Università degli Studi di Milano - Bicocca
Scuola di Scienza
Dipartimento di Informatica, Sistemi e Comunicazione
Corso di Scienze Informatiche

Metodo del Calcolo Scientifico - Assignment 1 - Decomposizione con metodo di Cholesky

Relazione di:
Pellegrini Damiano 886261
Sanvito Marco 886493

Anno accademico 2024-2025

Sommario

Il presente progetto si propone di confrontare soluzioni proprietarie e open-source per la risoluzione di sistemi lineari sparsi mediante la fattorizzazione di Cholesky. Verrà analizzato come, attraverso un'implementazione personalizzata, siano stati ottenuti risultati più accurati e veloci rispetto alle soluzioni proprietarie prese in esame, approfondendo le sfide incontrate durante il processo di sviluppo.

Indice

Introduzione

La fattorizzazione di Cholesky rappresenta un metodo efficiente per risolvere sistemi lineari quando la matrice dei coefficienti è simmetrica e definita positiva. Data una matrice A, la fattorizzazione di Cholesky produce una matrice triangolare inferiore L tale che:

$$A = L \cdot L^T \tag{1}$$

Dove:

- A è una matrice simmetrica definita positiva (il che implica che tutti i suoi autovalori sono strettamente positivi)
- L è una matrice triangolare inferiore
- L^T è la trasposta di L

Questa decomposizione risulta particolarmente utile nella risoluzione di sistemi lineari, nell'ottimizzazione numerica e nelle applicazioni statistiche.

Esiste anche una variante della fattorizzazione di Cholesky che introduce una matrice diagonale D, portando alla seguente espressione:

$$A = L \cdot D \cdot L^T \tag{2}$$

Dove:

- A è una matrice simmetrica definita positiva
- L è una matrice triangolare inferiore con diagonale unitaria (tutti gli elementi diagonali sono 1)
- D è una matrice diagonale contenente i pivot della fattorizzazione
- L^T è la trasposta di L

Questa variante è particolarmente utile quando si desidera evitare di estrarre la radice quadrata degli elementi diagonali di L, semplificando così i calcoli e migliorando la stabilità numerica.

Nelle applicazioni pratiche, molti problemi ingegneristici e scientifici generano matrici di grandi dimensioni in cui la maggior parte degli elementi sono zero (matrici sparse). Questo porta a un problema durante la fattorizzazione di Cholesky, cioè la gestione del fill-in. Il fill-in è un fenomeno che si verifica durante la fattorizzazione di Cholesky, in cui gli zeri nella matrice originale diventano non zero nella matrice triangolare inferiore L. Questo può portare a un significativo aumento del numero di elementi non-zero e, di conseguenza, a maggiori requisiti di memoria e tempo di calcolo per la fattorizzazione. Il fill-in è fortemente influenzato dall'ordinamento delle righe e delle colonne della matrice originale e quindi un ordinamento appropriato può ridurre drasticamente il numero di elementi non-zero che appaiono durante la fattorizzazione. Sono stati quindi sviluppati diversi algoritmi e tecniche per ridurre il fill-in. Tra questi, uno dei più utilizzati è l'AMD (Approximate Minimum Degree), che opera riordinando le righe e colonne in base al grado dei nodi nel grafo associato alla matrice. Altri approcci includono l'ordinamento Nested Dissection e le tecniche di Minimum Fill.

In questa relazione, analizzeremo in dettaglio l'implementazione della fattorizzazione di Cholesky in MATLAB, esplorando le sue caratteristiche, i punti di forza e le eventuali limitazioni. Successivamente, applicheremo questi concetti per sviluppare un'implementazione open source in C++ (e se possibile un'implementazione comparabile a quella di MATLAB), confrontando approfonditamente le due soluzioni in termini di efficienza computazionale, gestione della

memoria e scalabilità su diverse tipologie di matrici sparse. Questo confronto ci permetterà di verificare se ha senso avventurarsi in librerie open source per la fattorizzazione di Cholesky, oppure se è più opportuno pagare per una libreria commerciale come MATLAB.

1. Matlab

1.1. Introduzione a MATLAB

MATLAB (acronimo di «MATrix LABoratory») è un ambiente di calcolo numerico avanzato e un linguaggio di programmazione di alto livello sviluppato da MathWorks. Concepito originariamente negli anni "70 dal matematico Cleve Moler come interfaccia user-friendly per le librerie numeriche LINPACK ed EISPACK, MATLAB si è evoluto in un ecosistema completo per il calcolo scientifico e l"analisi numerica, diventando uno strumento fondamentale in svariati campi scientifici e ingegneristici.

Le caratteristiche distintive che hanno contribuito al suo successo includono:

- Gestione nativa ed efficiente delle operazioni matriciali
- Vasta collezione di funzioni matematiche ottimizzate per diverse applicazioni computazionali
- Strumenti avanzati di visualizzazione scientifica e generazione di grafici interattivi
- Ambiente di sviluppo integrato (IDE) con funzionalità complete di debug e profiling
- Architettura estensibile attraverso toolbox specializzati per domini specifici (elaborazione segnali, ottimizzazione, machine learning, statistica, ecc.)
- Interfacce per l'integrazione con altri linguaggi di programmazione come C, C++ e Python

La sintassi di MATLAB è intuitiva e orientata alla risoluzione di problemi matematici, rendendo relativamente semplice l'implementazione di algoritmi complessi con poche righe di codice.

1.2. Fattorizzazione di Cholesky in MATLAB

MATLAB implementa la fattorizzazione di Cholesky attraverso la funzione built-in chol, specificamente progettata per determinare la decomposizione di Cholesky di una matrice simmetrica definita positiva. La funzione chol offre diverse varianti sintattiche per adattarsi a esigenze computazionali specifiche, di cui la principale è:

$$R = \text{chol}(A) \tag{3}$$

Dove:

- A è una matrice simmetrica definita positiva (ovvero una matrice per cui tutti gli autovalori sono positivi)
- R è una matrice triangolare superiore tale che $A = R^T R$

Questa funzione realizza la fattorizzazione di Cholesky di una matrice simmetrica definita positiva A. Nel caso in cui A non sia simmetrica, MATLAB la tratta come se fosse simmetrica utilizzando solo la parte triangolare superiore. Se A non è definita positiva, MATLAB restituisce un errore.

Poiché il nostro caso si concentra su matrici sparse di dimensioni variabili, utilizzeremo la seguente sintassi:

$$[R, \text{flag}, p] = \text{chol}(A, \text{'vector'}) \tag{4}$$

Dove:

- A è una matrice simmetrica definita positiva
- R è una matrice triangolare superiore risultante dalla fattorizzazione
- flag è un indicatore che assume valore 0 se la matrice è definita positiva, 1 altrimenti
- p è un vettore di permutazione che ottimizza l'ordinamento delle righe e colonne di A

Per le matrici sparse di grandi dimensioni, un aspetto cruciale è la gestione del *fill-in* — fenomeno per cui elementi inizialmente nulli diventano non-zero durante la fattorizzazione, aumentando significativamente la complessità computazionale e l'utilizzo di memoria.

MATLAB affronta questo problema utilizzando l'algoritmo AMD (Approximate Minimum Degree), una strategia di riordinamento che analizza la struttura di sparsità della matrice e approssima una permutazione ottimale delle righe e colonne. Questa permutazione minimizza il fill-in atteso durante la fattorizzazione, riducendo notevolmente sia i requisiti di memoria che il tempo di calcolo.

La relazione matematica che esprime questa permutazione è:

$$R^T R = A(p, p) \tag{5}$$

dove p rappresenta il vettore di permutazione e $A(p, p)$ indica la matrice A con righe e colonne riordinate secondo p. Questo approccio produce una fattorizzazione matematicamente equivalente ma computazionalmente molto più efficiente, con un fattore sparso R che preserva maggiormente la struttura di sparsità originale.

Questa implementazione consente di ridurre la complessità algoritmica da $O(n^3)$ nel caso denso a $O(nc \cdot \text{nnz}(A))$ nel caso sparso ben ordinato, dove $\text{nnz}(A)$ rappresenta il numero di elementi non-zero. Questa complessità ottimizzata è ottenibile in casi favorevoli e dipende fortemente dall'efficacia del riordinamento e dalla struttura specifica della matrice.

1.2.1. Funzione chol in MATLAB

Per rendere più completo il confronto ed avere una base di partenza, abbiamo deciso di analizzare la struttura interna dell'implementazione di chol in MATLAB. Dall'analisi è emerso che questa funzione utilizza internamente il pacchetto CHOLMOD (CHOLesky MODification), un componente della libreria SuiteSparse T. Davis [1]. SuiteSparse rappresenta una raccolta completa e altamente ottimizzata di algoritmi per l'algebra lineare sparsa, sviluppata principalmente sotto la guida di Timothy A. Davis e disponibile come software open source su GitHub T. Davis [1].

Per verificare empiricamente l'utilizzo di SuiteSparse in MATLAB, abbiamo applicato, con piccole modifiche, uno script diagnostico che identifica le librerie matematiche sottostanti e le relative versioni tramite funzioni di debug non documentate. Lo script di partenza è disponibile presso undocumentedmatlab.com.

I risultati dello script confermano che MATLAB si affida effettivamente a molteplici componenti della libreria SuiteSparse per le operazioni su matrici sparse.

L'output generato dallo script ha evidenziato le seguenti librerie:

- Found: AMD version 2.2.0, May 31, 2007: approximate minimum degree ordering
- Found: colamd version 2.5, May 5, 2006: OK.

- Found: CHOLMOD version 1.7.0, Sept 20, 2008: : status: OK
- UMFPACK V5.4.0 (May 20, 2009), Control:
- SuiteSparseQR, version 1.1.0 (Sept 20, 2008)

1.2.2. Analisi di CHOLMOD

Approfondendo l'architettura del pacchetto CHOLMOD, abbiamo scoperto che questo si basa essenzialmente sulle librerie BLAS (Basic Linear Algebra Subprograms) e LAPACK (Linear Algebra PACKage) per eseguire operazioni di algebra lineare ad alte prestazioni. Ulteriori dettagli sul funzionamento di queste librerie sono consultabili presso netlib.org/blas [2] e netlib.org/lapack [3].

Queste librerie esistono in diverse implementazioni, ciascuna ottimizzata per architetture hardware specifiche. Le implementazioni più diffuse includono OpenBLAS (ottimizzata per molteplici architetture), Apple Accelerate (Implementazione per sistemi operativi MacOS) e Intel MKL (oneAPI Math Kernel Library), quest'ultima particolarmente performante su processori Intel.

Nel caso specifico di MATLAB su Windows e Linux, l'implementazione utilizzata è Intel MKL, che garantisce prestazioni ottimali su architetture x86 e x86-64.

È possibile verificare questa configurazione attraverso i seguenti comandi MATLAB:

version("-blas") version("-lapack")

I risultati ottenuti sono i seguenti (la versione potrebbe variare a seconda della release di MATLAB):

ans = "Intel(R) oneAPI Math Kernel Library Version 2024.1-Product Build 20240215
for Intel(R) 64 architecture applications (CNR branch AVX2)" ans = "Intel(R) oneAPI Math Kernel Library Version 2024.1-Product Build 20240215
for Intel(R) 64 architecture applications (CNR branch AVX2) supporting Linear Algebra PACKage (LAPACK 3.11.0)"

Dato che Intel MKL e Apple Accelerate sono librerie commerciali, abbiamo deciso di fare non solo un confronto tra MATLAB e Open Source, ma anche di analizzare le differenze tra le varie implementazioni di BLAS e LAPACK.

1.3. Implementazione in MATLAB

1.3.1. Parametri analizzati

Tempo di esecuzione:

- **loadTime:** tempo necessario per caricare la matrice dal file in formato MATLAB (MAT) (ms)
- **decompTime:** tempo per eseguire la fattorizzazione di Cholesky (ms)
- **solveTime:** tempo per risolvere un sistema lineare usando i fattori (ms)

Per il calcolo del tempo, abbiamo utilizzato il profiler di MATLAB attraverso profile, che misurano il tempo di esecuzione assieme ad altre informazioni di un blocco di codice.

Utilizzo di memoria:

- **loadMem:** memoria utilizzata per caricare la matrice (Bytes)
- **decompMem:** memoria utilizzata per la fattorizzazione (Bytes)

- **solveMem:** memoria utilizzata per trovare la soluzione (Bytes)

Per il calcolo della memoria, abbiamo utilizzato una funzionalità del profiler di MATLAB non documentata, che permette di calcolare la memoria utilizzata in una porzione di codice. Ovviamente, essendo non documentata, non è garantita la sua stabilità e correttezza, ma l'abbiamo ritenuto il metodo migliore per il calcolo la memoria utilizzata.

Per utilizzare questa funzionalità, abbiamo usato il comando profile -memory on che avvisa il profiler di tenere traccia della memoria utilizzata.

Accuratezza:

- **Errore Relativo:** errore relativo della soluzione calcolata rispetto alla soluzione attesa

1.3.2. Metodologia

Per ogni matrice, abbiamo eseguito i seguenti passaggi:

- Caricamento della matrice in memoria da file in formato MATLAB
- Esecuzione della fattorizzazione di Cholesky sulla matrice
- Risoluzione del sistema lineare Ax = b
- Calcolo dell'errore relativo tra la soluzione calcolata e quella attesa

Per risolvere il sistema lineare $Ac \cdot x = b$ dove il termine b è noto ed è scelto in modo che la soluzione esatta sia il vettore $x_e = [1, 1, 1, 1, 1, 1, ...]$, cioè $b = Ac \cdot x_e$.

I risultati vengono poi esportati in un file CSV per successiva analisi e confronto con altre implementazioni.

1.4. Documentazione di MATLAB

MATLAB, in quanto software commerciale, presenta una documentazione eccellente: ben strutturata, dettagliata e arricchita da numerosi esempi applicativi. L'ecosistema integrato di funzioni predefinite e toolbox specializzati consente agli utenti di implementare rapidamente soluzioni a problemi complessi con un minimo di codice, riducendo significativamente i tempi di sviluppo rispetto a soluzioni che richiederebbero l'integrazione manuale di diverse librerie.

Questo rappresenta un vantaggio sostanziale rispetto a molte alternative open source, dove la documentazione può risultare frammentaria, incompleta o non aggiornata. In ambito scientifico e ingegneristico, la rapidità di prototipazione e sviluppo offerta da MATLAB giustifica spesso l'investimento economico, soprattutto considerando i costi indiretti legati al tempo di sviluppo.

Tuttavia, è importante evidenziare alcune limitazioni. La documentazione ufficiale, pur essendo esaustiva nell'illustrare l'utilizzo delle funzioni, raramente rivela i dettagli implementativi sottostanti. Nel caso della fattorizzazione di Cholesky, ad esempio, la documentazione specifica parametri e comportamenti attesi, ma non approfondisce gli algoritmi utilizzati o le ottimizzazioni applicate. Questa opacità può risultare problematica durante il debugging di casi particolari o quando si necessita di comprendere le ragioni di determinati comportamenti computazionali.

1.5. Commenti

Un aspetto particolarmente interessante emerso dalla nostra analisi riguarda l'architettura interna di MATLAB: alcune funzionalità core, inclusa la fattorizzazione di Cholesky, si basano su librerie open source come SuiteSparse, ottimizzate tramite implementazioni non open-source,

ma con licenza libera previa citazione, utilizzabili di BLAS/LAPACK come Intel MKL. Questo solleva interrogativi legittimi sul valore aggiunto del software commerciale rispetto all'utilizzo diretto delle librerie open source sottostanti.
Il valore di MATLAB risiede quindi non tanto nell'esclusività degli algoritmi implementati, quanto nell'integrazione di questi in un ambiente coerente, ben documentato e ottimizzato per la produttività scientifica. La questione se questo valore aggiunto giustifichi il costo di licenza dipende fortemente dal contesto applicativo, dalle esigenze specifiche dell'utente e dai vincoli di tempo e risorse del progetto. Nell'ambito della ricerca, non a scopo di lucro, sviluppare uno strumento che faccia utilizzo di librerie commerciali/proprietarie e open-source potrebbe rivelarsi ragionevole.

