

# UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED  
ELETTRICA E MATEMATICA APPLICATA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA  
(LM-32)



REPORT

## **HPC: “*LU Factorization - Linear Systems*”** **(6 CFU)**

**CANALE I-Z**

### **Esaminando**

Antony Storti  
Matr. 0622702353  
[a.storti2@studenti.unisa.it](mailto:a.storti2@studenti.unisa.it)

### **Docente**

Prof. Francesco Moscato  
[fmoscato@unisa.it](mailto:fmoscato@unisa.it)

---

**Anno Accademico 2023 – 2024**

# INTRODUZIONE

Le operazioni sulle matrici sono essenziali nel calcolo scientifico. Algoritmi efficienti per manipolare le matrici sono quindi di notevole interesse pratico. Il *calcolo numerico* è una branca della matematica che si occupa dello sviluppo e dell'applicazione di algoritmi e metodi numerici per risolvere problemi matematici complessi. Mentre la matematica tradizionale si basa principalmente su metodi analitici che conducono a soluzioni esatte o formule chiuse, il calcolo numerico si concentra sull'uso di algoritmi e approssimazioni per ottenere risultati numerici approssimati.

## Perché il Calcolo Numerico?

La gran parte dei problemi matematici che provengono dalle diverse branche applicative delle discipline scientifiche e dal mondo dell'industria hanno grandi dimensioni, cioè coinvolgono un numero elevato di incognite. Anche qualora detti problemi fossero risolubili elementarmente, il tempo richiesto per determinarne la soluzione sarebbe troppo elevato. Pensiamo ai calcoli che deve fare un computer di bordo di un jet militare, per mantenere la rotta, vi è l'assoluta necessità di farlo nel minor tempo possibile. I jet utilizzano sistemi di navigazione inerziale che si basano su sensori di accelerazione e giroscopi per stimare la posizione, la velocità e l'orientamento del velivolo. Gli algoritmi numerici devono elaborare continuamente i dati provenienti da questi sensori per mantenere una stima accurata della posizione e dell'orientamento, consentendo al jet di seguire la rotta desiderata. Nella pratica il tutto si riduce nell'impostare e risolvere sistemi di equazioni. La potenza di calcolo offerta dai calcolatori moderni è innegabile, consentendo la risoluzione di problemi di grandi dimensioni in tempi relativamente brevi.

Tuttavia, questa potenza di calcolo è vincolata dalla presenza di una memoria finita, il che comporta l'adozione di un'aritmetica di macchina. Quest'ultima presenta due svantaggi principali:

- la rappresentazione finita dei numeri reali introduce inevitabili errori di arrotondamento. I calcolatori utilizzano una rappresentazione a virgola mobile, consentendo la precisione solo di un numero limitato di numeri reali. Gli altri numeri, se non appartenenti a questa categoria, vengono approssimati attraverso i cosiddetti "numeri di macchina". L'errore di rappresentazione dipende dalla specifica macchina e dal software utilizzato.

Per esempio, in MATLAB, l'errore di rappresentazione è approssimativamente di  $2 \times 10^{-16}$ .

- le operazioni elementari di addizione, sottrazione, moltiplicazione e divisione possono introdurre errori, spesso dello stesso ordine di grandezza degli errori di rappresentazione. Anche se queste operazioni sono eseguite in modo efficiente, è importante riconoscere che possono non essere sempre eseguite con precisione assoluta, soprattutto in contesti di calcoli numerici intensivi.

Quanto ivi detto è un annoso problema che rientra nel nome di **stabilità numerica**.

Nell'ambito della risoluzione efficiente di sistemi lineari, la **Decomposizione LU** si configura come una metodologia cruciale, poiché poco sensibile agli errori. Essa si colloca al centro di numerose applicazioni scientifiche e ingegneristiche, come la risoluzione di equazioni differenziali, l'ottimizzazione e la simulazione di sistemi complessi. Tuttavia, con l'aumentare delle dimensioni delle matrici coinvolte, la computazione tradizionale della decomposizione LU mostra limiti in termini di tempo di esecuzione.

Il presente progetto si propone di affrontare questa sfida attraverso lo sviluppo di un algoritmo parallelo dedicato alla LU Decomposition. L'obiettivo principale è migliorare significativamente le prestazioni computazionali, sfruttando l'architettura parallela di moderni processori *multicore* o usando l'accelerazione prestazionale offerta dalle GPU.

La scelta di un approccio parallelo si giustifica nell'esigenza di ottimizzare il tempo di esecuzione, considerando la crescente disponibilità di risorse computazionali parallele. L'implementazione di un algoritmo parallelo per la fattorizzazione LU permette di accelerare la soluzione di sistemi lineari di grandi dimensioni, ma apre anche le porte a un'ottimizzazione su larga scala nelle applicazioni che richiedono operazioni matriciali complesse.

Nel corso di questa relazione, esaminerò dettagliatamente il processo di progettazione e implementazione dell'algoritmo parallelo per la LU Decomposition. Analizzerò le strategie utilizzate per suddividere il problema in sotto-task paralleli, minimizzando le comunicazioni e sfruttando al meglio le potenzialità di calcolo distribuito. Inoltre, valuterò le performance ottenute attraverso test su matrici di dimensioni crescenti e confronti con implementazioni sequenziali di riferimento.

Nella presente relazione, per questioni di discorsività, non verrà incluso il file sorgente: si rimanda alla visione dello stesso all'interno della relativa cartella.

## PROBLEMA AFFRONTATO <sup>[1]</sup>

Nel seguente paragrafo si descriverà il funzionamento generale della Decomposizione LU per poter comprenderne la relativa implementazione in linguaggio C. La trattazione sarà informale, siccome, tale relazione è da intendersi come un resoconto dell'attività progettuale. Per una trattazione rigorosa e formale dell'argomento si rimanda ai riferimenti bibliografici.

Un sistema lineare può essere espresso come un'equazione matriciale in cui ogni elemento della matrice o del vettore appartiene a un campo, tipicamente il campo  $\mathbb{R}$  dei numeri reali.

La Decomposizione LU parte da un sistema lineare:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m \end{cases}$$

*Figura 1: Sistema di Equazioni Lineari*

Una soluzione del sistema è un insieme di valori che le soddisfa contemporaneamente tutte. Tratterò solo il caso in cui ci sono esattamente  $n$  equazioni in  $n$  incognite.

Il sistema in Figura 1 può essere espresso nella seguente forma matriciale:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

*Figura 2: Forma Matriciale*

Ora dalle nozioni matematiche del liceo, sappiamo che è possibile risolvere il sistema, sapendo che il vettore contenente le soluzioni è dato da:

$$X = A^{-1} * b$$

Dal mero punto di vista algebrico questa affermazione è sicuramente vera, però, sappiamo bene che è molto laborioso calcolare l'inversa di una matrice anche se di piccole dimensioni. In termini rigorosi ciò si traduce nel dire che questa metodologia risolutiva soffre del problema dell'*instabilità numerica*. Fortunatamente, esistono altri metodi per ricavare le soluzioni del sistema in un modo più efficiente dal punto di vista temporale e di carico computazionale. Una di queste è la Fattorizzazione LU, l'idea che sta alla base di questa procedura è trovare tre matrici, di eguale dimensione di A, tali che:

$$PA = LU$$

Dove:

- L è una matrice triangolare inferiore;
- U è una matrice triangolare superiore;
- P è una matrice di permutazione.

