, 10-07-2017

# Appendice

## Informazioni sulla parte pratica

- **Sistema operativo**: Ubuntu, Windows 10;
- **Linguaggi**: MATLAB;
- **Browsers**: Google Chrome, Mozilla Firefox;
- **IDE/editor**: MATLAB;
- **Linee di codice**: 1800;
- **Valutazione tempo di sviluppo**: 120 ore-uomo.