2. C++

2.1. Introduzione a C++

C++ rappresenta una scelta ottimale per l'implementazione di algoritmi di algebra lineare grazie alle sue caratteristiche di efficienza, controllo di basso livello e supporto per la programmazione orientata agli oggetti.
Nel contesto della fattorizzazione di Cholesky, C++ ci permette di:

- Ottimizzare le operazioni su matrici sparse di grandi dimensioni
- Integrare librerie specializzate per l'algebra lineare
- Controllare precisamente l'allocazione della memoria
- Sfruttare costrutti template per implementazioni generiche

2.2. Tool Chain C++

Per lo sviluppo del nostro progetto abbiamo utilizzato diversi strumenti in base all'ambiente operativo:

2.2.1. Ambiente Windows

In ambiente Windows, la nostra implementazione si è basata su:

- **Microsoft Visual C++ (MSVC)**: Il compilatore ufficiale di Microsoft che offre ottimizzazioni specifiche per architetture Intel/AMD.
- **Intel Fortran Compiler (IFX)**: Compilatore Intel per Fortran che abbiamo utilizzato per compilare alcune componenti di SuiteSparse.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.2.2. Ambiente Linux

Per garantire la portabilità del codice e per effettuare test comparativi, abbiamo anche utilizzato:

- **GNU Compiler Collection (GCC)**: Compilatore C++ standard in ambienti Linux, utilizzato nella versione 11.3 con pieno supporto per C++17.

- **GNU Fortran (GFortran)**: Necessario per compilare alcune componenti delle librerie BLAS e LAPACK utilizzate dal progetto.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.2.3. Ambiente MacOS

Per testare un ambiente più professionale ed utilizzare una libreria proprietaria differente, ci siamo forniti di:

- **Apple clang (clang)**: Compilatore C++ standard in ambienti Apple MacOS, utilizzato nella versione 17.0.0 con pieno supporto per C++17.
- **GNU Fortran (GFortran)**: Necessario per compilare alcune componenti delle librerie BLAS e LAPACK utilizzate dal progetto.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.3. Librerie C++ per la fattorizzazione di Cholesky

2.3.1. SuiteSparse

SuiteSparse fornisce algoritmi altamente ottimizzati per matrici sparse. In particolare, abbiamo integrato CHOLMOD (con le sue dipendenze), la componente specializzata per la fattorizzazione di Cholesky di matrici sparse simmetriche definite positive, con licenza GNU LGPL.

CHOLMOD offre prestazioni superiori rispetto altre implementazioni per matrici di grandi dimensioni grazie a:

- Algoritmi di ordinamento avanzati (AMD, COLAMD, METIS) che riducono il fill-in durante la fattorizzazione
- Decomposizione supernodale che sfrutta operazioni BLAS di livello 3
- Supporto per calcoli multithreaded che sfruttano processori multi-core
- Gestione ottimizzata della memoria che riduce il sovraccarico per matrici molto sparse

2.3.2. Eigen

Eigen è una libreria C++ header-only di algebra lineare ad alte prestazioni, completamente sviluppata in template per massimizzare l'ottimizzazione in fase di compilazione, con licenza MPL2.

Una caratteristica distintiva di Eigen è la sua architettura estensibile che permette l'integrazione con diverse librerie esterne specializzate. Nel nostro progetto, abbiamo scelto di utilizzare l'interfaccia con CHOLMOD di SuiteSparse:

- **Interfaccia CHOLMOD**: Abbiamo sfruttato principalmente il modulo `CholmodSupport` di Eigen che permette di utilizzare gli algoritmi avanzati di CHOLMOD mantenendo la sintassi familiare di Eigen.

- **Alternative considerate**: Sarebbe stato possibile utilizzare l'implementazione nativa di Eigen (`SimplicialLLT`) con o senza supporto BLAS/LAPACK, che risulta adeguata per matrici di dimensioni moderate, ma dato che volevamo basarci sull'implementazione di MATLAB abbiamo optato per l'altra strada.
- **Alternative proprietarie**: Eigen supporta anche interfacce verso librerie proprietarie come Pardiso di Intel oneAPI e Accelerate di Apple, che offrono implementazioni altamente ottimizzate ma non open-source.

Per la fattorizzazione di Cholesky, il nostro approccio primario è stato:
`CholmodSupport::CholmodDecomposition` che delega il calcolo effettivo a CHOLMOD, beneficiando degli algoritmi di ordinamento avanzati e dell'ottimizzazione per sistemi multi-core e scegliendo in automatico che algoritmo di ordinamento usare e che tipo di fattorizzazione (supermodal vs simplicial).
La flessibilità di Eigen ci ha permesso di integrare efficacemente la potenza di CHOLMOD mantenendo un'interfaccia coerente e familiare nel codice principale, senza compromettere l'approccio open-source del progetto.

2.3.3. Fast Matrix Market

Per la lettura delle matrici sparse dal formato Matrix Market (MTX), abbiamo integrato la libreria Fast Matrix Market, con licenza BSD-2. Questa libreria ha consentito di importare efficientemente dataset di test di grandi dimensioni.
Fast Matrix Market si distingue per:

- Lettura parallelizzata che sfrutta tutti i core disponibili
- Parsing efficiente che riduce significativamente i tempi di caricamento
- Integrazione diretta con Eigen senza necessità di conversioni intermedie
- Supporto per diverse precisioni numeriche (float, double, complex)

2.4. Librerie BLAS e LAPACK

Le librerie BLAS (Basic Linear Algebra Subprograms) e LAPACK (Linear Algebra PACKage) rappresentano fondamenti essenziali per l'algebra lineare computazionale. Queste librerie standardizzate forniscono implementazioni ottimizzate di operazioni matriciali e vettoriali di base che costituiscono i blocchi fondamentali per algoritmi più complessi, inclusa la fattorizzazione di Cholesky.

2.4.1. Intel MKL

Intel Math Kernel Library (MKL) rappresenta l'implementazione commerciale di riferimento per BLAS e LAPACK, sviluppata e ottimizzata specificamente per processori Intel. Questa libreria offre prestazioni eccezionali su architetture x86 e x86-64 grazie a:

- Ottimizzazioni a livello di microarchitettura che sfruttano set di istruzioni specifici (AVX, AVX2, AVX-512)
- Parallelizzazione automatica che utilizza efficacemente processori multi-core
- Gestione intelligente della cache e della memoria per massimizzare il throughput
- Routine specializzate per matrici sparse che riducono significativamente il tempo di calcolo

2.4.2. OpenBLAS

OpenBLAS rappresenta l'alternativa open source più matura a Intel MKL, offrendo prestazioni competitive su diverse architetture hardware. Questa libreria deriva dal progetto GotoBLAS2 e si distingue per:

- Ottimizzazioni specifiche per diverse architetture (Intel, AMD, ARM, POWER)
- Supporto per parallelismo multi-thread attraverso implementazione OpenMP
- Compatibilità con l'interfaccia CBLAS standard
- Prestazioni scalabili fino a 256 core

2.4.3. Apple Accelerate

Accelerate è il framework di calcolo numerico sviluppato da Apple e integrato nativamente nei sistemi operativi macOS e iOS. Include implementazioni ottimizzate di BLAS e LAPACK specificamente progettate per l'hardware Apple, inclusi i processori M-Series basati su architettura Apple Silicon (ARM).

Caratteristiche distintive di Accelerate includono:

- Ottimizzazioni specifiche per chip Apple Silicon
- Integrazione profonda con l'ecosistema di librerie Apple e supporto per tecnologie come Grand Central Dispatch
- Supporto per calcoli vettoriali SIMD attraverso il framework vDSP
- Bilanciamento automatico tra prestazioni ed efficienza energetica

2.5. Implementazione in C++

2.5.1. Considerazioni generali

Avendo accesso al codice sorgente volendo è possibile adattare il codice alla risoluzione di un problema specifico, nel nostro caso abbiamo mantenuto un implementazione piuttosto generica, date le diverse matrici da trattare.

2.5.2. Parametri analizzati

A differenza di MATLAB, andiamo a ottenere anche il tipo di BLAS e il numero di thread utilizzati per l'esecuzione, nello specifico abbiamo misurato:

Tempo di esecuzione:

- **loadTime:** tempo necessario per caricare la matrice dal file in formato Matrix Market (MTX) (ms)
- **decompTime:** tempo per eseguire la fattorizzazione di Cholesky (ms)
- **solveTime:** tempo per risolvere un sistema lineare usando i fattori (ms)

Per misurare il tempo con precisione, abbiamo utilizzato le funzionalità della libreria standard C++:

```
auto start = std::chrono::high_resolution_clock::now();
// Operazione da misurare
auto end = std::chrono::high_resolution_clock::now();
auto duration =
    std::chrono::duration_cast<std::chrono::milliseconds>(
        end - start
    ).count();
```

Utilizzo di memoria:

- loadMem: memoria utilizzata per caricare la matrice (Bytes)
- decompMem: memoria utilizzata per la fattorizzazione (Bytes)
- solveMem: memoria utilizzata per trovare la soluzione (Bytes)

Il calcolo della memoria per le operazioni di caricamento della matrice è stato implementato manualmente, considerando:

```
size_t valuesSize = A.nonZeros() * sizeof(double);
size_t innerIndicesSize = A.nonZeros() * sizeof(int64_t);
size_t outerIndicesSize = (A.outerSize() + 1) * sizeof(int64_t);
auto loadMem = valuesSize + innerIndicesSize + outerIndicesSize;
```

Invece per il calcolo della memoria per le operazioni di fattorizzazione e risoluzione, abbiamo modificato parte del codice di CHOLMOD, aggiungendo un contatore per la memoria allocata. Questo contatore viene resettato prima di ogni operazione e aggiornato durante l'allocazione della memoria.

```
solver.cholmod().memory_allocated = 0;  // Reset contatore
// Operazione da misurare
auto operationMem = solver.cholmod().memory_allocated;
```

Accuratezza:

- Errore Relativo: errore relativo della soluzione calcolata rispetto alla soluzione attesa

Per ridurre l'errore nel calcolo dell'errore evitando il calcolo una delle due radici, abbiamo ricavato la seguente formula:

$$\sqrt{\frac{\|x - x_e\|^2}{\|x\|^2}} = \frac{\|x - x_e\|_2}{\|x\|_2} \tag{6}$$

Dove dato

$$\frac{\|x - x_e\|_2}{\|x\|_2} = \frac{\sqrt{(x - x_e) \cdot (x - x_e)}}{\sqrt{x \cdot x}} \tag{7}$$

con $\cdot$ prodotto scalare tra vettori, ho che

$$\frac{\|x - x_e\|^2}{\|x\|^2} = \frac{(x - x_e) \cdot (x - x_e)}{x \cdot x} \tag{8}$$

ovvero le somme delle componenti del vettore al quadrato.

2.5.3. Implementazione della fattorizzazione di Cholesky

In linea con l'approccio MATLAB, abbiamo implementato la fattorizzazione di Cholesky utilizzando la libreria CHOLMOD attraverso l'interfaccia fornita da Eigen:

```
Eigen::CholmodDecomposition<SparseMatrix> solver;
solver.compute(A); // Fattorizzazione della matrice A
xe = solver.solve(b); // Risoluzione del sistema Ax = b
```

Questa interfaccia consente di sfruttare le ottimizzazioni avanzate di CHOLMOD, inclusi gli algoritmi di ordinamento e la decomposizione supernodale, mantenendo al contempo la sintassi familiare di Eigen. La libreria CHOLMOD gestisce automaticamente la scelta dell'algoritmo di

ordinamento e del tipo di composizione più adatto in base alla struttura della matrice, ottimizzando così le prestazioni della fattorizzazione.

2.6. Documentazione e Integrazione Librerie C++

Per integrare efficacemente le librerie C++ nel nostro progetto, abbiamo dovuto affrontare diverse sfide legate alla documentazione e alla configurazione.

2.6.1. Eigen

La documentazione di Eigen rappresenta un eccellente esempio di riferimento tecnico per progetti open-source:

Completezza: Tutorial dettagliati, guida per le classi e documentazione delle API generata con Doxygen. Esempi: Numerosi esempi di codice che coprono tutti i moduli principali. Integrazione: Essendo header-only, l'integrazione richiede solo l'inclusione dei file header senza necessità di linking. Moduli esterni: La documentazione sul modulo CholmodSupport è più limitata rispetto ai moduli principali, richiedendo talvolta la consultazione del codice sorgente. L'integrazione di Eigen nel progetto è stata generalmente agevole grazie alla semplicità del modello header-only e ai chiari esempi disponibili nella documentazione ufficiale.

2.6.2. SuiteSparse

La documentazione di SuiteSparse, e in particolare di CHOLMOD, presenta caratteristiche distintive:

Documentazione scientifica: Articoli accademici dettagliati che descrivono gli algoritmi implementati. Documentazione tecnica: File README e documentazione interna al codice che descrivono l'API C. Limitazioni: Minore enfasi sugli esempi di integrazione in progetti C++ moderni. Build system: Documentazione limitata sull'integrazione con sistemi di build. Nonostante l'eccellente documentazione degli algoritmi sottostanti, l'integrazione di SuiteSparse ha richiesto maggiore impegno, specialmente per configurare correttamente le dipendenze tra i vari componenti. T. Davis [1]

2.6.3. Fast Matrix Market

La libreria Fast Matrix Market offre una documentazione concisa ma efficace:

GitHub README: Documenta chiaramente l'API principale e i casi d'uso comuni. Esempi: Include esempi di integrazione con Eigen che hanno facilitato significativamente l'adozione. Integrazione CMake: Fornisce configurazioni CMake moderne con supporto per find_package. L'integrazione di Fast Matrix Market è stata notevolmente semplice grazie alla documentazione mirata e agli esempi pratici, permettendo una rapida implementazione della lettura di matrici sparse in formato MTX.

2.6.4. Librerie BLAS e LAPACK

Le sfide più significative nel progetto sono emerse dall'integrazione delle implementazioni BLAS e LAPACK:

Documentazione frammentata: Ogni implementazione (Intel MKL, OpenBLAS, Accelerate) presenta una propria documentazione con convenzioni e approcci di configurazione diversi, ma questo non ha rappresentato il problema principale.

Difficoltà CMake: Abbiamo riscontrato notevoli difficoltà nell'integrazione attraverso CMake:
Mancanza di moduli CMake standardizzati per il rilevamento delle diverse implementazioni, rendendo inefficaci i moduli standard come FindBLAS e FindLAPACK. Necessità di linkare manualmente le librerie specificando esattamente i percorsi e i componenti richiesti, invece di poter utilizzare i meccanismi automatizzati di CMake. Configurazioni diverse richieste per Windows (MKL/MSVC) e Linux (OpenBLAS/GCC). Conflitti di simboli: In alcuni casi, quali l'utilizzo dell'interfaccia standard ILP64 LAPACK 3.11.0 dell'implementazione di Apple Accelerate ha causato conflitti di simboli difficili da risolvere.