Un importante teorema dell'*Algebra Matriciale* ci consente di affermare che ogni matrice non singolare (cioè con determinante nulla) possiede una decomposizione di questo tipo. Ometto per ovvie ragioni i vari passaggi algebrici e arriviamo alle conclusioni: date le tre matrici LUP possiamo applicare i seguenti metodi per ricavare le soluzioni del nostro sistema lineare:

$$y_i = b_i - \sum_{j=1}^{i-1} l_{i,j} y_j$$

*Figura 3: Metodo di sostituzione in avanti (Applicato ad L)*

$$x_i = \left( y_i - \sum_{j=i+1}^n u_{i,j} x_j \right) / u_{i,i}.$$

Figura 4: Metodo di sostituzione all'indietro (Applicato su U)

Dal punto di visto dello pseudo-codice:

```

LUP-SOLVE(L,U,π,b)
1  n ← rows[L]
2  for i ← 1 to n
3      do yi ← bπ[i] - ∑j=1i-1 lij yj
4  for i ← n downto 1
5      do xi ← (yi - ∑j=i+1n uij xj) / uii
6  return x

```

Figura 5: Pseudo-codice LUSolve()

Per decomporre una matrice in forma LU usiamo un metodo noto come **eliminazione gaussiana**, dal nome del celebre matematico tedesco del XIX secolo. È curioso notare il fatto che anche se porta (erroneamente) il suo nome, l'algoritmo era noto già nell'Antica Grecia...

*“Iniziamo sottraendo dei multipli opportuni dalla prima equazione alle altre, in modo da eliminare la prima variabile nelle altre equazioni. Poi, sottraiamo dei multipli opportuni dalla seconda equazione alla terza e alle successive, in modo da eliminare la prima e la seconda variabile da tale equazione. Continuiamo questo processo finché il sistema restante non avrà una forma triangolare superiore: ecco che abbiamo trovato la matrice U. La matrice L è composta dai coefficienti per cui sono state moltiplicate le righe e che hanno permesso di eliminare le variabili che precedono x nellaesima equazione.”*

È abbastanza immediato notare che se la diagonale della matrice che viene passata alla funzione di decomposizione contiene degli zeri allora la procedura cerca di dividere per zero, il che è un problema anche se la diagonale non ha zero ma contiene dei numeri molto piccoli in valore assoluto, incorriamo nell'instabilità numerica.

Quelli che sono i denominatori a ogni passo della decomposizione LU sono detti **pivot** ed occupano le posizioni lungo la diagonale della matrice U. La matrice di permutazione P serve proprio ad evitare le divisioni per zero, o, per numeri molto piccoli. L'uso delle permutazioni per evitare queste divisioni infruttuose è detto *pivoting*.

Ritengo utile fornire una spiegazione grafica che possa rendere questo procedimento, in realtà molto semplice, il più chiaro possibile:

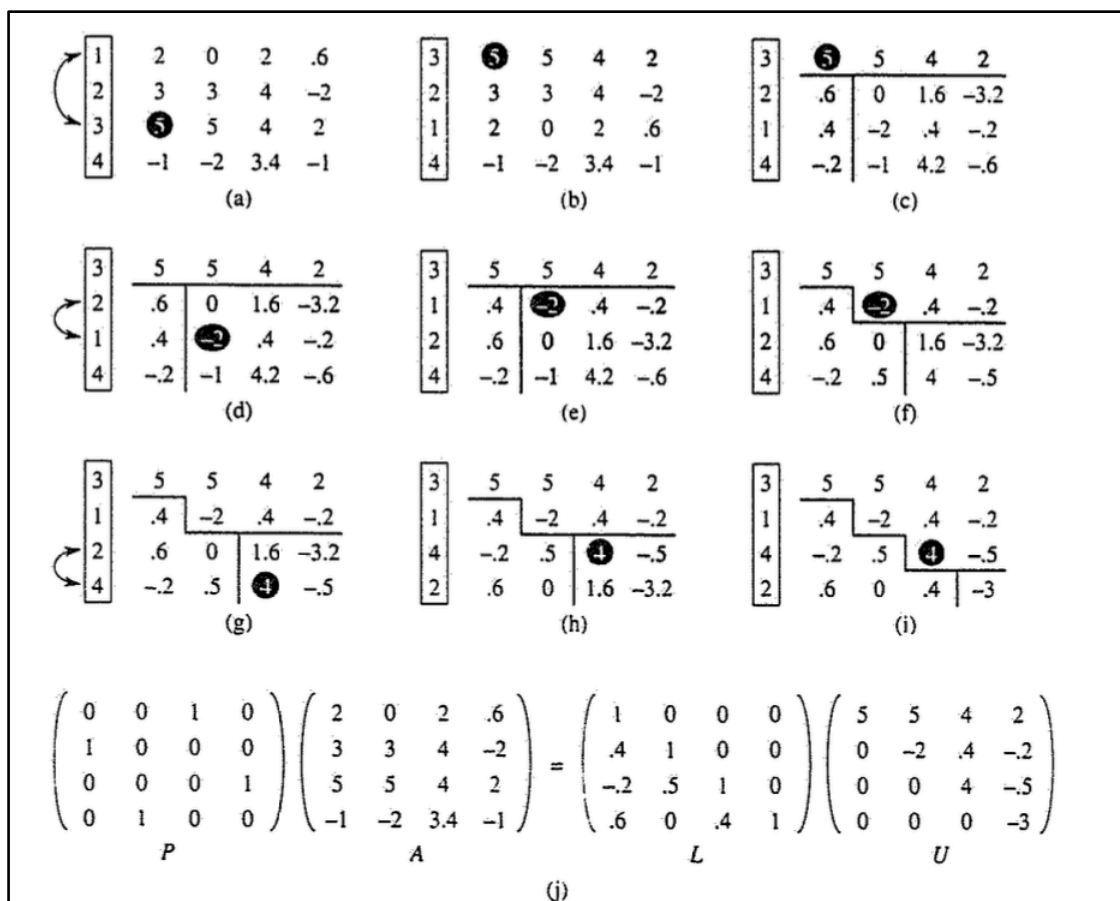


Figura 31.2 Il comportamento di LUP-DECOMPOSITION. (a) La matrice di input A con la permutazione delle righe sulla sinistra. Il primo passo dell'algoritmo determina che l'elemento 5 in nero nella terza riga è il perno per la prima colonna. (b) Le righe 1 e 3 sono scambiate e la permutazione è aggiornata. La colonna e la riga in grigio rappresentano  $v$  e  $w^T$ . (c) Il vettore  $v$  è sostituito da  $v/5$  e la parte in basso a destra della matrice è aggiornata con il complemento di Schur. Le linee dividono la matrice in tre regioni: elementi di U (sopra), elementi di L (a sinistra) e elementi del complemento di Schur (in basso a destra). (d)-(f) Il secondo passo. (g)-(i) Il terzo passo. Nessun ulteriore cambiamento occorre nel quarto ed ultimo passo. (j) La fattorizzazione LUP,  $PA = LU$ .