2.6.5. Conclusioni

Dall'esperienza di integrazione delle diverse librerie, abbiamo tratto importanti conclusioni:
Le librerie con documentazione orientata agli esempi (Eigen, Fast Matrix Market) hanno richiesto tempi di integrazione significativamente minori.
Le dipendenze transitive non documentate tra librerie C/C++ rappresentano una sfida significativa per l'integrazione tramite CMake.
L'approccio più efficace è risultato essere lo sviluppo di configurazioni CMake modulari che isolano le complessità di ogni libreria.
La documentazione delle librerie di algebra lineare spesso privilegia la descrizione matematica degli algoritmi a scapito dei dettagli di integrazione tecnica.
Queste sfide di integrazione, sebbene impegnative, hanno permesso di sviluppare un sistema robusto e flessibile che può adattarsi a diverse implementazioni BLAS/LAPACK mantenendo un'interfaccia coerente attraverso Eigen.

2.7. Commenti

Il nostro progetto ha dimostrato con successo l'integrazione di librerie specializzate per l'algebra lineare in un ecosistema C++ moderno. Utilizzando Eigen come interfaccia ad alto livello e SuiteSparse (in particolare CHOLMOD) come motore di calcolo per la fattorizzazione di Cholesky, siamo riusciti a costruire un sistema flessibile e performante, capace di gestire matrici sparse di grandi dimensioni.
Un aspetto distintivo della nostra implementazione è stata la capacità di sfruttare diverse implementazioni di BLAS e LAPACK (Intel MKL, OpenBLAS, Accelerate), permettendoci di confrontare direttamente le prestazioni di soluzioni commerciali e open-source. Questa flessibilità ci ha consentito di simulare efficacemente il comportamento di MATLAB, che utilizza internamente CHOLMOD con implementazioni BLAS ottimizzate.
L'obiettivo principale dell'esperimento era verificare se un'alternativa completamente open-source potesse offrire prestazioni paragonabili alla soluzione commerciale di MATLAB.
Per quanto l'ideazione di una soluzione artigianale possa sembrare complicato, lo sviluppo del codice in se è stata forse la parte meno impegnativa. Maggiori difficoltà invece, le abbiamo incontrate nell'integrazione delle diverse librerie, specialmente quelle legate a BLAS e LAPACK, hanno rivelato la necessità di migliorare gli strumenti di build e la documentazione per questi componenti fondamentali dell'ecosistema di calcolo scientifico.
Nonostante queste sfide, il nostro progetto dimostra che è possibile costruire una piattaforma di calcolo numerico avanzata basata interamente su tecnologie open-source, offrendo un'alternativa

valida a soluzioni commerciali come MATLAB per applicazioni che richiedono la fattorizzazione di Cholesky su matrici sparse di grandi dimensioni.

3. Risultati

3.1. Specifiche del sistema

Specifiche del sistema per Windows:

- **Processore**: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
- **Architettura**: Sistema operativo a 64 bit, processore basato su x64
- **RAM installata**: 16 GB
- **Archiviazione**: 238 GB SSD e 1 TB SSD Esterno
- **Scheda grafica**: NVIDIA GeForce GTX 1060 with Max-Q Design (6 GB)
- **Memoria Virtuale**: 7680 MB su SSD e 32768 MB su HDD Esterno
- **Sistema operativo Windows**: Windows 10 Home

Specifiche del sistema per Linux:

Processore, Architettura, RAM, Archiviazione e Scheda Grafica sono gli stessi del sistema Windows.

- **Memoria Virtuale**: 40448 MB su HDD Esterno
- **Sistema operativo Linux**: WSL 2 con Ubuntu 25.04

La memoria virtuale era possibile averla solo su un solo disco, quindi è stata scelta (per mancanza di spazio) quella dell'HDD esterno.

Specifiche del sistema per MacOS:

- **Processore**: Apple M1 Pro
- **Architettura**: ARM64 (10 core)
- **RAM installata**: 16 GB
- **Archiviazione**: 1 TB SSD
- **Scheda grafica**: Apple M1 Pro GPU (16 core)
- **Memoria Virtuale**: 56 GB su SSD
- **Sistema operativo**: MacOS Sonoma 15.4

3.2. Matrici analizzate

Ordinate in base al numero di elementi non zero, le matrici analizzate sono:

Nome Matrice	Righe & Colonne	Valori non zero
ex15	6867	98671
shallow_water1	81920	327680
cfd1	70656	1828364
cfd2	123440	3087898
parabolic_fem	525825	3674625
apache2	715176	4817870
G3_circuit	1585478	7660826
StocF-1465	1465137	21005389
Flan_1565	1564794	117406044

Tabella 1: Matrici analizzate

Tutte queste matrici sono sparse, simmetriche e positive definite. Sono state scaricate dal repository https://sparse.tamu.edu/.

3.3. Considerazioni Iniziali

Dato che sia MATLAB che C++ utilizzano la stessa libreria CHOLMOD, ci si aspetta che i risultati siano simili. Tuttavia, potrebbero emergere delle differenze, soprattutto a causa della versione obsoleta di MATLAB, meno aggiornata rispetto a quella di C++. Inoltre, l'uso differente delle librerie BLAS potrebbe influenzare ulteriormente i risultati.

Prima di andare a fare il confronto tra MATLAB e C++, è necessario analizzare i risultati di MATLAB e di C++, in modo da avere un'idea di cosa aspettarsi.

3.4. Risultati MATLAB

Durante l'esecuzione, due matrici hanno causato un errore interno della libreria CHOLMOD in Windows, mentre in Linux hanno provocato la terminazione forzata del processo. Di conseguenza, non sono state incluse nei risultati. Le matrici problematiche sono:

- Flan_1565
- StocF-1465

3.4.1. Memoria

Matrice	Uso Memoria Windows (MB)			Uso Memoria Linux (MB)		
	load	decomp	solve	load	decomp	solve
ex15	0	3.48	0	2.15	3.86	0.34
shallow_water1	5.02	36.05	0	6.16	37.5	0.91
cfd1	27.91	560.29	0	27.34	548.81	0.83
cfd2	55.51	1114.18	0	51.04	1124.59	1.23
parabolic_fem	64.23	560.49	4.02	64.64	558.96	4.3
apache2	79.13	2722.27	5.46	80.28	1.76e13	5.74
G3_circuit	129.25	2915.38	12.13	143.25	1.76e13	12.39

Tabella 2: Memoria utilizzata dai sistemi operativi su MATLAB

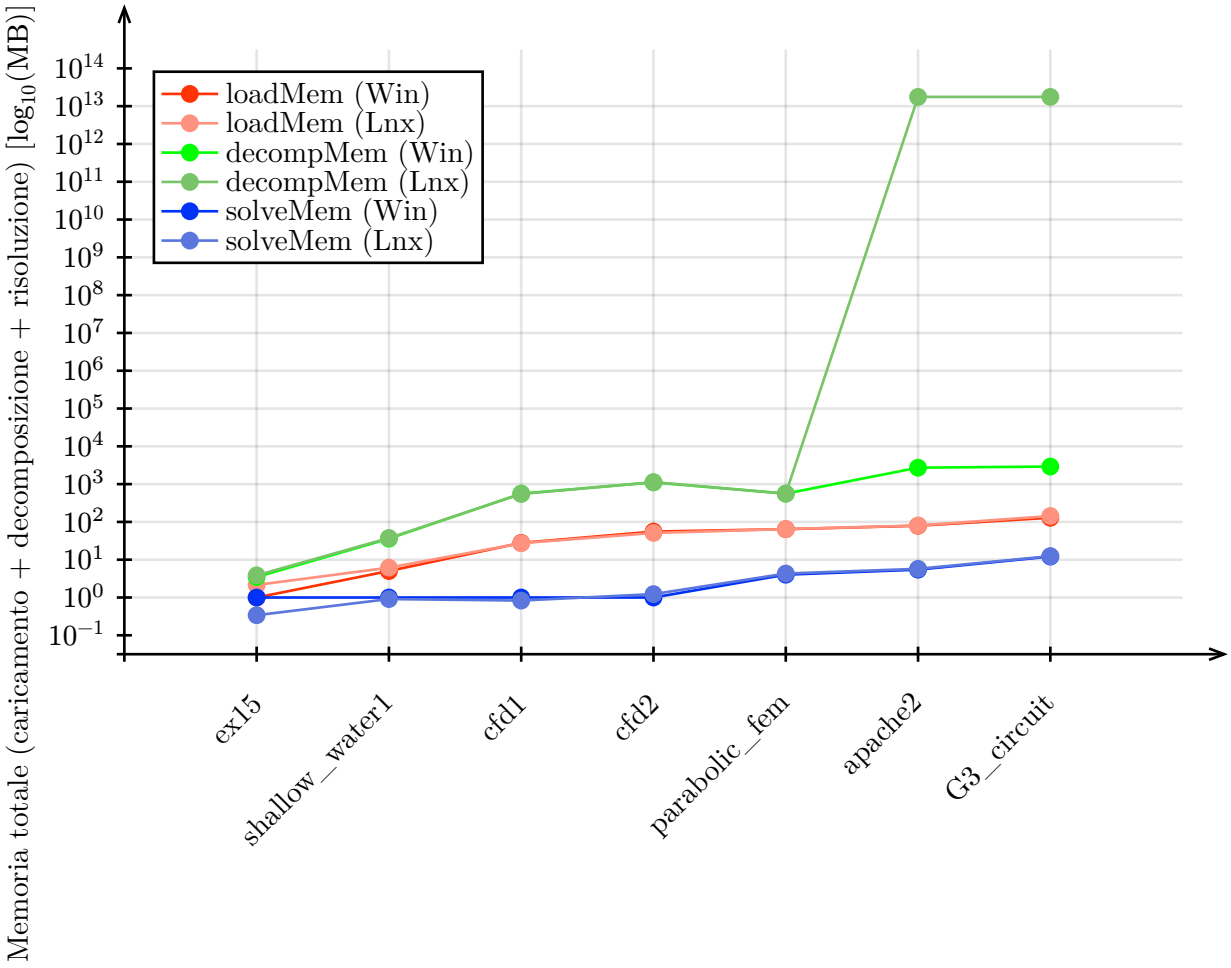

Figura 1: Confronto utilizzo memoria tra sistemi operativi su MATLAB

Dalla tabella emerge chiaramente che il profiler di memoria di MATLAB potrebbe non essere completamente affidabile, poiché riporta valori pari a zero per il caricamento di alcune matrici e

valori anomali per la decomposizione delle matrici Apache e G3_Circuit su Linux. Questo fenomeno dei valori anomali è probabilmente dovuto alla presenza di WSL2 e all'utilizzo della libreria esterna CHOLMOD, ma dipende anche dal metodo con cui MATLAB registra l'uso della memoria. Se il profiler utilizza un approccio basato sul campionamento, allora i valori pari a zero sarebbero comprensibili.

In generale, l'uso della memoria sembra aumentare con la dimensione della matrice e risulta maggiore per la decomposizione rispetto al caricamento e alla risoluzione.

3.4.2. Tempi

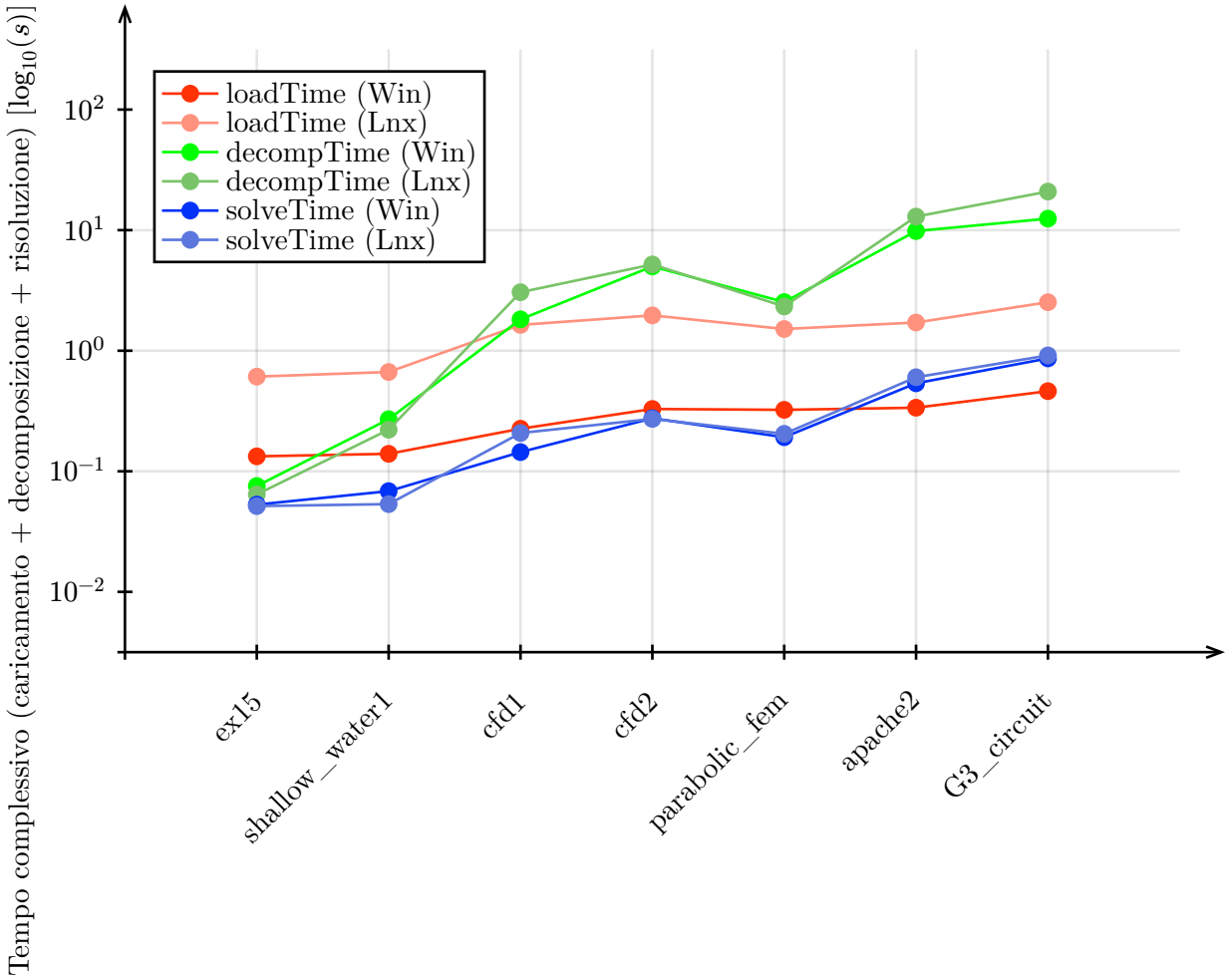

Figura 2: Confronto tempi tra sistemi operativi su MATLAB

Analizzando i tempi separatamente, notiamo che l'unica grande differenza tra i due sistemi operativi riguarda il tempo di caricamento, significativamente più elevato in Linux. Questo è probabilmente dovuto al fatto che, su Linux, utilizziamo WSL2, il quale non ha accesso diretto all'hardware e deve operare attraverso un layer di virtualizzazione.

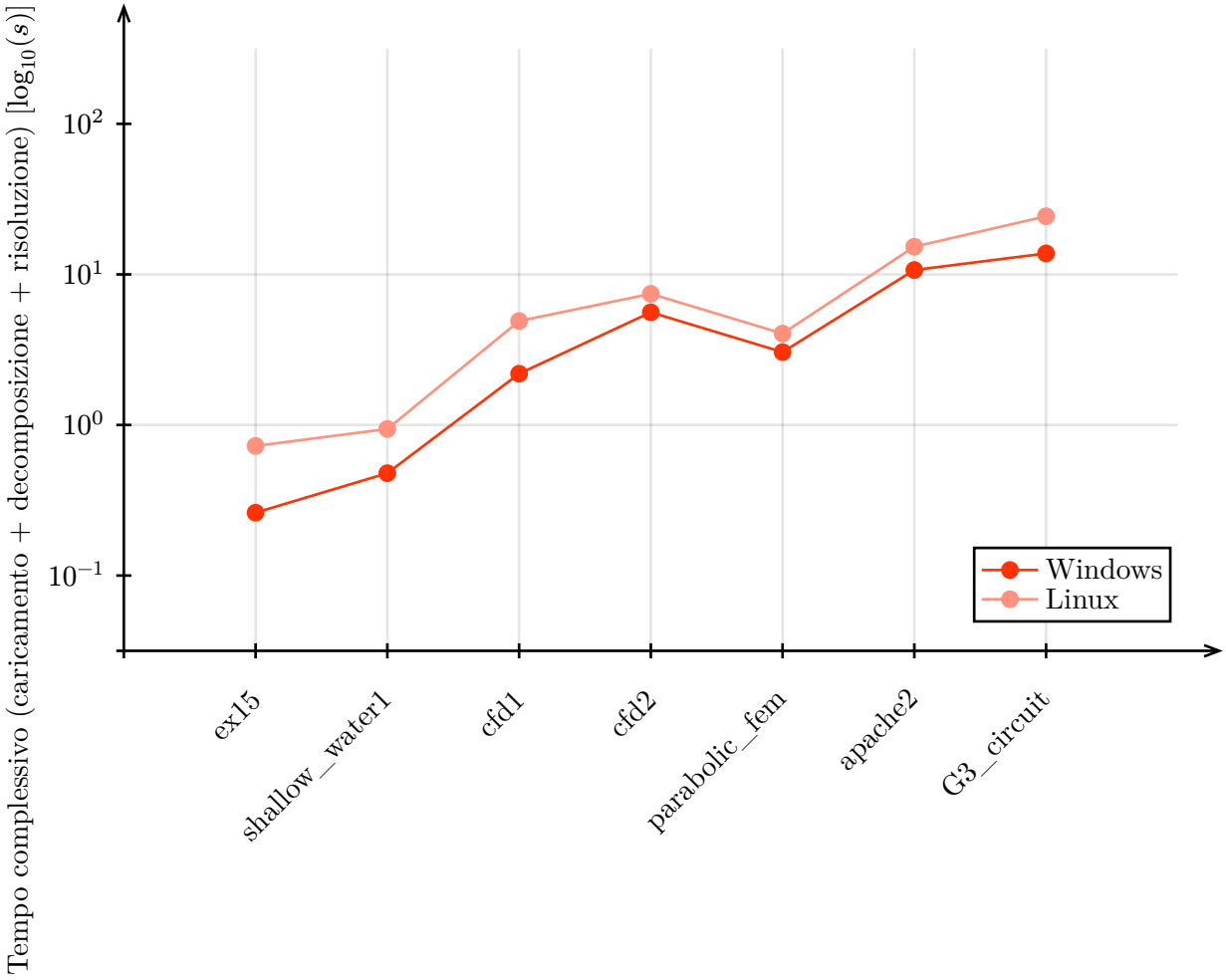

Figura 3: Confronto tempo complessivo tra sistemi operativi su MATLAB

Se osserviamo i tempi complessivi, non si nota una grande differenza tra i due sistemi operativi. Windows sembra più veloce, ma è importante considerare che il tempo di caricamento è significativamente più alto in Linux, il che contribuisce a un tempo complessivo maggiore.

3.4.3. Errore Relativo

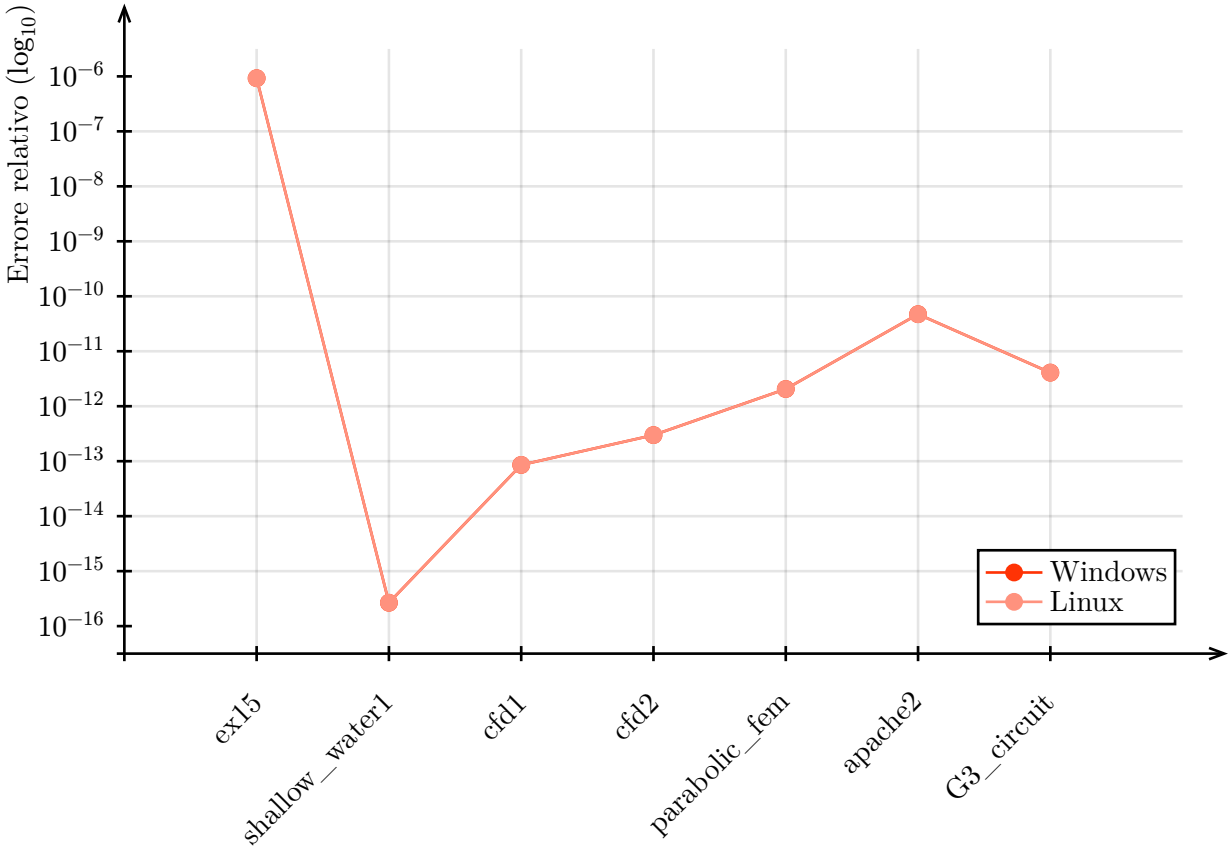

Figura 4: Confronto errore relativo tra sistemi operativi su MATLAB

Osservando l'errore, notiamo che è identico su entrambe le piattaforme, quindi in MATLAB non si riscontra alcuna differenza tra i due sistemi operativi. Questo è comprensibile, dato che vengono utilizzate la stessa libreria CHOLMOD e la stessa libreria BLAS.

3.5. Risultati C++

A differenza di MATLAB, C++ è riuscito a completare tutte le matrici, inoltre avendo accesso a più informazioni, siamo andati a vedere quanti thread venissero utilizzati da BLAS.

3.5.1. Threads

- Windows/Linux - MKL - 4 threads
- Windows/Linux - OpenBLAS - 8 threads
- macOS - Accelerate - (-1) threads (numero di thread scelti dinamicamente)
- macOS - OpenBLAS - 10 threads

Da questi dati, si nota che è l'architettura del sistema a determinare il numero di thread utilizzati, e non il sistema operativo. Inoltre, ci aspettiamo che macOS ottenga risultati leggermente migliori grazie alla possibilità di utilizzare più thread, con un impatto più evidente sulle matrici di dimensioni maggiori.

3.5.2. Memoria

La memoria è identica nei tre sistemi operativi, quindi non è necessario ripeterla per ciascuno. Questo dipende dal metodo di allocazione interna di CHOLMOD e ha senso, dato che la memoria è determinata principalmente dall'architettura del sistema piuttosto che dal sistema operativo.

Nome Matrice	**Mem Load (MB)**	**Mem Decomp (MB)**	**Picco Mem Decomp (MB)**	**Mem Risoluzione (MB)**	**Picco Mem Risoluzione (MB)**
ex15	1.56	9.6	4.21	0.11	0.11
shallow_water1	5.63	68.41	46.23	1.25	1.25
cfd1	28.44	331.16	221.42	1.09	1.09
cfd2	48.06	581.7	396.06	1.89	1.89
parabolic_fem	60.08	691.59	471.25	8.03	8.03
apache2	78.97	1734.88	1407.36	10.93	10.93
G3_circuit	128.99	1777.83	1224.56	24.2	24.2
StocF-1465	331.69	11411.76	10103.63	22.41	22.41
Flan_1565	1803.41	21283.12	14451.67	23.92	23.92

Tabella 3: Confronto utilizzo memoria tra sistemi operativi su C++

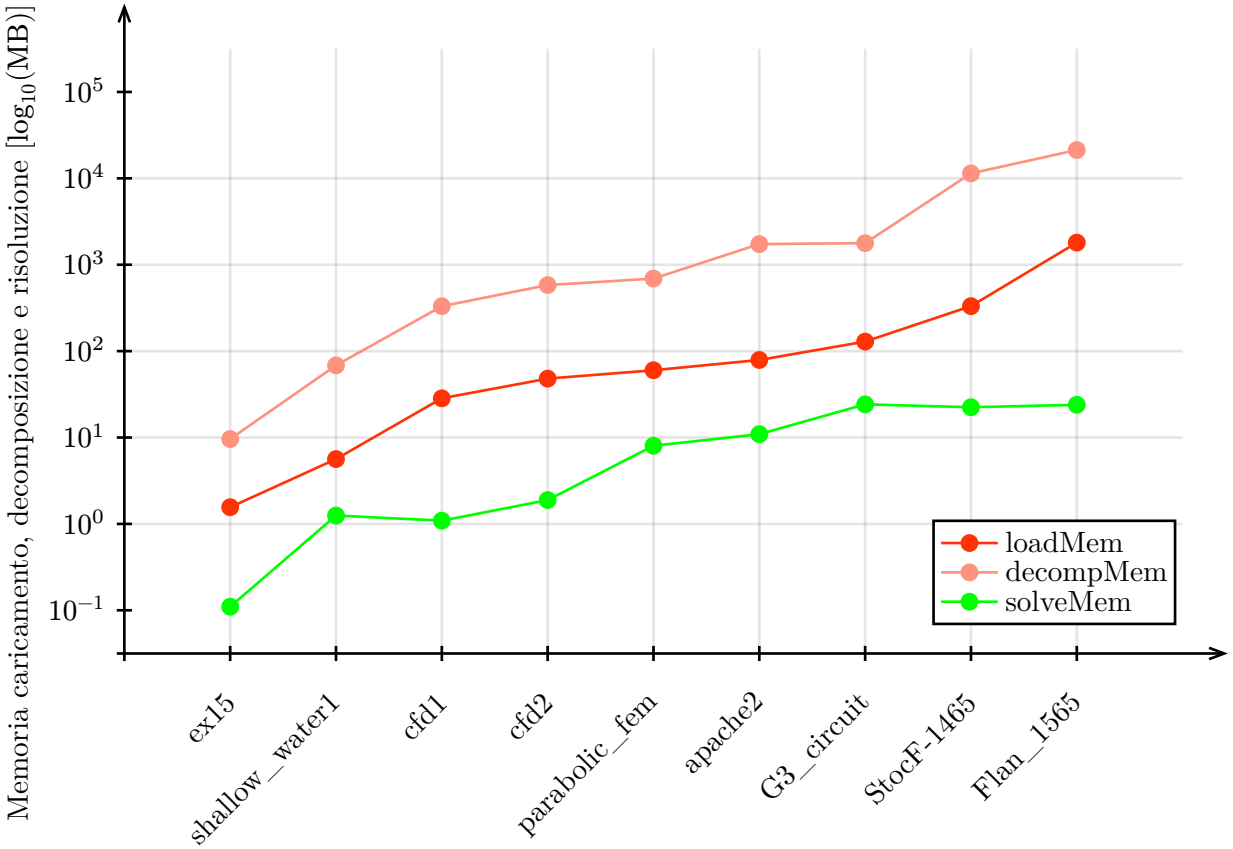

Figura 5: Confronto utilizzo memoria tra sistemi operativi su C++

Analizzando la memoria, notiamo che l'uso maggiore avviene durante la decomposizione e il caricamento della matrice. Questo è comprensibile, poiché nel processo di risoluzione si utilizza il risultato della decomposizione. Inoltre, l'utilizzo della memoria sembra aumentare con l'incremento della dimensione della matrice.