Dal punto di vista dello pseudo-codice:

```
LUP-DECOMPOSITION(A)
1   $n \leftarrow \text{rows}[A]$ 
2  for  $i \leftarrow 1$  to  $n$ 
3      do  $\pi[i] \leftarrow i$ 
4  for  $k \leftarrow 1$  to  $n$ 
5      do  $p \leftarrow 0$ 
6          for  $i \leftarrow k$  to  $n$ 
7              do if  $|a_{ik}| > p$ 
8                  then  $p \leftarrow |a_{ik}|$ 
9                       $k' \leftarrow i$ 
10         if  $p = 0$ 
11             then error "matrice non invertibile"
12         scambia  $\pi[k] \leftrightarrow \pi[k']$ 
13         for  $i \leftarrow 1$  to  $n$ 
14             do scambia  $a_{ki} \leftrightarrow a_{k'i}$ 
15         for  $i \leftarrow k+1$  to  $n$ 
16             do  $a_{ik} \leftarrow a_{ik}/a_{kk}$ 
17                 for  $j \leftarrow k+1$  to  $n$ 
18                     do  $a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj}$ 
```

Figura 6: Pseudo-codice *LUDecompose()*

Analizzando lo pseudo-codice è evidente che l'algoritmo presenta una complessità temporale:

- **$O(n^3)$**   $\rightarrow$  nella fase di *LUDecomposition()*
- **$O(n^2)$**   $\rightarrow$  nella fase di *LUSolve()*



## VERSIONE SEQUENZIALE <sup>[2]</sup>

Prima di poter sviluppare un approccio parallelo per la risoluzione di questo algoritmo, è stato necessario codificare una versione sequenziale che sia corretta e la più efficiente possibile.

Molte delle implementazioni di questo algoritmo sono basate su un approccio puramente di analisi teorica e ignorano gli aspetti pratici relativi all'implementazione. L'**overhead** associato all'uso di elementi bidimensionali, quali sono le matrici, è un qualcosa di cruciale. Usando le capacità di *addressing* del C si riduce significativamente il costo di accesso agli elementi delle matrici.

### ARRAY in C

Molti linguaggi di alto livello supportano array multidimensionali come standard.

Relegando tutta l'attività al compilatore, il programmatore non ha bisogno di scendere nei dettagli implementativi che gestiscono l'accesso agli elementi di un array.

Nel linguaggio C <sup>[3]</sup> gli array sono **memorizzati internamente per righe**, ad esempio:

*double matrice*[10][10]

Per accedere ad un particolare elemento, *data* [*i*][*j*], di quella matrice l'indirizzo dell'elemento è ricavato dalla seguente equazione:

$$BASE + (i * 10 + j) * ELSIZE$$

Dove:

- **BASE** → è l'indirizzo dell'elemento zero dell'array *matrice*;
- **ELSIZE** → è la dimensione del tipo di dato degli elementi dell'array.

Il codice assembly generato da quell'equazione coinvolge:

1. una **moltiplicazione**;
2. uno **shift**;
3. due **addizioni**.

È importante notare che questa equazione di accesso è relativamente semplice nel C, perché tutti gli array hanno un *Lower Bound* di zero, altri linguaggi, che lasciano scegliere all'utente il lower bound, come il Pascal o il Fortran, richiedono un costo computazionale ancora maggiore. Quando gli accessi a questi elementi cominciano ad essere veramente molti, i compilatori non sono in grado di ottimizzare da soli gli accessi.

La chiave per ridurre l'overhead associato alla gestione delle matrici è riconoscere le situazioni in cui più elementi di un array devono essere acceduti in un certo ordine; proprio come nella decomposizione LU con pivoting.

In questo caso, il primo elemento della colonna può essere acceduto usando l'equazione di accesso, ma tale costo computazionale può essere ridotto per tutti gli altri che stanno sotto. Usando opportunamente i **puntatori**, le matrici possono essere viste da un punto di vista lineare, ciò ci permette di aggirare il fatto che il C ordina le matrici internamente per riga e ci consente di operare per colonna senza avere alcun impatto negativo sulle prestazioni.

È come se implementasse noi programmatori la nostra equazione di accesso agli elementi. Questa cosa può sembrare strana al giorno d'oggi, ma i vecchi programmatori assembly non avevano alcun supporto agli multidimensionali, dovevano essere loro ad implementare la propria logica di accesso, sempre e comunque!

Il vantaggio di un linguaggio ad alto livello come il C è proprio quello di astrarci completamente dalle implementazioni fisiche e può essere facile argomentare che l'uso dei puntatori per codificarono la nostra diversa logica di accesso agli elementi di un array è contraria alla filosofia di linguaggio di alto livello.

Questo può essere vero per molti linguaggi ma non è un'associazione sempre vera in C!!!

Se gli autori del linguaggio ci hanno fornito uno strumento così potente e di basso livello come i puntatori è ovviamente un modo per poter incrementare l'efficienza del codice sviluppato.

Ed è proprio per ragioni di efficienza se il C, dopo più di cinquant'anni dalla sua nascita, è ancora un linguaggio utilizzatissimo in ogni ambito, soprattutto dove è necessaria la massima precisione e il minor consumo di risorse!!!

Dunque, è lampante che l'uso forte dei puntatori è perfettamente coerente con la filosofia del linguaggio C ed è proprio questa l'implementazione sequenziale che ho voluto adottare.

Quanto detto viene evidenziato in quest'analisi computazionale:

	<i>Theoretical Results</i>	<i>Array Implementation</i>	<i>Pointer Implementation</i>
<b>Decomposition</b>			
Multiplications	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$n^3 + 5n^2 - 6n$	$\frac{1}{3}n^3 + \frac{3}{2}n^2 + \frac{7}{6}n - 3$
Additions	0	$\frac{4}{3}n^3 + 11n^2 - \frac{37}{3}n$	$\frac{2}{3}n^3 + \frac{13}{2}n^2 - \frac{19}{6}n - 4$
Shifts	0	$\frac{2}{3}n^3 + \frac{11}{2}n^2 - \frac{37}{6}n$	$\frac{9}{2}n^2 - \frac{3}{2}n - 3$
Subtractions	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
Divides	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$
<b>Solve</b>			
Multiplications	$n^2 - n$	$2n^2 - n$	$n^2 + n$
Additions	0	$2n^2$	$\frac{1}{2}n^2 + \frac{5}{2}n$
Shifts	0	$n^2$	$3n$
Subtractions	$n^2 - n$	$n^2 - n$	$\frac{3}{2}n^2 - \frac{1}{2}n$
Divides	$n$	$n$	$n$

*Figura 7: Analisi Computazionale*

Il codice sviluppato, ricavato da un lavoro di Alan Meyer (1987) è allegato alla consegna.

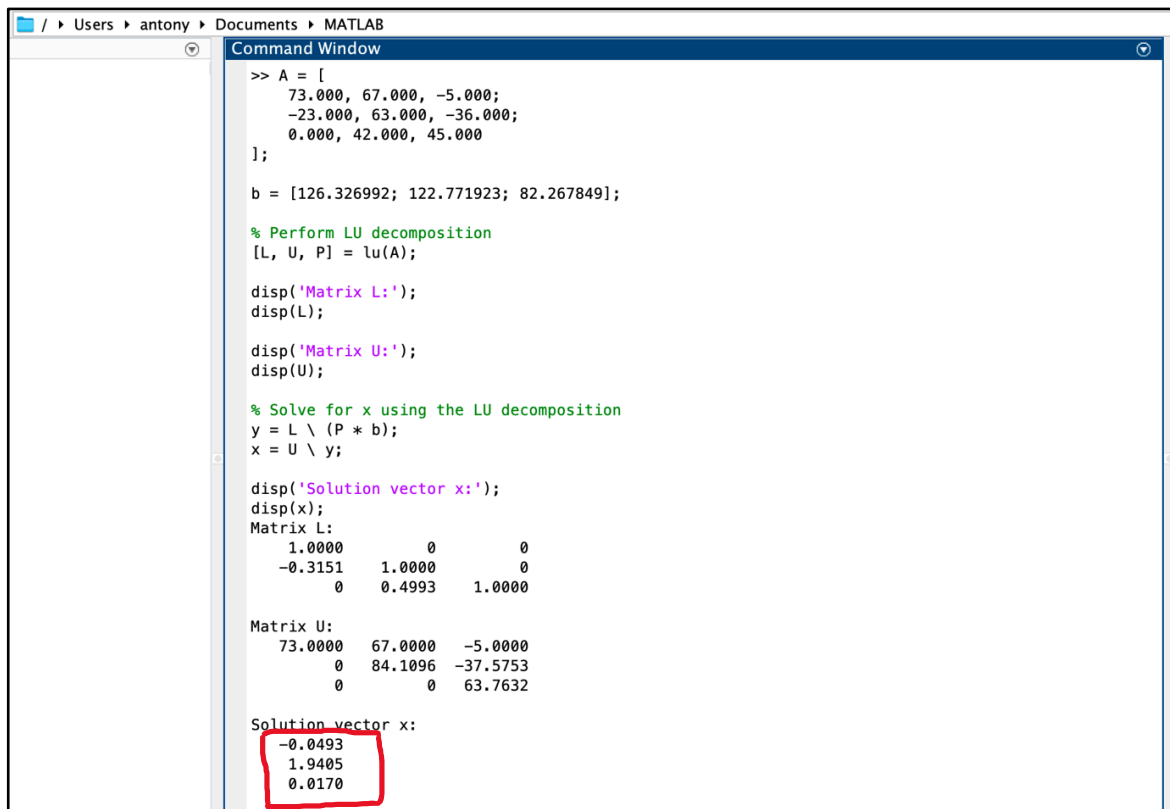
## VERSIONE PARALLELA: “OpenMP + MPI” [4]