3.5.3. Tempi

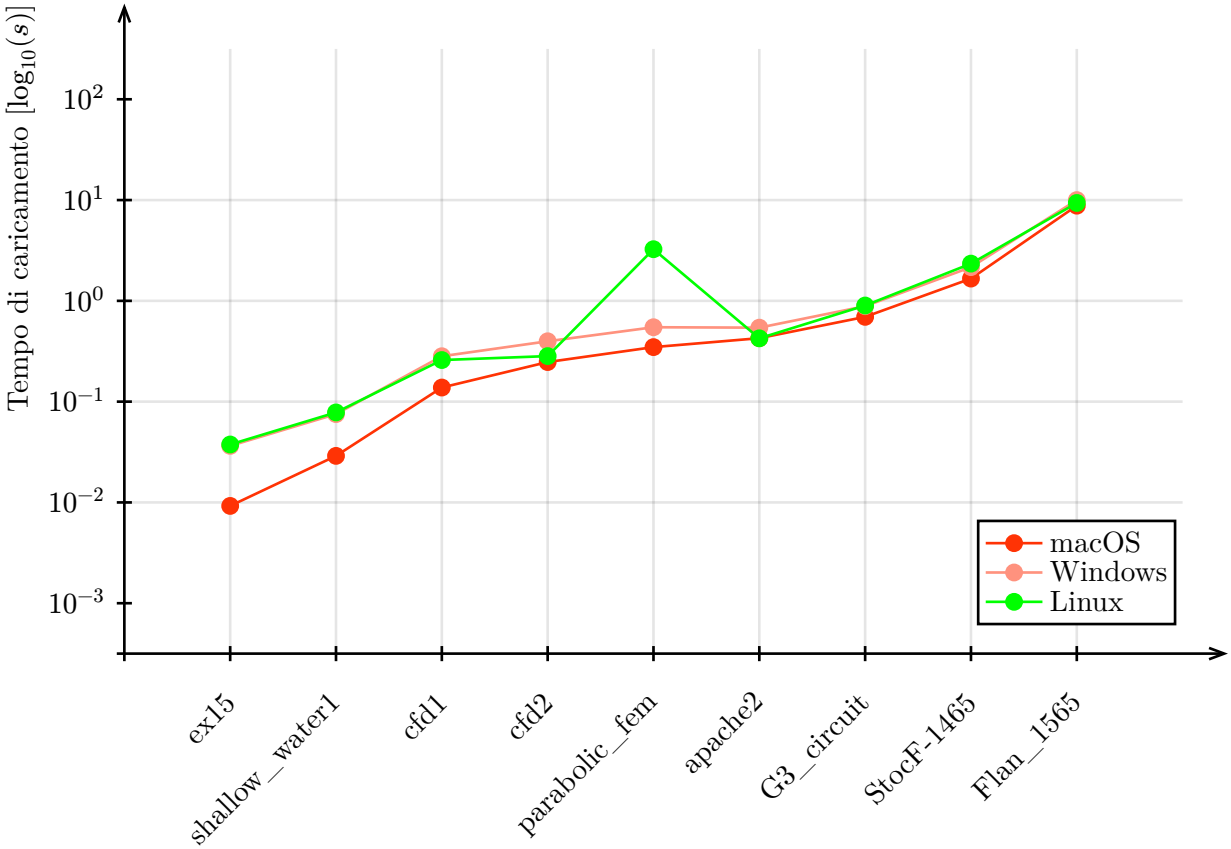

Figura 6: Confronto tempo di caricamento tra sistemi operativi su C++

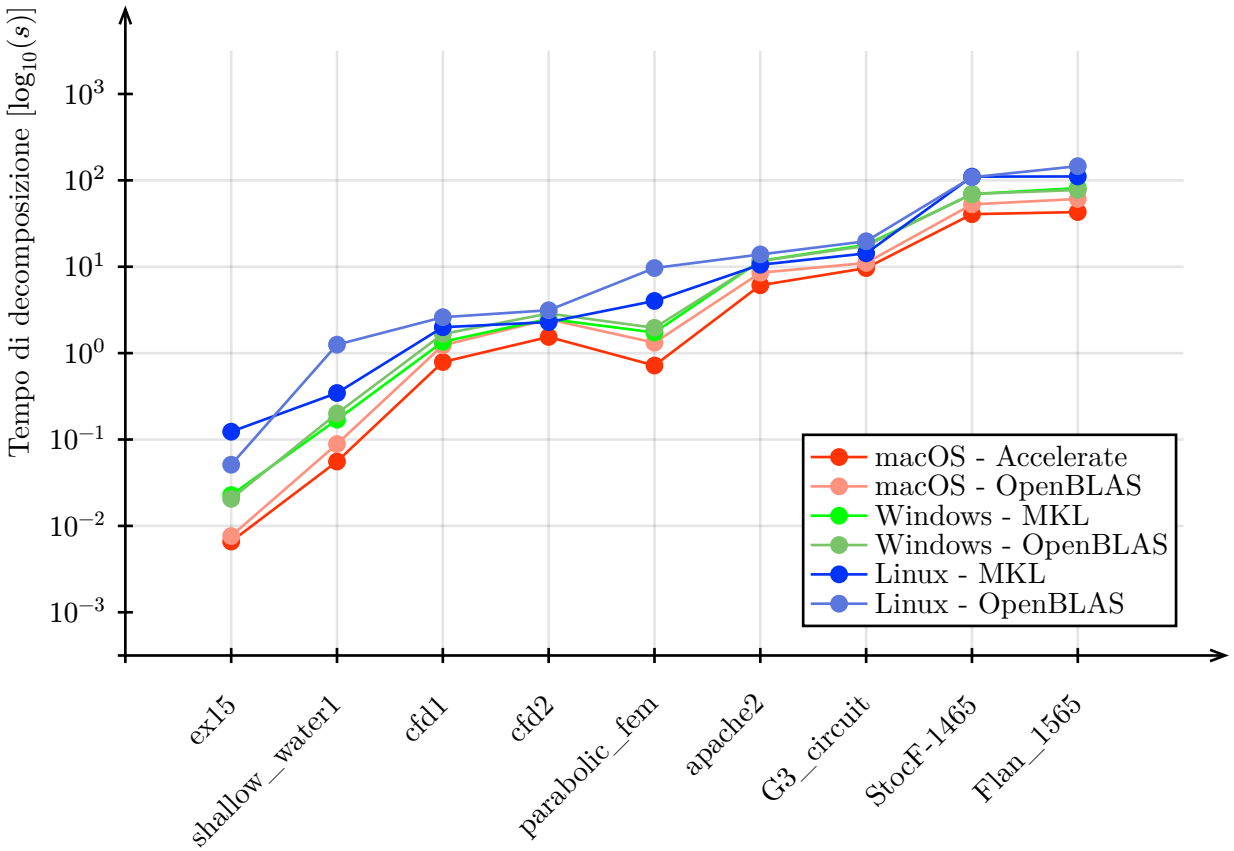

Figura 7: Confronto tempo di decomposizione tra sistemi operativi e BLAS su C++

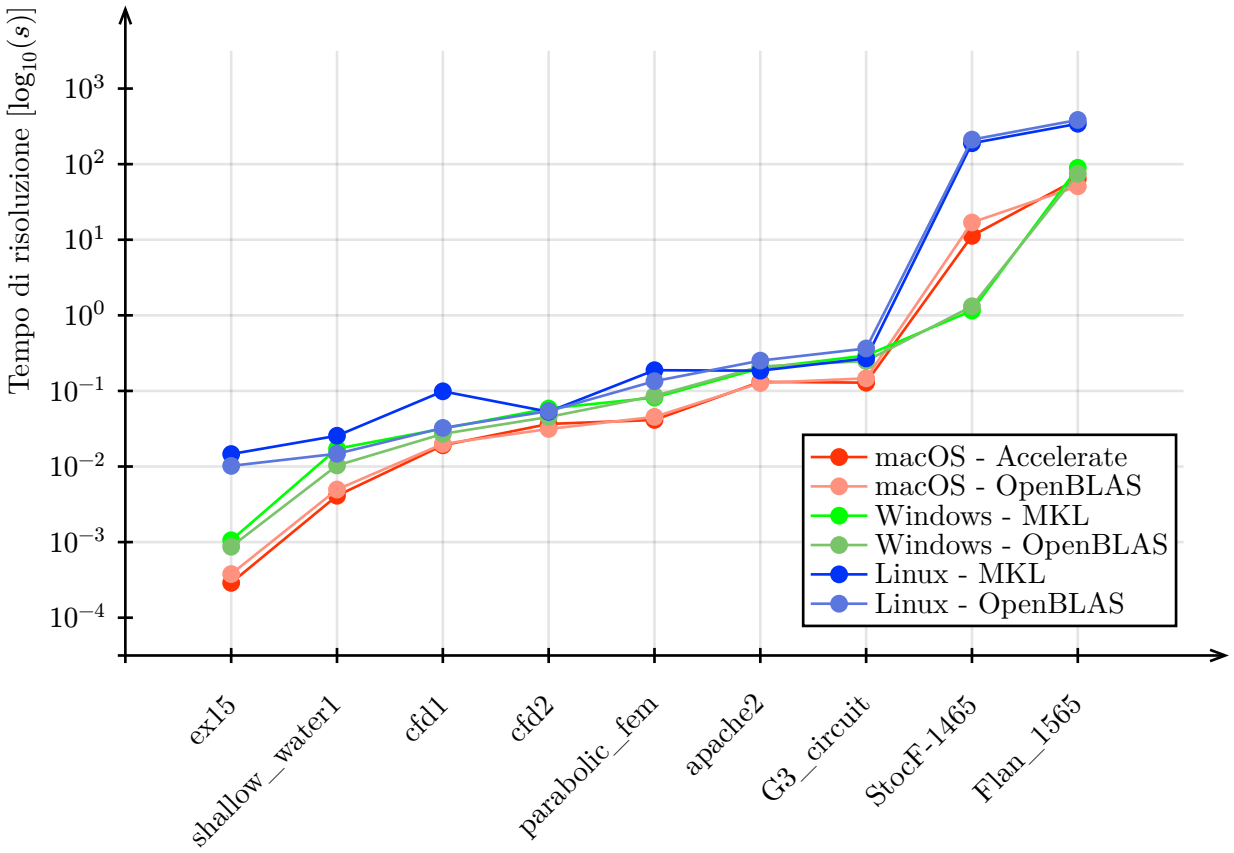

Figura 8: Confronto tempo di risoluzione tra sistemi operativi e BLAS su C++

Analizzando i tempi, notiamo che, in generale, Linux risulta più lento rispetto agli altri sistemi operativi, principalmente a causa dell'uso di WSL2. MacOS, invece, sembra essere il più veloce, anche se la differenza rispetto a Windows non è particolarmente marcata. Questo è probabilmente dovuto a un hardware superiore rispetto a quello disponibile per Windows e Linux. Inoltre, si osserva un piccolo outlier nel tempo di caricamento della matrice *parabolic_fem* su Linux.

Riepilogo dei Tempi Compessivi.

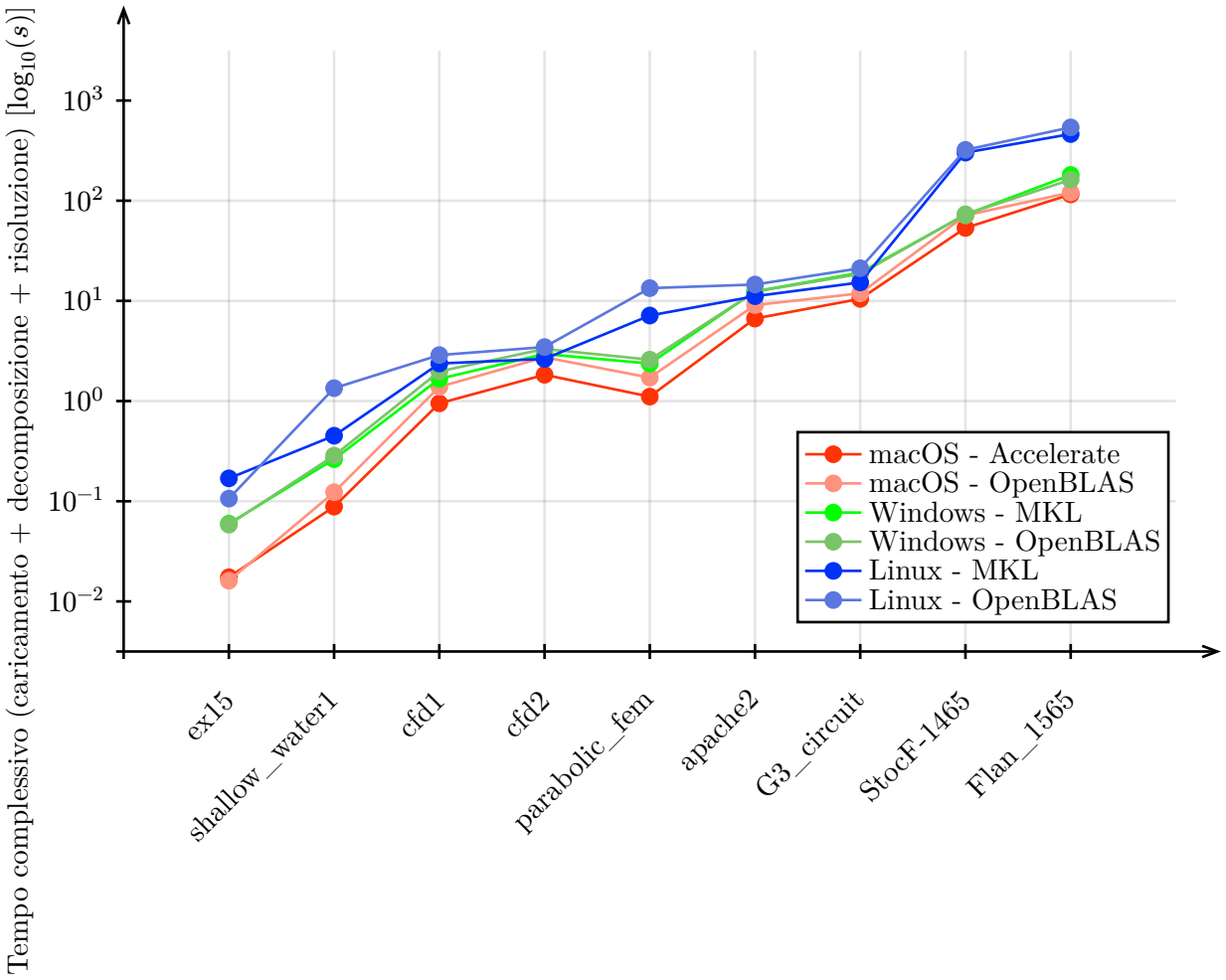

Figura 9: Confronto tempo complessivo tra sistemi operativi e BLAS su C++

Analizzando i tempi complessivi, come previsto, Linux risulta il più lento, mentre macOS è il più veloce. Tuttavia, nel complesso, la differenza tra i tre sistemi operativi non sembra essere significativa, il che rappresenta un buon risultato, poiché indica che la libreria CHOLMOD opera in modo simile su tutte le piattaforme e con diversi BLAS.

3.5.4. Errore Relativo

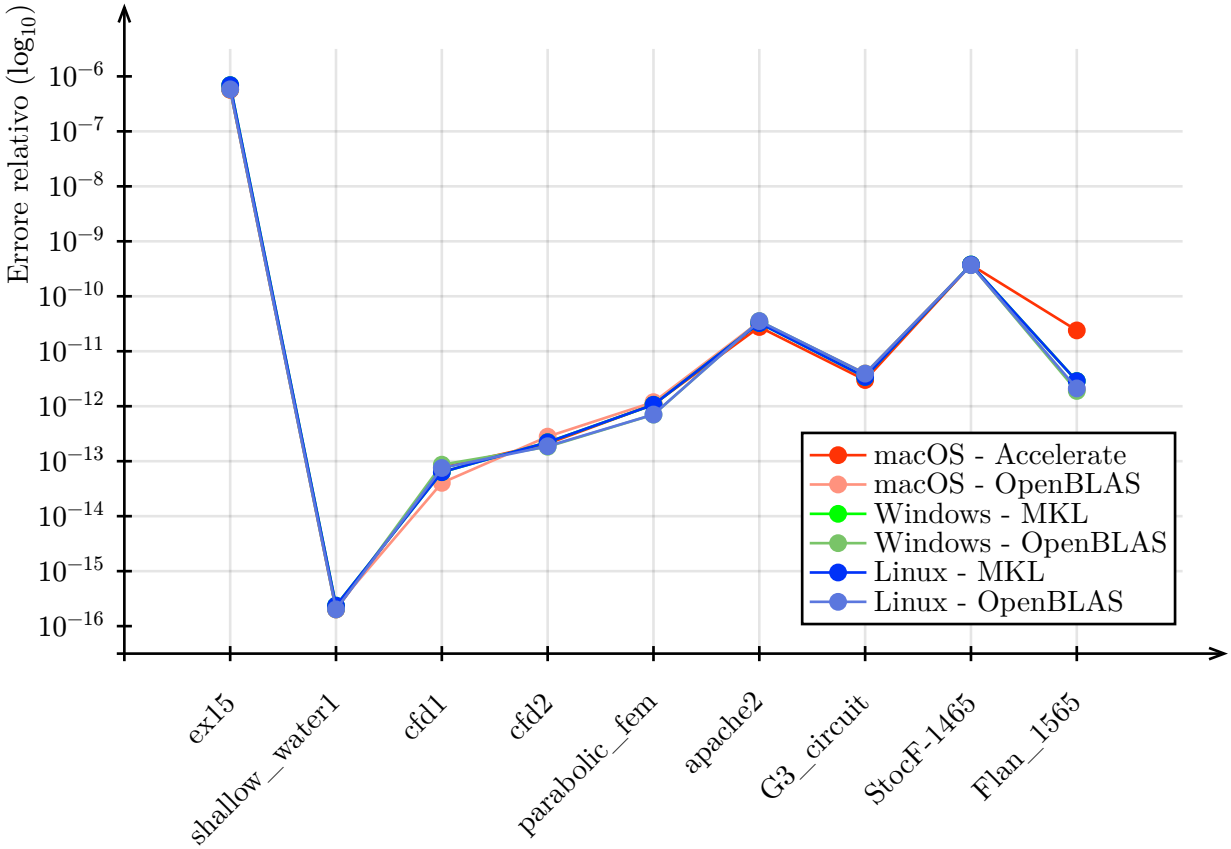

Figura 10: Confronto errore relativo tra sistemi operativi e BLAS su C++

Osservando l'errore, notiamo alcune piccole differenze tra i sistemi operativi e i BLAS, ma queste non sono significative. Questo è un buon risultato, poiché indica che la libreria CHOLMOD opera in modo simile su tutte le piattaforme e con diversi BLAS. È probabile che queste differenze siano dovute ai diversi compilatori e alle loro ottimizzazioni, tranne per un outlier nella matrice *Flan_1565* su macOS Accelerate in cui l'errore è più alto rispetto agli altri sistemi operativi e BLAS.