Partendo dalla soluzione sequenziale, sviluppata precedentemente, ho realizzato la mia versione parallela dell'algoritmo. Valutando la complessità computazionale, mi è stato subito chiaro che la parte che assorbiva più risorse era quella di decomposizione, motivo per cui ho cercato di ottimizzarla nel migliore dei modi. Viste le natura del problema ho deciso di utilizzare un approccio a memoria condivisa per gestire la funzione di decomposizione. Usare un approccio a scambio di messaggi, in questa fase, sarebbe stato deleterio.

In sintesi:

- **OpenMP** → è stato usato per parallelizzare la funzione di *LUDecompose()*
- **MPI** → è stato usato per parallelizzare la fase di *LUSolve()*

Prima di passare alla fase di valutazione delle prestazioni, ritengo opportuno dimostrare che il mio codice non solo esegue l'algoritmo nel minor tempo possibile (sulla mia macchina) ma restituisce i valori corretti. Per convalidare ciò, ho fornito in input una matrice nota ed ho confrontato le soluzioni dei miei algoritmi paralleli con quelle di MATLAB.



```
>> A = [
    73.000, 67.000, -5.000;
   -23.000, 63.000, -36.000;
    0.000, 42.000, 45.000
];

b = [126.326992; 122.771923; 82.267849];

% Perform LU decomposition
[L, U, P] = lu(A);

disp('Matrix L:');
disp(L);

disp('Matrix U:');
disp(U);

% Solve for x using the LU decomposition
y = L \ (P * b);
x = U \ y;

disp('Solution vector x:');
disp(x);
Matrix L:
    1.0000         0         0
   -0.3151     1.0000         0
         0     0.4993     1.0000

Matrix U:
    73.0000    67.0000   -5.0000
         0    84.1096  -37.5753
         0         0    63.7632

Solution vector x:
   -0.0493
    1.9405
    0.0170
```

```

=====+
=====+ | Decomposizione LU: Risoluzione di Sistemi Lineari | =====+
=====+

Matrix 'A' is:

73.000      67.000      -5.000
-23.000     63.000     -36.000
 0.000     42.000     45.000

-----

Matrix 'L' is:

73.000      0.000      0.000
-23.000     84.110      0.000
 0.000     42.000     63.763

-----

Matrix 'U' is:

1.000      0.918     -0.068
0.000      1.000     -0.447
0.000      0.000      1.000

b1 = 126.326992
b2 = 122.771923
b3 = 82.267849

Risultato del sistema di equazioni:

x1 = -0.049327
x2 = 1.940494
x3 = 0.017047

=====
Il Tempo impiegato per risolvere il sistema è: 0.000599

```

Come si evince dalle immagini, i risultati sono esattamente gli stessi di quelli computati da MATLAB. Però, ci si potrebbe chiedere perché le matrici LU non sono le stesse?

Questa discrepanza è dovuta ai *limiti di rappresentazione in virgola mobile del mio compilatore C*, i quali sono molto inferiori a quelli del software MATLAB. Per dimostrare che nonostante appaiano in due forme diverse le matrici computeate siano corrette ho effettuato la moltiplicazione delle stesse e di seguito è riportato il risultato.

```

/Users/antony/Documents/MATLAB
Command Window

>> L = [
    73.000, 0.000, 0.000;
   -23.000, 84.110, 0.000;
    0.000, 42.000, 63.763
];

U = [
    1.000, 0.918, -0.068;
    0.000, 1.000, -0.447;
    0.000, 0.000, 1.000
];

% Verify LU decomposition
A_reconstructed = L * U;

disp('Reconstructed Matrix A:');
disp(A_reconstructed);
Reconstructed Matrix A:
    73.0000    67.0140   -4.9640
   -23.0000    62.9960  -36.0332
         0    42.0000    44.9890

fx >>

```

## VALUTAZIONE PRESTAZIONI: “OpenMP + MPI”

Per testare le prestazioni del mio algoritmo parallelo ho eseguito circa 500 test partendo da una matrice con dimensione 250 fino a 10.000. Siccome l'algoritmo ha una complessità computazionale cubica: l'andamento generale, da rispettare in ogni caso, è dato dalla figura sottostante.