3.6. Confronto MATLAB e C++

Innanzitutto, è importante notare che, in MATLAB, le due matrici più grandi non sono state completate. Questo è probabilmente dovuto alla versione obsoleta di CHOLMOD presente in MATLAB, che non include miglioramenti negli algoritmi di riordinamento né correzioni degli errori nel codice.

3.6.1. Memoria

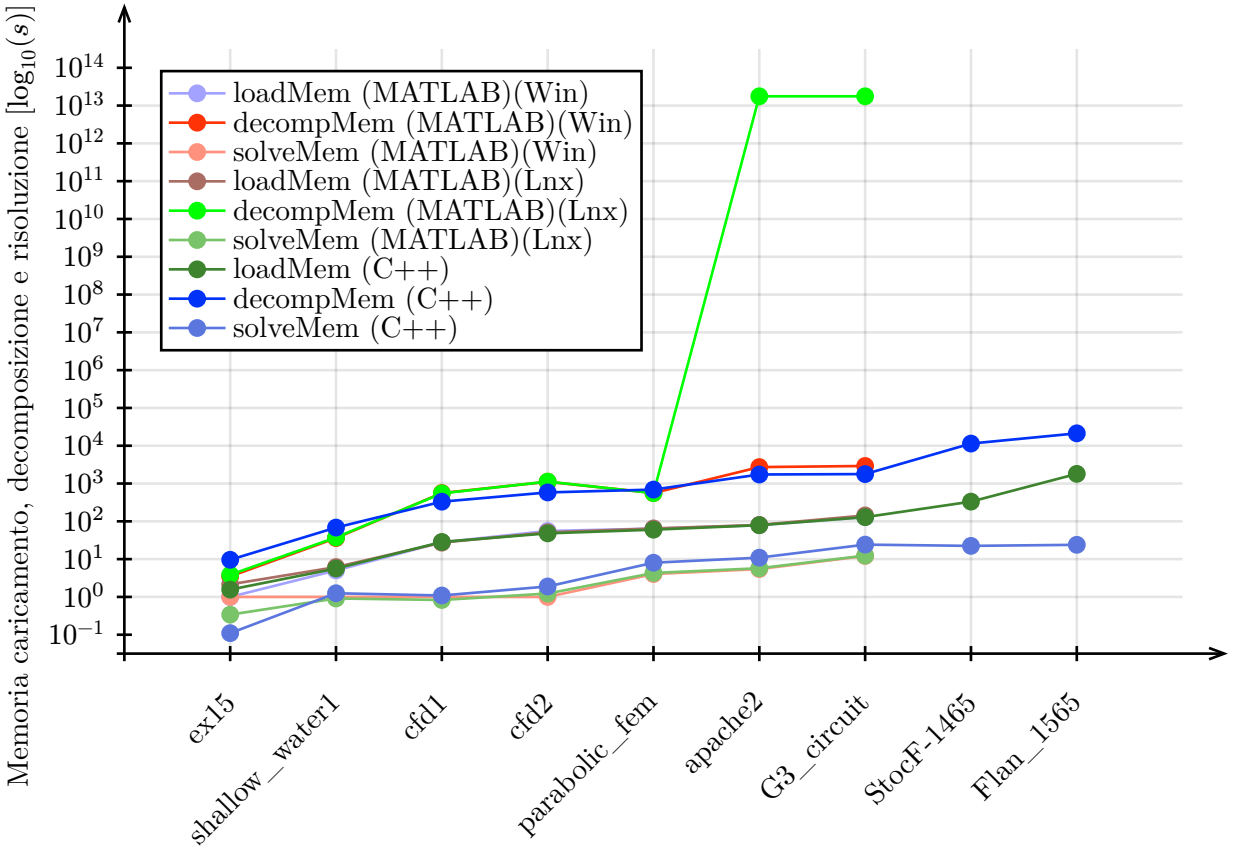

Figura 11: Confronto utilizzo memoria tra MATLAB e C++

Inanziutto dato che sono presenti due anomalie nelle matrici *apache2* e *G3_circuit* in MATLAB Linux, e quindi quindi non sono state considerate nell'analisi essendo sicuri che siano anomalie evidenti.

L'analisi dell'uso della memoria basandoci sulla Figura 11 mostra i seguenti punti salienti:

1. **Carico di memoria iniziale:** Osserviamo che le matrici più piccole, come *ex15* e *shallow_water1*, hanno un consumo ridotto di memoria, inferiore ai 10 MB. Questo è prevedibile, poiché la loro complessità computazionale è limitata. Tuttavia, quando esaminiamo matrici molto più grandi, come *Flan_1565*, notiamo un incremento drastico del carico di memoria, che supera i 1.8 GB. Questo indica che l'allocazione della memoria iniziale cresce proporzionalmente alla dimensione e alla complessità della matrice.
2. **Memoria richiesta per la decomposizione:** Il processo di decomposizione delle matrici rappresenta il momento più intensivo in termini di memoria. Ad esempio, la decomposizione della matrice *Flan_1565* richiede oltre 21 GB di memoria. Questo suggerisce che, per strutture di grande dimensione, l'algoritmo utilizzato deve gestire un enorme quantitativo di dati e operazioni, generando un picco di utilizzo. Matrici di media grandezza come *apache2* e *G3_circuit* richiedono invece circa 1.7-1.8 GB, evidenziando una crescita meno drastica ma comunque consistente. Questo è dovuto al fenomeno del

fill-in che anche se ridotto dato l'utilizzo di algoritmi di riordinamento, è comunque presente e richiede una certa quantità di memoria.

3. **Picco di memoria di decomposizione:** In diversi casi, il picco di memoria durante la fase di decomposizione è inferiore al valore totale della memoria richiesta. Questo può significare che l'allocazione della memoria varia nel tempo e viene gestita dinamicamente, evitando sprechi di risorse. In pratica, la memoria viene allocata progressivamente secondo necessità, ottimizzando l'uso delle risorse disponibili.
4. **Memoria richiesta per la risoluzione:** Un aspetto interessante è che, rispetto alla decomposizione, la fase di risoluzione della matrice ha un impatto molto più contenuto sull'utilizzo della memoria. Questo accade perché la risoluzione si basa sui risultati ottenuti in fase di decomposizione e non richiede un'elaborazione intensiva sugli stessi dati. Di conseguenza, il consumo di memoria rimane relativamente basso.
5. **Confronto memoria MATLAB e C++:** In generale, l'uso della memoria in MATLAB e C++ è simile, con piccole variazioni. Tuttavia, MATLAB tende a utilizzare una quantità leggermente maggiore di memoria per la decomposizione rispetto a C++, probabilmente a causa della gestione interna delle strutture dati e dell'overhead associato all'ambiente di esecuzione.