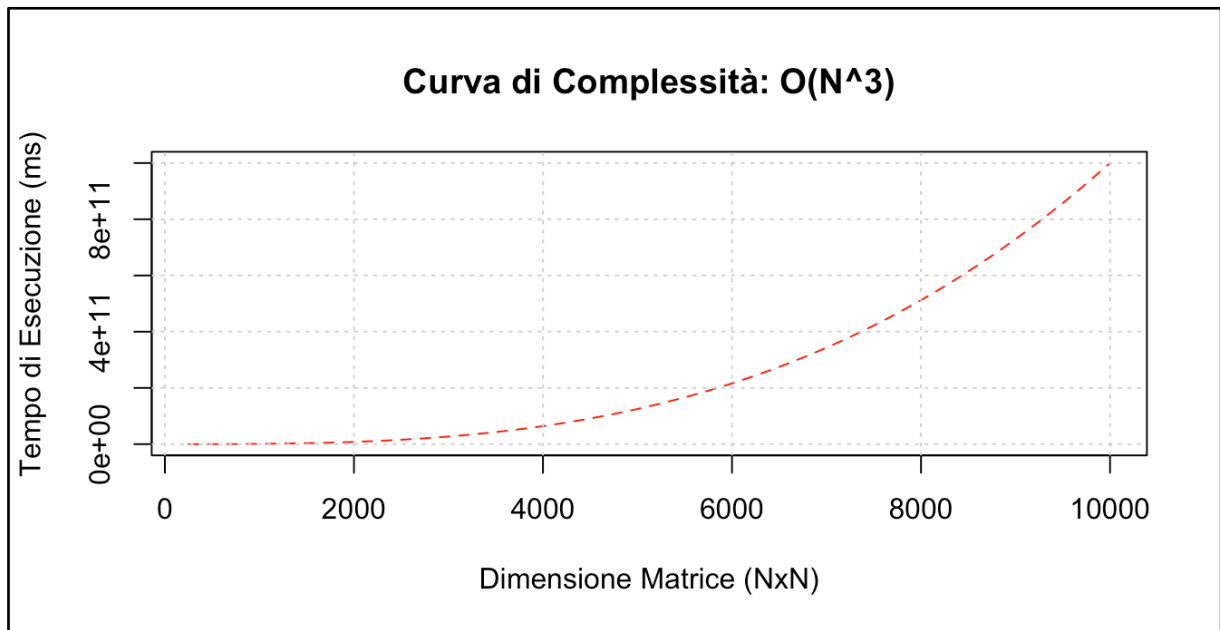
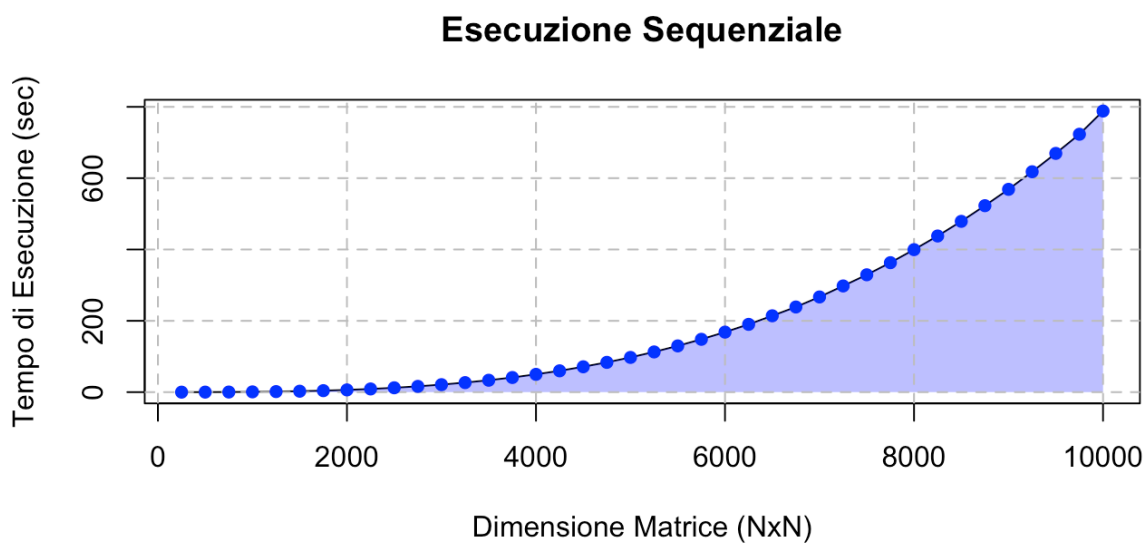
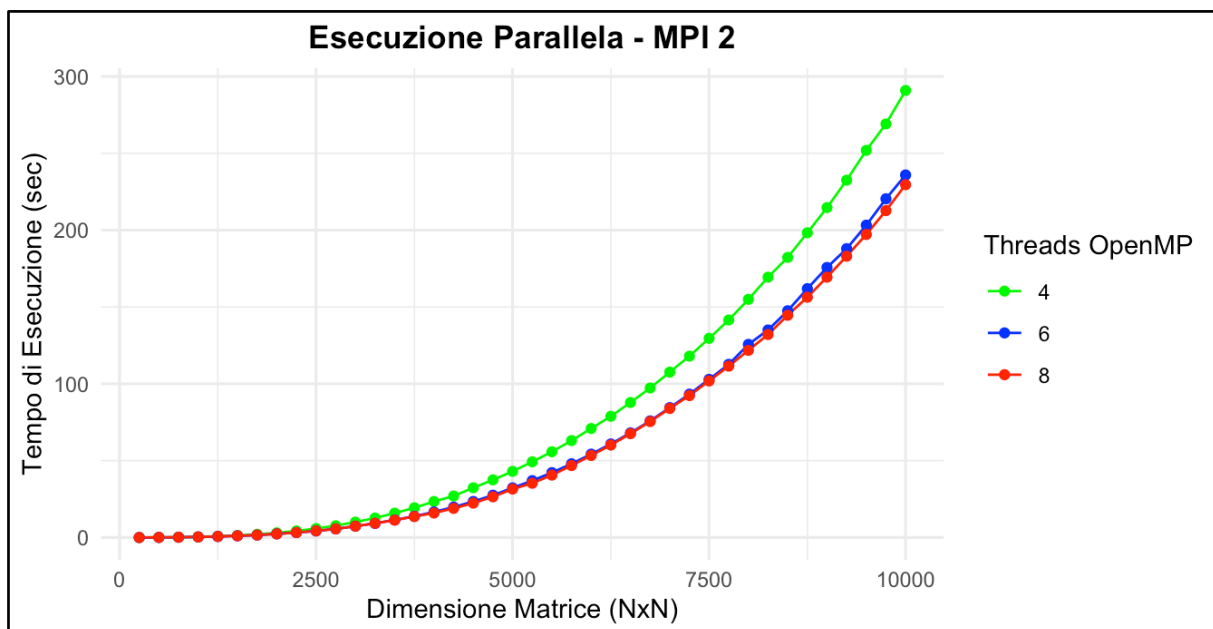
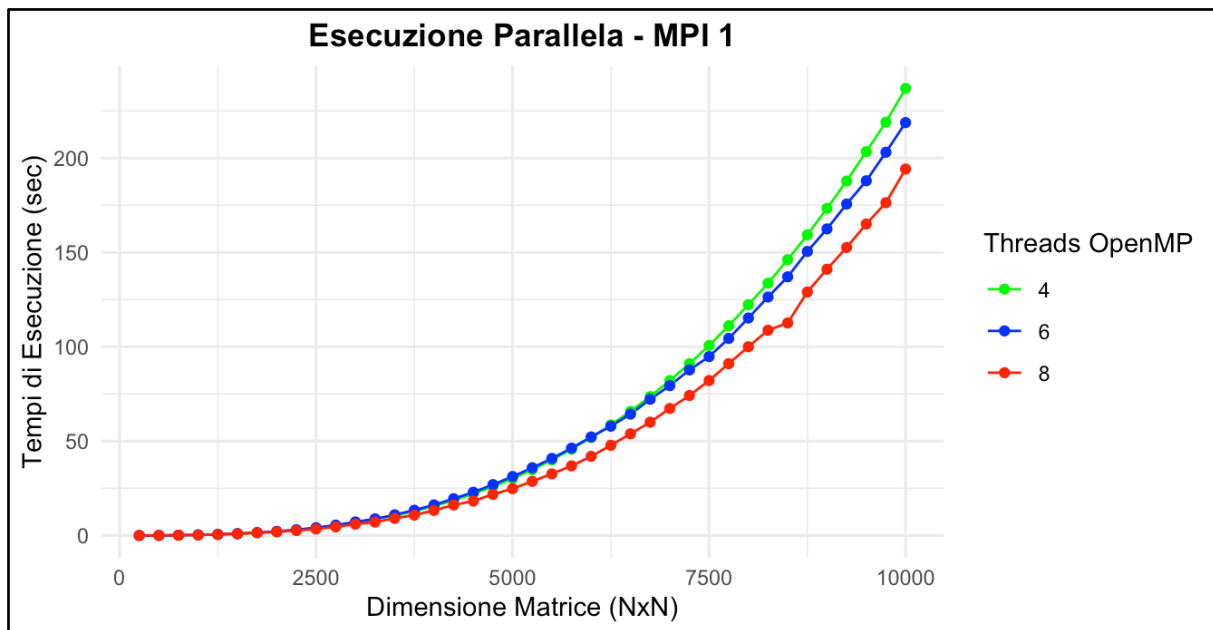
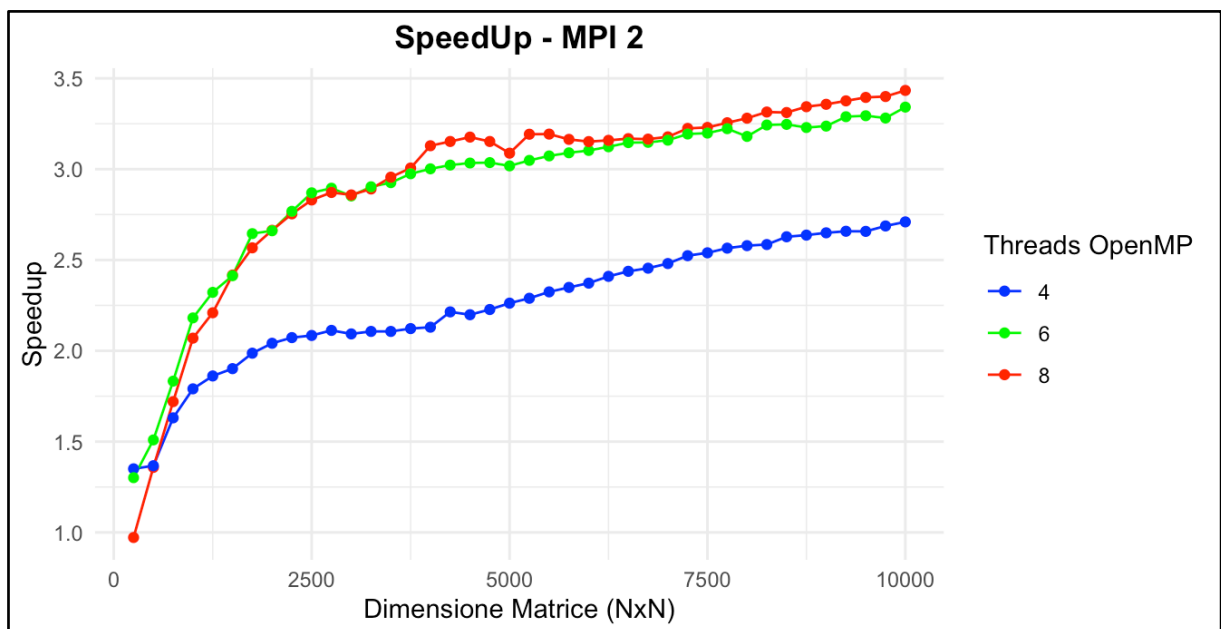
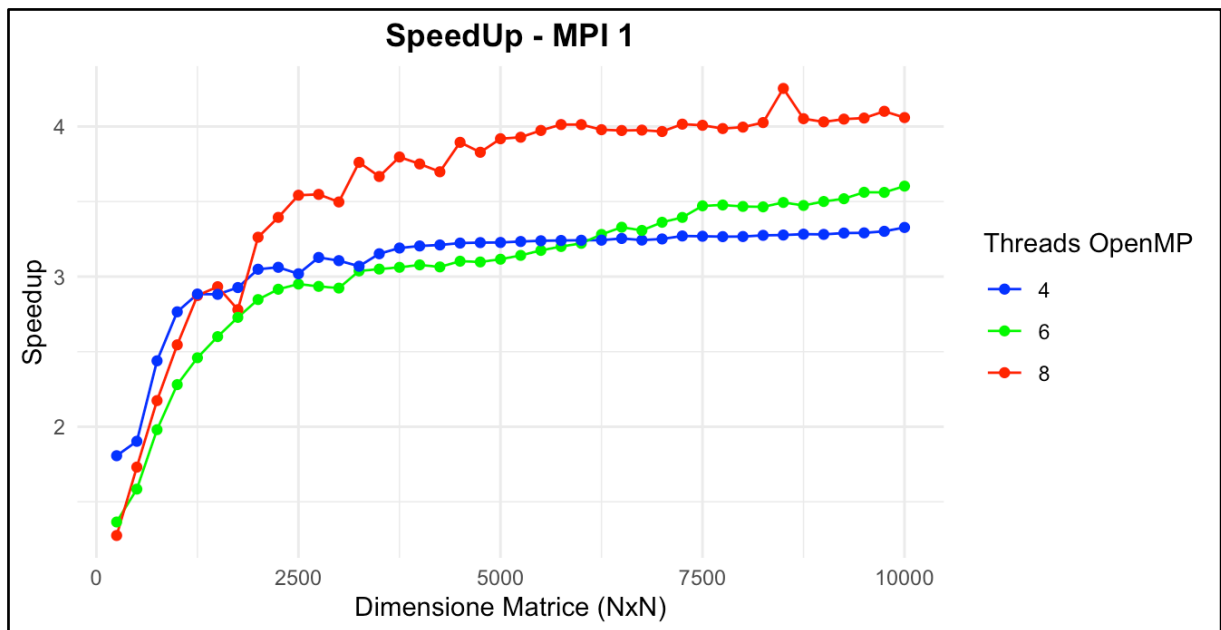