3.6.2. Tempi

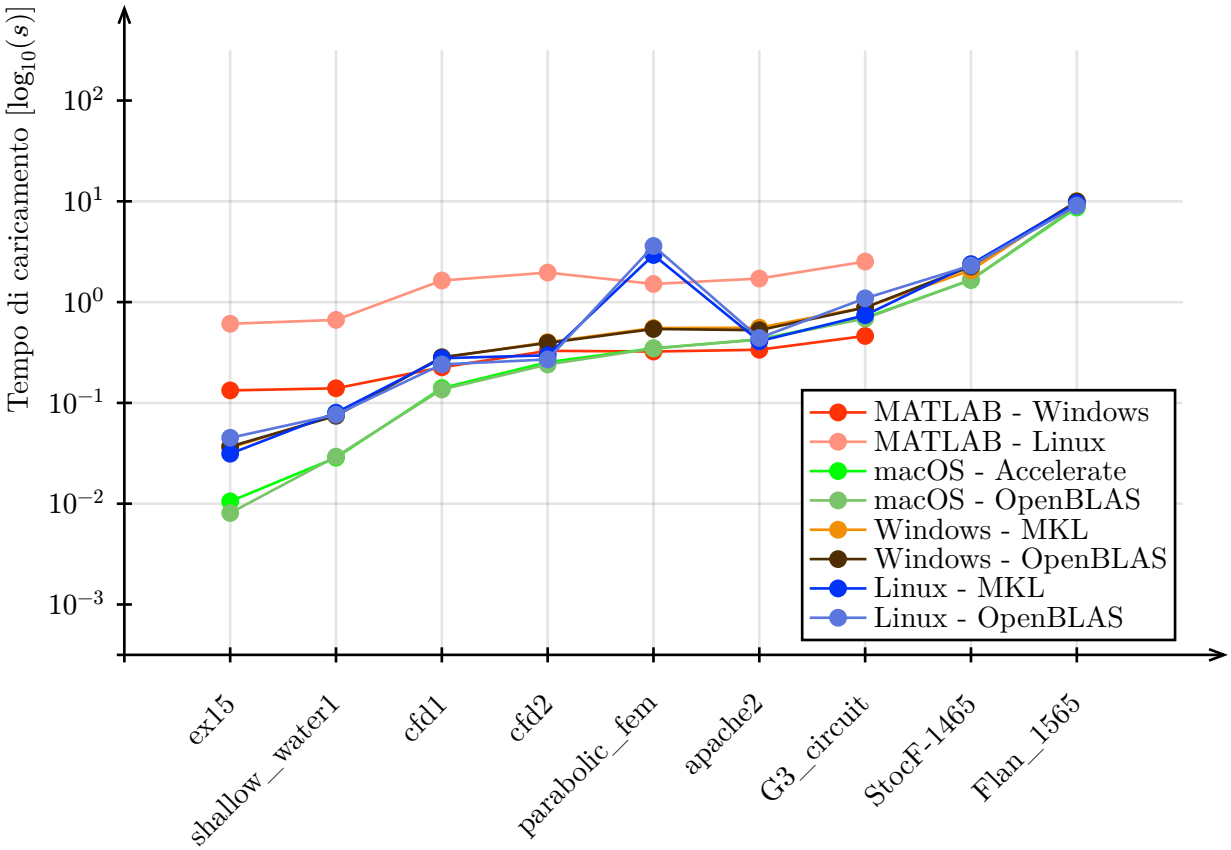

Figura 12: Confronto tempo caricamento tra MATLAB e C++

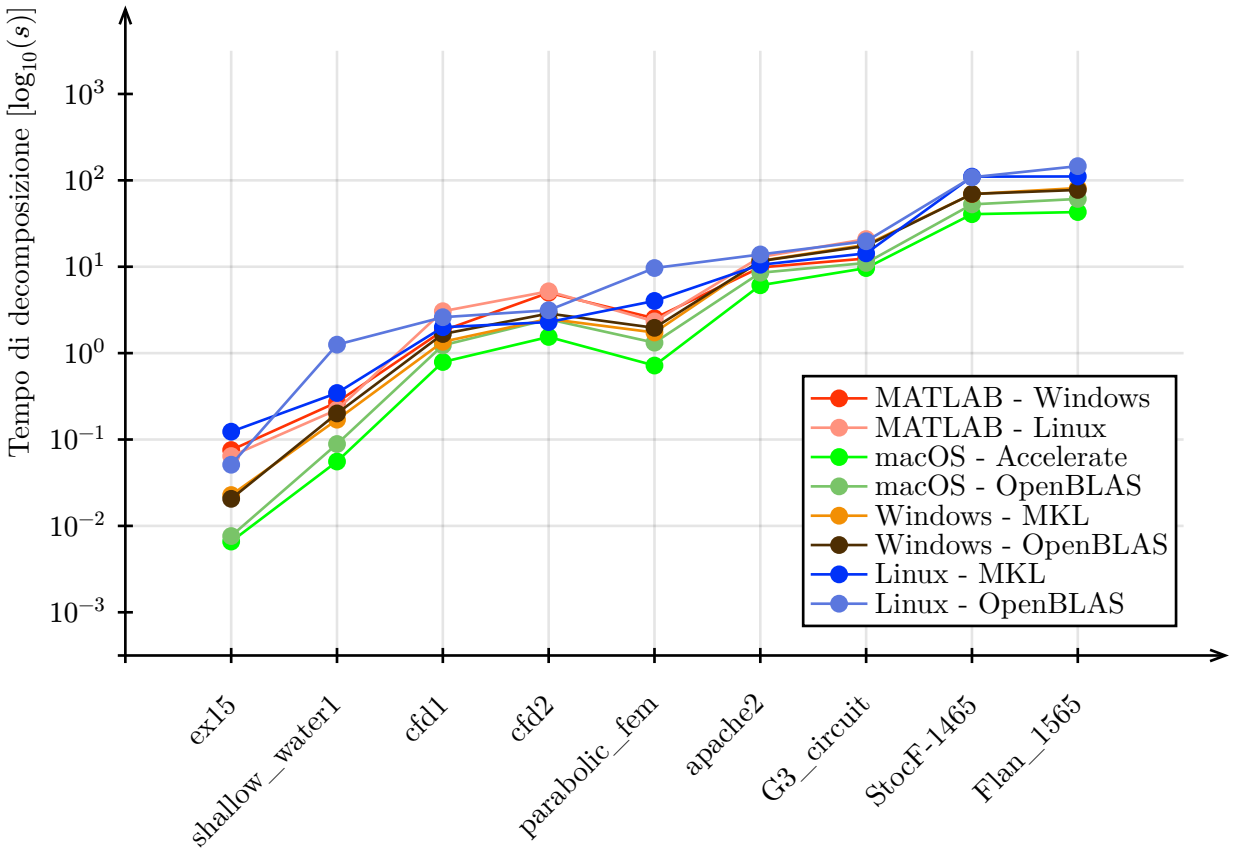

Figura 13: Confronto tempo decomposizione tra MATLAB e C++

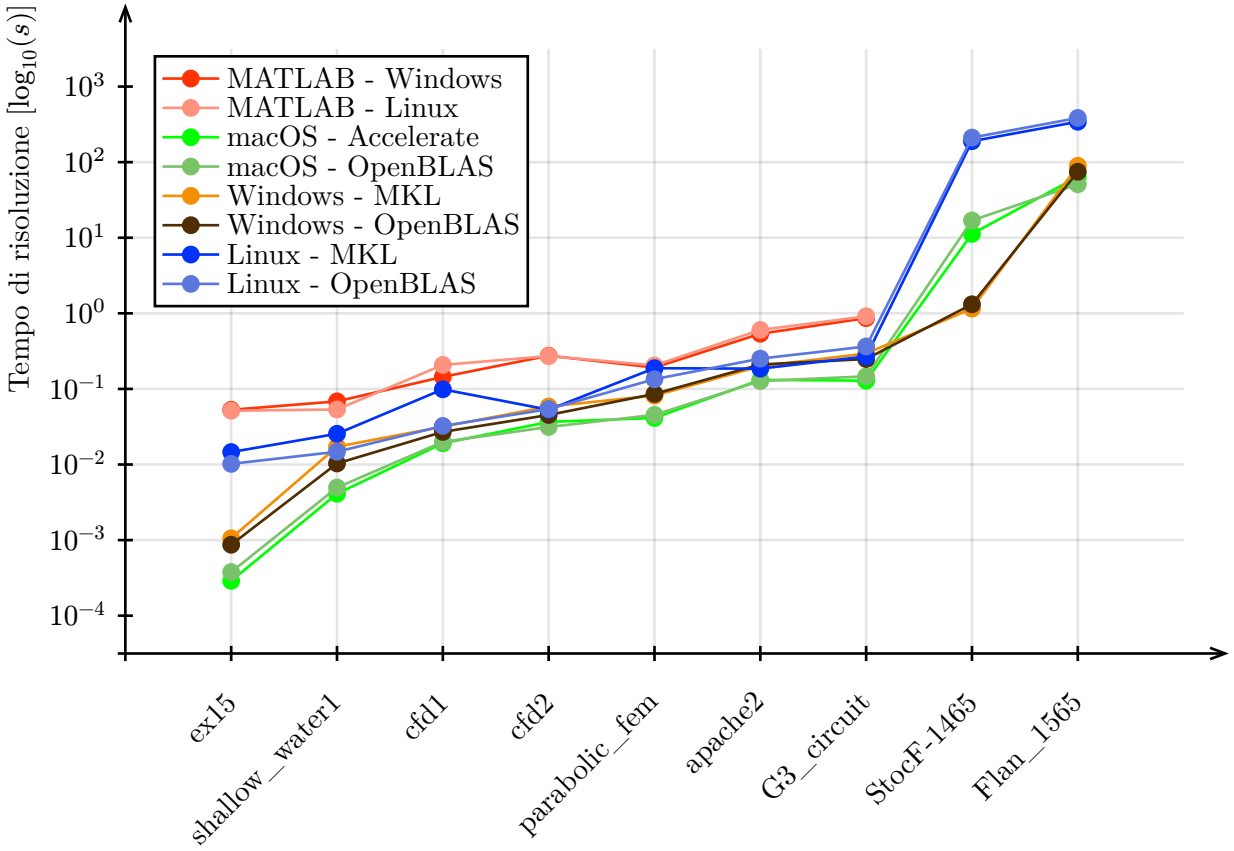

Figura 14: Confronto tempo risoluzione tra MATLAB e C++

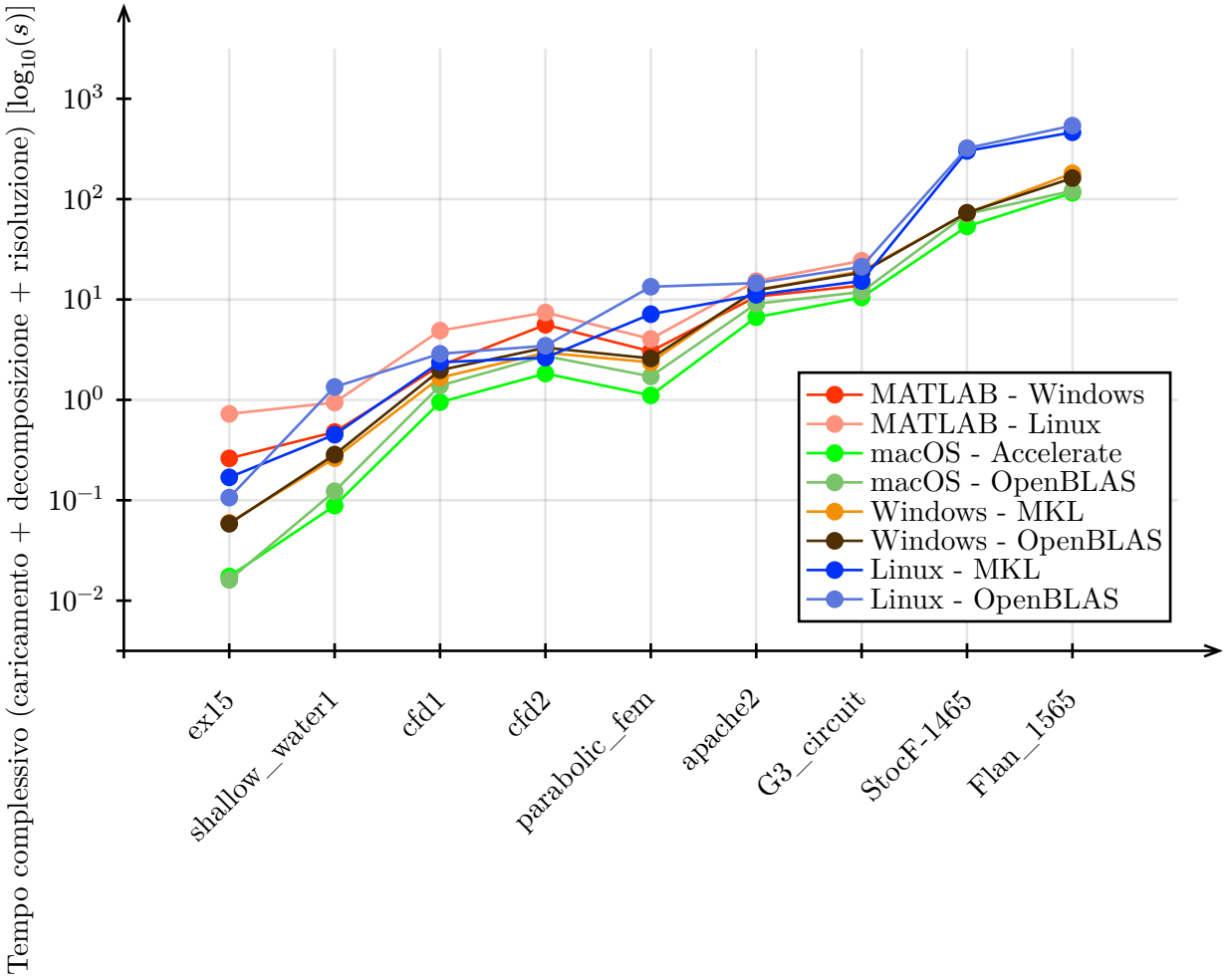

Figura 15: Confronto tempo complessivo tra MATLAB e C++

L'analisi del tempo di esecuzione basandoci su Figura 12, Figura 13, Figura 14 e Figura 15 evidenzia alcuni aspetti chiave:

1. **Tempo di caricamento:** Le matrici più piccole, come *ex15* e *shallow_water1*, hanno tempi di caricamento molto ridotti, inferiori al millisecondo. Man mano che la dimensione cresce, il tempo aumenta significativamente. Per le matrici più grandi, come *Flan_1565*, il caricamento può richiedere diversi secondi. Non è presente una differenza significativa tra C++ e MATLAB tranner per MATLAB - Linux, dove il caricamento è più lento.
2. **Tempo di decomposizione:** Questa fase è la più dispendiosa in termini di tempo. Per matrici grandi come *Flan_1565*, la decomposizione richiede oltre 100 secondi, evidenziando la complessità del processo. Però si può notare che in generale tra i diversi sistemi operativi e le librerie BLAS non ci sono differenze significative.
3. **Tempo di risoluzione:** Diversamente dalla decomposizione, la fase di risoluzione è generalmente molto più veloce. Questo avviene perché la risoluzione sfrutta la struttura fattorizzata della matrice, riducendo il numero di operazioni necessarie. Per la maggior parte delle matrici, il tempo di risoluzione è inferiore a 1 secondo, a riprova dell'efficacia

dei metodi numerici impiegati, e anche qua non si nota una grossa differenza tra MATLAB e C++.

4. **Tempo complessivo:** Sommando le tre fasi, emerge chiaramente che la decomposizione è il passaggio dominante in termini di costo computazionale. Ottimizzare questo processo tramite migliori algoritmi o librerie specializzate potrebbe portare a una riduzione significativa dei tempi di esecuzione, specialmente per matrici di grandi dimensioni.
5. **Confronto MATLAB e C++:** In generale, i tempi di esecuzione tra MATLAB e C++ sono comparabili, con piccole variazioni. Tuttavia, MATLAB tende a essere leggermente più lento, soprattutto nella fase di caricamento e decomposizione. Questo potrebbe essere dovuto all'overhead dell'ambiente MATLAB e alla sua gestione delle strutture dati.

3.6.3. Errore Relativo

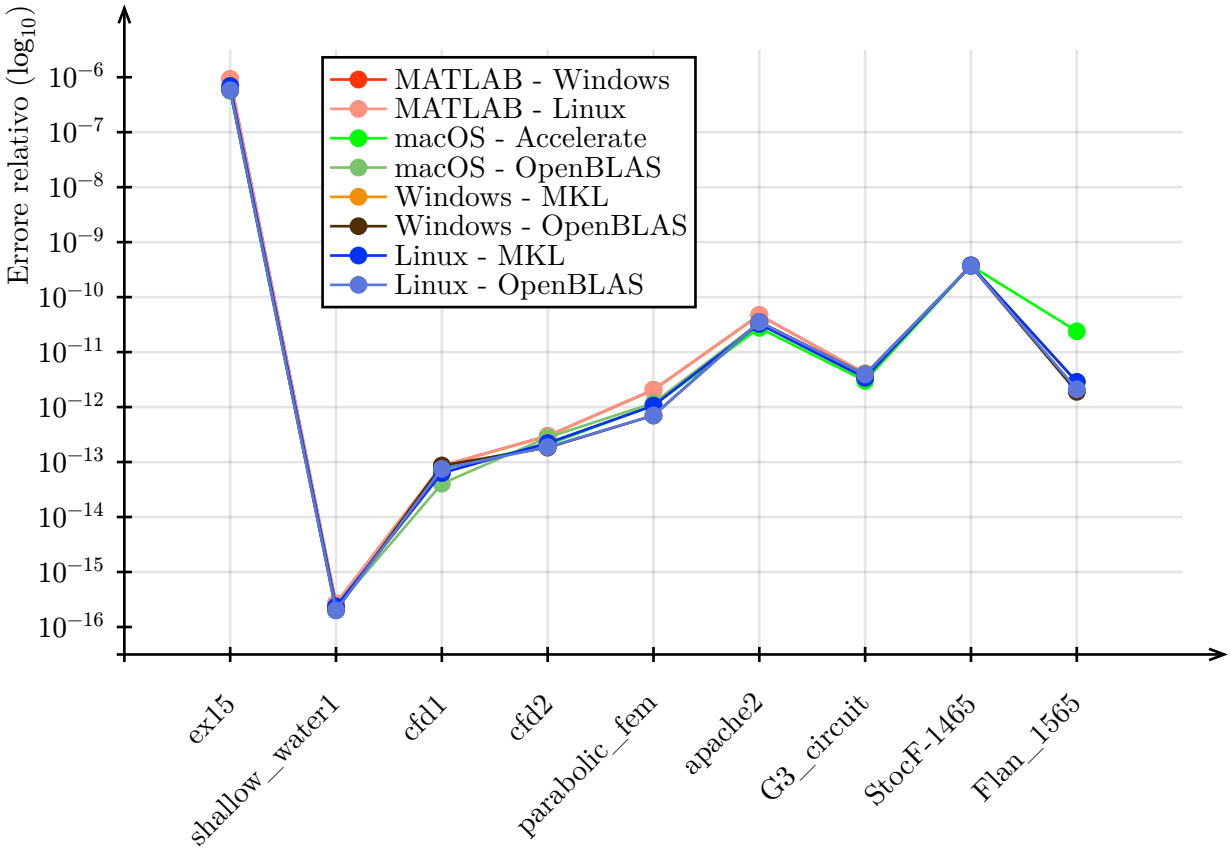

Figura 16: Confronto errore relativo tra MATLAB e C++

L'analisi dell'errore relativo riportata in Figura 16 mostra come la precisione numerica vari a seconda delle librerie e dell'ambiente di esecuzione:

1. **Ordine di grandezza dell'errore:** In generale, l'errore relativo oscilla tra $10^{\{-6\}}$ e $10^{\{-16\}}$ ma apparte per macOS - Accelerate nella matrice «Flan_1565» che ha un errore un pò piu altro rispetto alle altre implementazioni non sembrano esserci differenze significative tra i vari sistemi operativi e le librerie BLAS.

2. **Variazione tra le matrici:** Notiamo come non è la dimensione della matrice a influenzare l'errore, ma piuttosto la struttura e le proprietà intrinseche della matrice stessa. Ad esempio, matrici con una maggiore densità di zeri o con una struttura particolare possono portare a errori più elevati. E nel nostro caso notiamo come è la matriche «ex15» che è la piu piccola a generare un errore piu alto rispetto alle altre.

3.7. Considerazioni Finali sul Confronto

In conclusione, notiamo come ci aspettavamo dato l'utilizzo della stessa libreria CHOLMOD, che le prestazioni di MATLAB e C++ sono comparabili, con piccole variazioni dovute all'overhead di MATLAB. Tuttavia, C++ mostra una maggiore flessibilità e potenziale per ottimizzazioni future, grazie alla possibilità di utilizzare diverse librerie BLAS e di personalizzare l'implementazione. Però bisogna anche considera che MATLAB non è riuscito a completare le due matrici più grandi, il che potrebbe indicare limitazioni nella versione di CHOLMOD utilizzata o nella gestione della memoria. Questo rappresenta un punto a favore di C++, che ha gestito con successo tutte le matrici testate.

Però è anche interessante notare che non ci siano grandi differenze rispetto all'utilizzo della libreria Intel MKL o OpenBLAS, il che suggerisce che CHOLMOD è ben ottimizzata per lavorare con entrambe le librerie BLAS. E che le librerie BLAS siano ben ottimizzate e che la scelta con full open source di OpenBLAS non abbia impatti significativi sulle prestazioni rispetto a Intel MKL, che è una libreria proprietaria.

Inoltre l'analisi dell'uso della memoria fa capire quanto sia importante avere abbastanza memoria disponibile per gestire matrici di grandi dimensioni, specialmente durante la fase di decomposizione. Questo è un aspetto cruciale da considerare quando si lavora con applicazioni che richiedono l'elaborazione di grandi dataset o matrici sparse. Possibile grazie all'uso della memoria virtuale e della gestione dinamica della memoria, che permette di allocare risorse in modo efficiente e ridurre il rischio di esaurimento della memoria fisica.

4. Conclusioni

L'analisi comparativa tra MATLAB e una soluzione open-source basata su C++ ha fornito risultati significativi riguardo alla fattorizzazione di Cholesky applicata a matrici sparse simmetriche definite positive. MATLAB, nonostante l'ampia documentazione e l'integrazione con CHOLMOD, mostra alcune limitazioni dovute all'uso di versioni obsolete delle librerie e alla chiusura del proprio ecosistema. Questo ha portato a difficoltà nel trattamento di matrici di grandi dimensioni, dimostrando che aggiornamenti più frequenti delle librerie interne potrebbero migliorare le prestazioni e la stabilità del software.

D'altra parte, la nostra implementazione in C++ ha evidenziato vantaggi concreti grazie all'uso diretto di Eigen e SuiteSparse, consentendo un maggiore controllo sull'ottimizzazione e sull'integrazione con diverse librerie BLAS e LAPACK. L'adozione di Intel MKL, OpenBLAS e Apple Accelerate ha dimostrato come la scelta delle librerie di algebra lineare influenzi direttamente il tempo di decomposizione e la gestione della memoria, senza grosse differenze di prestazioni tra l'utilizzo di BLAS diversi su sistemi operativi differenti.

Un aspetto critico emerso è che MATLAB non è stato in grado di completare la fattorizzazione sulle due matrici più grandi, mentre la soluzione C++ ha gestito l'intero set di dati. Questo

evidenzia l'importanza di mantenere le librerie costantemente aggiornate per garantire la robustezza dell'algoritmo. Inoltre, la flessibilità offerta dall'approccio in C++ ha permesso di verificare l'impatto del numero di thread sulla fattorizzazione, rivelando che l'allocazione dinamica dei thread su macOS ha prodotto tempi di decomposizione leggermente migliori rispetto a Windows e Linux.