Figura 8: Complessità computazionale teorica

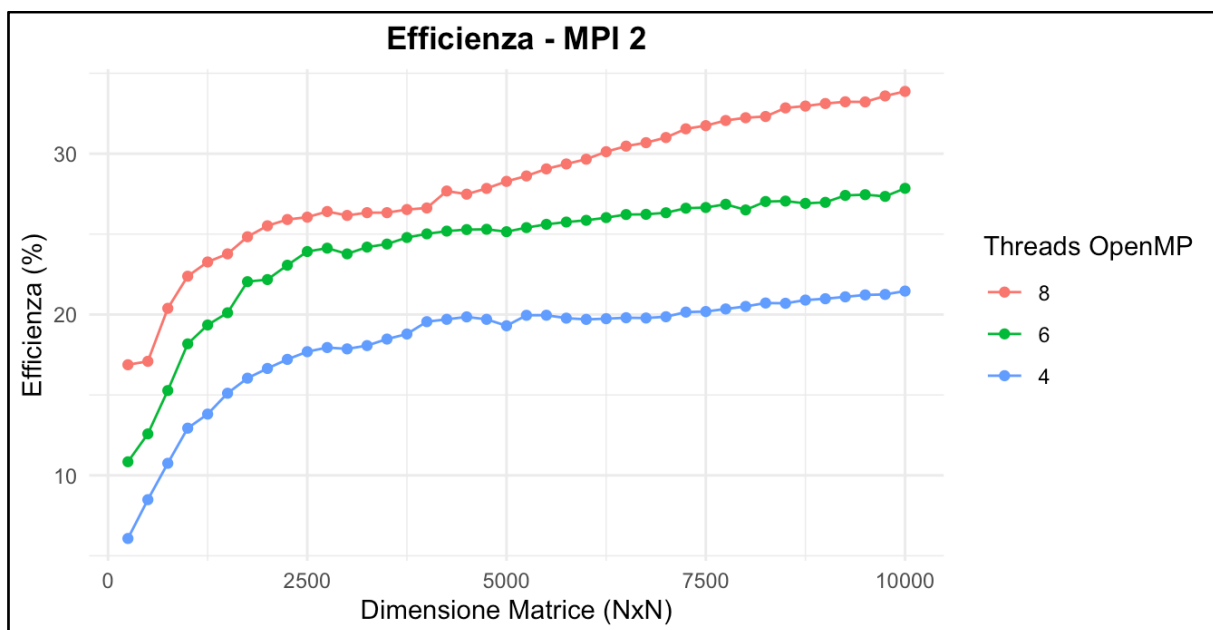
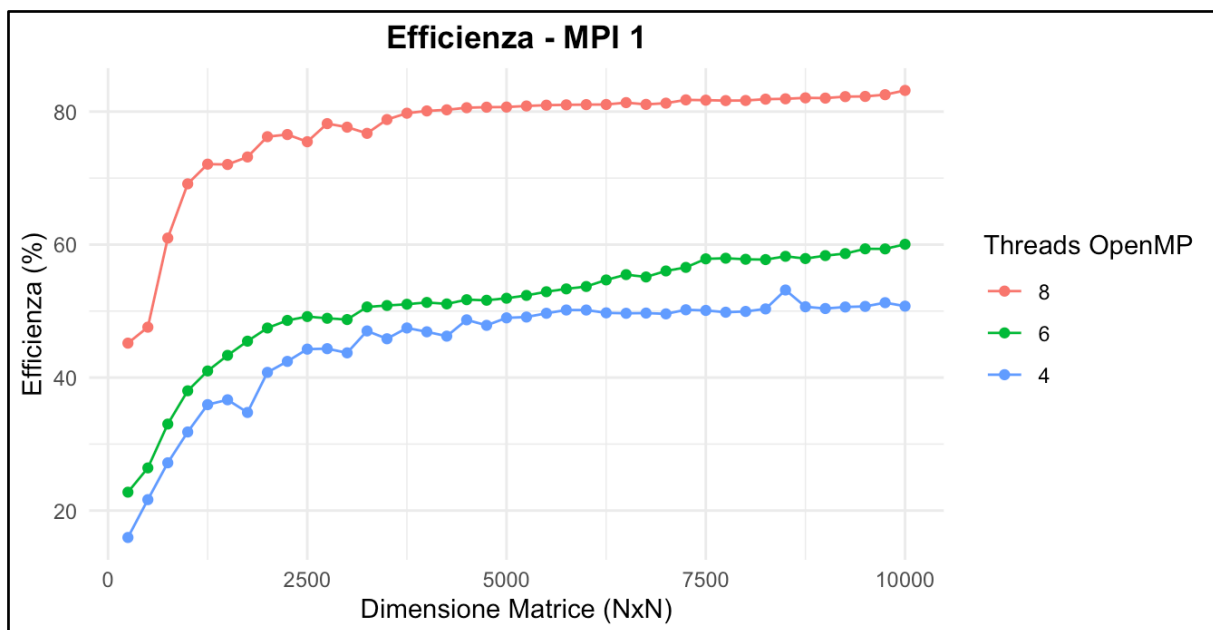
Di seguito i risultati dei miei test:











## VERSIONE PARALLELA: “OpenMP + CUDA” [5]

Partendo dalla soluzione parallela, sviluppata precedentemente, ho realizzato la versione CUDA dell’algoritmo. La parte che assorbiva più risorse era quella di decomposizione, motivo per cui ho cercato di ottimizzarla utilizzando un approccio di elaborazione su kernel GPU.

Usare qui un approccio di computazione su CPU parallelo, sarebbe stato deleterio.

Le GPU sono dotate di un gran numero di core di elaborazione, spesso centinaia o addirittura migliaia. Questi core possono eseguire simultaneamente le stesse operazioni su differenti parti dei dati, permettendo una parallelizzazione massiva delle operazioni matriciali.

In sintesi:

- **CUDA** → è stato usato per parallelizzare la funzione di *LUDecompose()*
- **OpenMP** → è stato usato per parallelizzare la fase di *LUSolve()*

Inizialmente i test su questa versione parallela avevano un risultato peggiore di quello sequenziale, la causa era da ricercarsi nel fatto che l’esecuzione avveniva su *Google Colab*.

In sintesi, migliaia di altri utenti usavano la stessa mia GPU nello stesso istante di tempo e questo contribuiva a creare un deterioramento delle prestazioni incredibile.

Per risolvere il problema ho ripetuto i test con un account Premium e durante le ore serali.

Così facendo sono riuscito a tirar fuori le vere potenzialità del *Calcolo Numerico* su GPU. Proprio per la intrinseca differenza tra una GPU e una CPU non sarebbe stato possibile avere risultati mediocri, avendo un algoritmo sicuramente funzionante dal punto di vista implementativo. Anche se parlare di *SpeedUP* in campo CUDA trova il tempo che trova, si sarebbe dovuto misurare il **Bandwidth**, cosa non possibile in Colab, per quel che ho constatato. Infatti, il collo di bottiglia dell’intero algoritmo è causato dalle latenze nei trasferimenti di informazioni tra Host (CPU) e Device (Nvidia GPU), e, dall’impossibilità su Colab di usare più di due core della CPU del dispositivo Host.

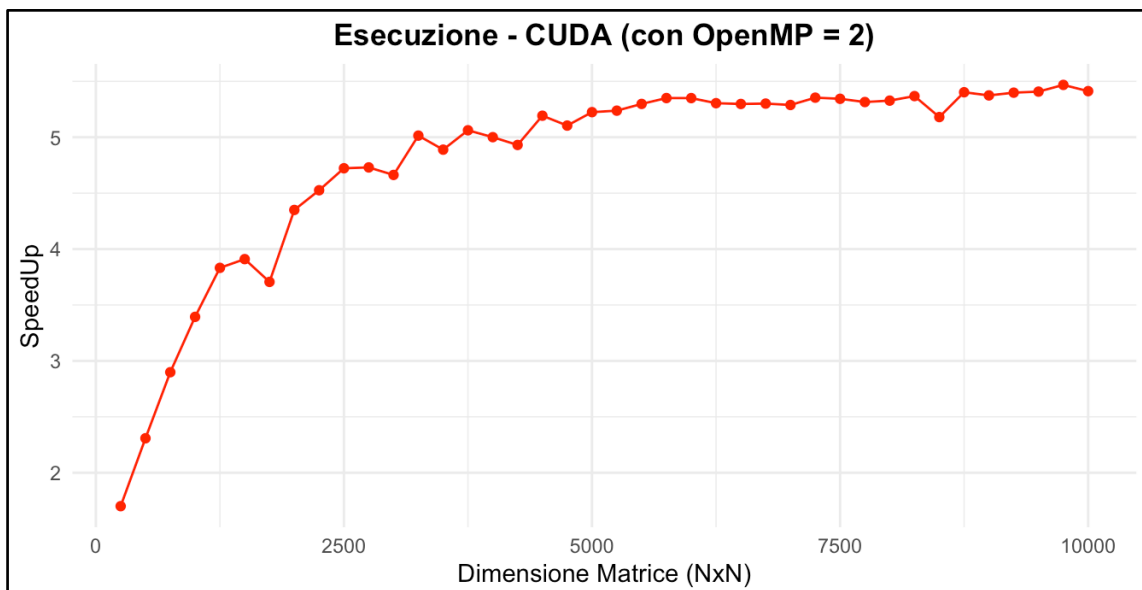
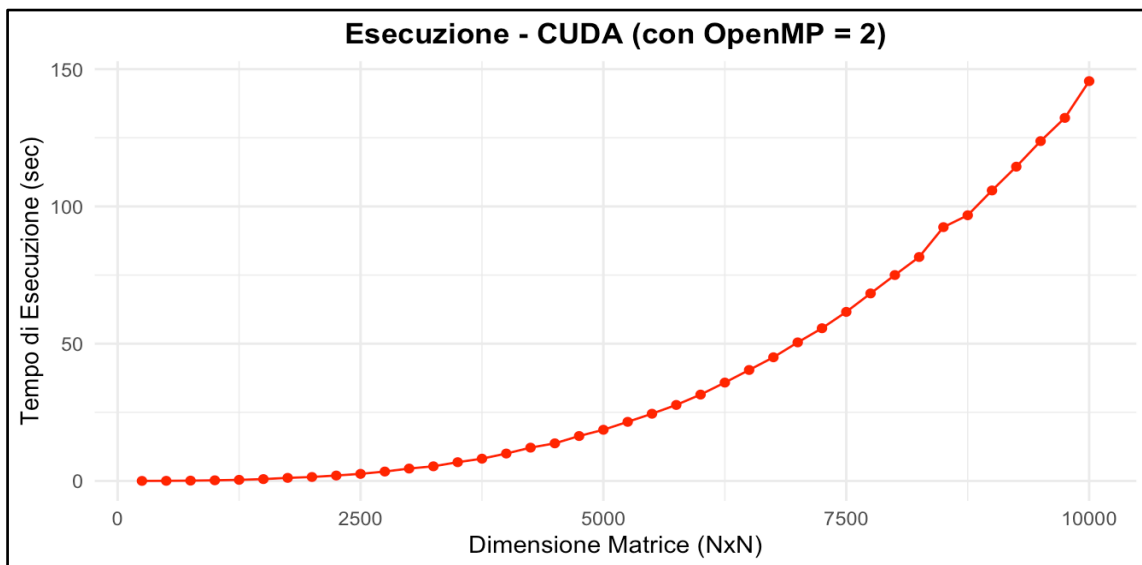
Anche per questo algoritmo si è verificata la correttezza dei risultati restituiti confrontandoli con quelli di MATLAB (*Vedi quanto discusso in “OpenMP + MPI”*).

## VALUTAZIONE PRESTAZIONI: “OpenMP + CUDA”

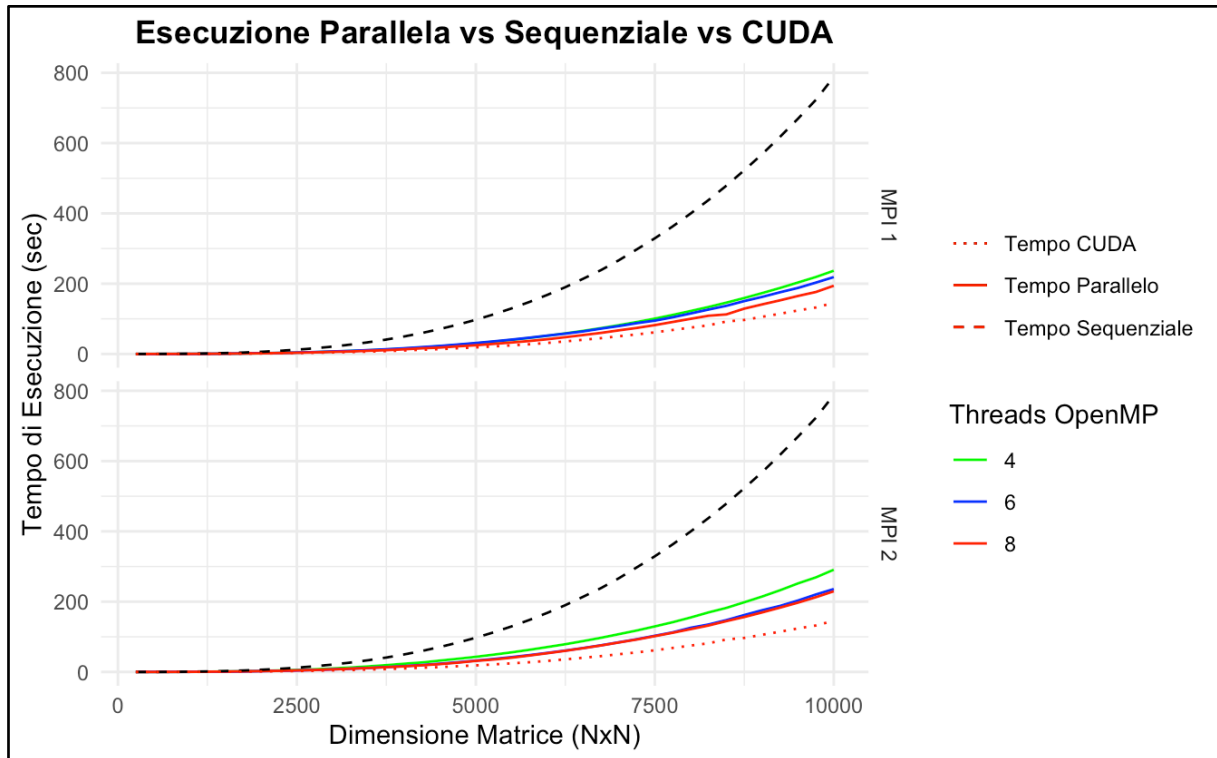
```
!nvidia-smi
```

Thu Jan 11 09:46:47 2024

NVIDIA-SMI 535.104.05			Driver Version: 535.104.05			CUDA Version: 12.2		
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	MIG M.
0	Tesla T4		Off	00000000:00:04.0	Off		0	
N/A	58C	P8	11W / 70W	0MiB / 15360MiB		0%	Default	N/A



## CONCLUSIONI



I grafici, elaborati con R, sono alquanto auto-esplicativi, motivo per cui non mi dilungherò troppo nel fornirne un'analisi testuale. L'algoritmo CUDA, per le ragioni sopra discusse, è quello più performante. La versione parallela OpenMP+MPI dà il meglio di sé solo quando MPI è assente ( $MPI == 1$ ): l'algoritmo di LU Decomposizione mal si adatta ad un approccio di parallelismo a memoria non condivisa; infatti, tutte le operazioni possono essere agevolmente eseguite in contemporanea. L'approccio di attese e scambio di messaggi di MPI implica il drastico peggioramento dell'efficienza globale e l'aumento dei tempi di esecuzione. Sarebbe stato possibile ottenere uno *speedup* ancora maggiore eliminando totalmente la parte MPI in *LuSolve()*, sostituendola con una parallelizzazione OpenMP.

Globalmente posso affermare di avere ottenuto dei buoni risultati che denotano un alto livello di scalabilità dei miei due algoritmi.

## INIDICE DELLE FIGURE

FIGURA 1: SISTEMA DI EQUAZIONI LINEARI.....	4
FIGURA 2: FORMA MATRICIALE .....	4
FIGURA 3: METODO DI SOSTITUZIONE IN AVANTI (APPLICATO AD L) .....	5
FIGURA 4: METODO DI SOSTITUZIONE ALL'INDIETRO (APPLICATO SU U).....	6
FIGURA 5: PSEUDO-CODICE LUSOLVE().....	6
FIGURA 6: PSEUDO-CODICE LUDECOMPOSE() .....	8
FIGURA 7: ANALISI COMPUTAZIONALE .....	11
FIGURA 8: COMPLESSITÀ COMPUTAZIONALE TEORICA.....	14

# Bibliografia

- [1] T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST, C. STEIN, “Introduzione agli algoritmi e strutture dati”, SECONDA EDIZIONE, MCGRAW-HILL, 2005.
- [2] Alan Meyer, “An efficient implementation of LU decomposition in C” (1987)
- [3] Brian W. Kernighan, Dennis M. Ritchie - The C Programming Language (1978), p.104
- [4] S. Lee, “A faster LU Decomposition for parallel C programs” (1996)
- [5] B. Sen, “Investigation of the performance of LU decomposition method using CUDA” (2011)