Dal punto di vista computazionale, la nostra analisi conferma che le soluzioni completamente open-source possono competere con MATLAB, offrendo prestazioni comparabili e, in alcuni casi, superiori. Tuttavia, MATLAB rimane una scelta vantaggiosa per la prototipazione rapida e la semplicità d'uso, risultando più accessibile per utenti meno esperti in programmazione.

Inoltre, abbiamo osservato che, nonostante MATLAB sia un software proprietario, utilizza librerie open-source come CHOLMOD, parte di SuiteSparse. Questo solleva interrogativi sull'effettiva necessità di un software proprietario, quando si possono adottare direttamente librerie open-source che offrono la stessa funzionalità con maggiore flessibilità e senza i vincoli di un ecosistema chiuso.

Si può quindi concludere che, se l'obiettivo è risolvere sistemi lineari sparsi in modo efficiente, la scelta del sistema operativo non è determinante quanto quella delle librerie BLAS e LAPACK adottate e l'architettura del sistema. Le prestazioni dipendono principalmente dalla quantità di RAM disponibile e dalla capacità del processore di gestire operazioni parallele, più che dal sistema operativo stesso. Inoltre, qualora fosse necessario utilizzare la memoria virtuale, è consigliabile disporre di un SSD veloce per evitare rallentamenti significativi, poiché l'uso di HDD tradizionali può compromettere le prestazioni durante operazioni di calcolo intensive, dato l'uso intensivo della memoria durante la decomposizione.

In definitiva, la scelta tra software proprietario e open-source dipende dalle esigenze specifiche: MATLAB garantisce un ambiente integrato e una facilità d'uso indiscutibile, mentre C++ con Eigen e CHOLMOD offre maggiore flessibilità e scalabilità su hardware moderno. Per applicazioni che richiedono prestazioni elevate su matrici di grandi dimensioni, un'implementazione open-source si dimostra una valida alternativa, consentendo ottimizzazioni avanzate e un controllo diretto sull'allocazione della memoria e sui metodi di decomposizione utilizzati.

5. Appendice Codici

5.1. C++ e CMake

File: main.cpp

C++

```cpp
#include <iostream>
#include <fstream>
#include <chrono>
#include <thread>
#include <fast_matrix_market/app/Eigen.hpp>
#include <Eigen/SparseCore>
#include <filesystem>
#include <format>
```

```
9   #include <Eigen/CholmodSupport>
10  #ifdef EIGEN_USE_MKL_ALL
11  #include <mkl.h>
12  #endif
13  #ifdef OPEN_BLAS
14  #include <cblas.h>
15  #endif
16  #ifdef USING_ACCEL
17  #include <Accelerate/Accelerate.h>
18  #endif
19
20  typedef Eigen::SparseMatrix<double, Eigen::ColMajor, int64_t> SparseMatrix;
21
22  #ifdef EIGEN_USE_MKL_ALL
23  constexpr auto BLAS = "MKL";
24  #elif defined(OPEN_BLAS)
25  constexpr auto BLAS = "OpenBLAS";
26  #elif defined(USING_ACCEL)
27  constexpr auto BLAS = "Accelerate";
28  #else
29  constexpr auto BLAS = "Unknown";
30  #endif
31
32  constexpr auto HEADER_CSV =
    "os,blas,numThreads,timestamp,matrixName,rows,cols,nonZeros,loadTime,loadMem,
33
    decompTime,decompMem,decompPeakMem,solveTime,solveMem,solvePeakMem,relativeError";
34  constexpr auto OUT_FILE = "bench.csv";
35
36  #ifndef NO_SLEEP
37  constexpr std::chrono::seconds SLEEP_TIME{5};
38  #endif
39
40  const std::string getOSName();
41
42  const int getNumThreads();
43
44  int solveMatrixMarket(const std::filesystem::path& path);
45
46  int main(int argc, char* argv[], char** envp) {
```

```
47
48  #ifdef EIGEN_VECTORIZE
49    std::cerr << "Eigen CPU vectorization is enabled." << std::endl;
50  #endif
51    std::cerr << BLAS << " Num of threads: " << getNumThreads() << std::endl;
52
53    const std::filesystem::path matrices_dir = (argc > 1) ? argv[1] : "matrices";
54
55    // Check if directory exists before iterating
56    if (!std::filesystem::exists(matrices_dir) || !
      std::filesystem::is_directory(matrices_dir)) {
57        std::cerr << "Error: Directory '" << matrices_dir.string() << "' does not
          exist or is not a directory." << std::endl;
58        return 1;
59    }
60
61    // Loop through all files in the directory
62    for (const auto& entry : std::filesystem::directory_iterator(matrices_dir)) {
63      if (entry.is_regular_file() && entry.path().extension() == ".mtx") {
64        const auto matrixName { entry.path().stem().string() };
65
66        // Sleep for 10 seconds before processing the next file
67        #ifndef NO_SLEEP
68        std::cerr << std::format("Sleeping for {} seconds...", SLEEP_TIME) <<
          std::endl;
69        std::this_thread::sleep_for(SLEEP_TIME);
70        #endif
71
72        int result = solveMatrixMarket(entry.path());
73        if (result == 0) {
74          std::cerr << "Processed " << matrixName << std::endl;
75        } else {
76          std::cerr << "Error " << matrixName << std::endl;
77        }
78      }
79    }
80
81    return 0;
82  }
83
84  const std::string getOSName() {
```

```
85      #if defined(_WIN32)
86          return "Windows";
87      #elif defined(__APPLE__)
88          return "macOS";
89      #elif defined(__linux__)
90          return "Linux";
91      #elif defined(__unix__)
92          return "Unix";
93      #else
94          return "Unknown";
95      #endif
96  }
97
98  const int getNumThreads() {
99    #ifdef EIGEN_USE_MKL_ALL
100     return mkl_get_max_threads();
101   #elif defined(OPEN_BLAS)
102     return openblas_get_num_threads();
103   #elif defined(USING_ACCEL)
104     return BLASGetThreading() == BLAS_THREADING_SINGLE_THREADED ? 1 : -1;
105   #else
106     return -1;
107   #endif
108 }
109
110 int solveMatrixMarket(const std::filesystem::path& path) {
111   const auto timestamp = std::chrono::system_clock::now();
112
113   std::cerr << std::format("{} - Processing {}...", timestamp,
      path.stem().string()) << std::endl;
114
115   std::ifstream matrix_file(path, std::ios::in);
116   std::ofstream csv_file(OUT_FILE, std::ios::ate | std::ios::app);
117
118   if (std::filesystem::is_empty(OUT_FILE)) {
119     std::cerr << std::format("Writing headers to {}...", OUT_FILE) << std::endl;
120     csv_file << HEADER_CSV << std::endl;
121     csv_file.flush();
122   }
123
```

```
124     SparseMatrix A;
125
126     std::cerr << std::format("Reading matrix...") << std::endl;
127
128     auto start{ std::chrono::high_resolution_clock::now() };
129     fast_matrix_market::read_matrix_market_eigen(matrix_file, A);
130     auto end{ std::chrono::high_resolution_clock::now() };
131     matrix_file.close();
132
133     csv_file << std::format("{},{},{},{},{},{},{},{},", getOSName(), BLAS,
        getNumThreads(), timestamp, path.stem().string(), A.rows(), A.cols(),
        A.nonZeros());
134
135     const size_t valuesSize = A.nonZeros() * sizeof(double);
136
137     // Size of inner indices array (nonzeros * sizeof(index type))
138     const size_t innerIndicesSize = A.nonZeros() * sizeof(SuiteSparse_long);
139
140     // Size of outer indices array ((outerSize+1) * sizeof(index type))
141     const size_t outerIndicesSize = (A.outerSize() + 1) * sizeof(SuiteSparse_long);
142
143     const auto loadTime = std::chrono::duration_cast<std::chrono::duration<long
        double, std::milli>>(end - start).count();
144     const auto loadMem = valuesSize + innerIndicesSize + outerIndicesSize;
145
146     std::cerr << std::format("Matrix read took {} ms and {} bytes", loadTime,
        loadMem) << std::endl;
147     csv_file << std::format("{},{},", loadTime, loadMem);
148
149     Eigen::CholmodDecomposition<SparseMatrix> solver;
150
151     // Make sure the matrix is in compressed column form
152     A.makeCompressed();
153
154     solver.cholmod().memory_allocated = 0;
155     solver.cholmod().memory_inuse = 0;
156     solver.cholmod().memory_usage = 0;
157     std::cerr << std::format("Decomposing matrix...") << std::endl;
158     start = std::chrono::high_resolution_clock::now();
159     solver.compute(A);
160     end = std::chrono::high_resolution_clock::now();
```

```
161
162   if (solver.info() != Eigen::Success) {
163     std::cerr << "Decomposition failed." << std::endl;
164     csv_file << std::format("{},{},{},{}", "N/A", "N/A", "N/A", "Decomposition
      failed.") << std::endl;
165     csv_file.close();
166     return -1;
167   }
168
169   const auto decompTime = std::chrono::duration_cast<std::chrono::duration<long
      double, std::milli>>(end - start).count();
170   const auto decompMem = solver.cholmod().memory_allocated;
171   const auto decompPeakMem = solver.cholmod().memory_usage;
172
173   std::cerr << std::format("Decomposition succeeded with in {} ms and {} bytes",
      decompTime, decompMem) << std::endl;
174   csv_file << std::format("{},{},{},", decompTime, decompMem, decompPeakMem);
175
176   Eigen::VectorXd b(A.rows()), x(A.rows()), xe(A.rows());
177   x.setOnes();
178   b = A * x;
179
180   solver.cholmod().memory_allocated = 0;
181   solver.cholmod().memory_inuse = 0;
182   solver.cholmod().memory_usage = 0;
183   std::cerr << std::format("Solving matrix...") << std::endl;
184   start = std::chrono::high_resolution_clock::now();
185   xe = solver.solve(b);
186   end = std::chrono::high_resolution_clock::now();
187
188   const auto solveTime = std::chrono::duration_cast<std::chrono::duration<long
      double, std::milli>>(end - start).count();
189   const auto solveMem = solver.cholmod().memory_allocated;
190   const auto solvePeakMem = solver.cholmod().memory_usage;
191
192   std::cerr << std::format("Solve took {} ms and {} bytes", solveTime, solveMem)
      << std::endl;
193   csv_file << std::format("{},{},{},", solveTime, solveMem, solvePeakMem);
194
195   if (solver.info() != Eigen::Success) {
196     std::cerr << "Solving failed." << std::endl;
```

```
197     csv_file << std::format("{}", "Solving failed.") << std::endl;
198     csv_file.close();
199     return -1;
200   }
201
202   // Valutazione dell'errore
203   auto err = sqrt((x - xe).squaredNorm() / x.squaredNorm());
204   csv_file << std::format("{}", err) << std::endl;
205
206   std::cerr << std::format("Error: {}", err) << std::endl;
207
208   csv_file.close();
209
210   return 0;
211 }
```

Codice 1: main

File: CMakeLists.txt

```
1   cmake_minimum_required(VERSION 3.28)
2
3   # OneAPI environment check
4   if(NOT APPLE)
5    if(WIN32)
6       set(ONEAPI_SETVARS_PATH "call \"C:/Program Files (x86)/Intel/oneAPI/
        setvars.bat\"")
7     elseif(UNIX)
8      set(ONEAPI_SETVARS_PATH "source /opt/intel/oneapi/setvars.sh")
9    endif()
10
11   if(WIN32)
12      find_program(CMAKE_Fortran_COMPILER NAMES ifx.exe PATHS $ENV{CMPLR_ROOT}/bin)
13    endif()
14    set(MKL_LINK static)
15   find_package(MKL REQUIRED PATHS $ENV{MKLROOT})
16  endif()
17
18  project(MCS-Project LANGUAGES C CXX Fortran)
19
20  message(STATUS "CMAKE_SYSTEM_NAME: ${CMAKE_SYSTEM_NAME}")
21
22  # Define directories
```

```
23  set(SRC_DIR "${CMAKE_SOURCE_DIR}/src")
24  set(LIB_DIR "${CMAKE_SOURCE_DIR}/lib")
25  set(BIN_DIR "${CMAKE_SOURCE_DIR}/target")
26
27  # Create output directories (these will be created during the build)
28  file(MAKE_DIRECTORY ${BIN_DIR})
29
30  message(STATUS "Source Directory: ${SRC_DIR}")
31  message(STATUS "Library Directory: ${LIB_DIR}")
32  message(STATUS "Binary Directory: ${BIN_DIR}")
33
34  # SuiteSparse configuration
35  set(SUITESPARSE_DIR "${LIB_DIR}/suitesparse")
36
37  # Proper clean targets
38  add_custom_target(clean-src
39    COMMAND ${CMAKE_COMMAND} -E remove_directory ${BASE_OBJ_DIR}
40   COMMAND ${CMAKE_COMMAND} -E remove_directory ${BIN_DIR}
41    COMMENT "Cleaning object files and main executable"
42  )
43
44  add_custom_target(clean-lib
45   COMMAND ${CMAKE_COMMAND} -E remove_directory ${BASE_OBJ_DIR}/suitesparse_main-
     prefix
46   COMMAND ${CMAKE_COMMAND} -E remove_directory ${BASE_OBJ_DIR}/
     suitesparse_openblas-prefix
47   COMMAND ${CMAKE_COMMAND} -E remove_directory ${BASE_OBJ_DIR}/suitesparse_main
48    COMMAND ${CMAKE_COMMAND} -E remove_directory ${BASE_OBJ_DIR}/
     suitesparse_openblas
49    COMMENT "Cleaning SuiteSparse libraries"
50  )
51
52  add_custom_target(clean-all
53    DEPENDS clean-src clean-lib
54    COMMENT "Cleaning everything (sources and libraries)"
55  )
56
57  # Configure library paths based on platform
58  set(SUITESPARSE_LIB_NAMES
59   "suitesparseconfig"
60    "amd"
```

```
61      "camd"
62    "colamd"
63    "ccolamd"
64      "cholmod"
65  )
66
67  # Copy override files before building
68  add_custom_target(copy_override_files ALL
69    COMMAND ${CMAKE_COMMAND} -E copy_if_different "${LIB_DIR}/override/cholmod.h.in"
70    "${SUITESPARSE_DIR}/CHOLMOD/Config/cholmod.h.in"
71    COMMAND ${CMAKE_COMMAND} -E copy_if_different "${LIB_DIR}/override/
      t_cholmod_malloc.c"
72    "${SUITESPARSE_DIR}/CHOLMOD/Utility/t_cholmod_malloc.c"
73     COMMAND ${CMAKE_COMMAND} -E copy_if_different "${LIB_DIR}/override/
      t_cholmod_realloc.c"
74     "${SUITESPARSE_DIR}/CHOLMOD/Utility/t_cholmod_realloc.c"
75    COMMENT "Copying override files to SuiteSparse"
76  )
77
78  set(SUITESPARSE_MAIN_INSTALL_DIR ${CMAKE_SOURCE_DIR}/cmake/suitesparse_main)
79  set(SUITESPARSE_OPENBLAS_INSTALL_DIR ${CMAKE_SOURCE_DIR}/cmake/
    suitesparse_openblas)
80
81  make_directory(${SUITESPARSE_MAIN_INSTALL_DIR})
82  make_directory(${SUITESPARSE_OPENBLAS_INSTALL_DIR})
83
84  set(OPENBLAS_DIR ${CMAKE_SOURCE_DIR}/cmake/openblas)
85
86  include(ExternalProject)
87
88  # Set the common CMake options for SuiteSparse
89  list(APPEND SUITESPARSE_CMAKE_OPTIONS
90    -DSUITESPARSE_USE_64BIT_BLAS=ON
91    -DSUITESPARSE_LOCAL_INSTALL=1
92    -DBUILD_SHARED_LIBS=OFF
93    -DBUILD_STATIC_LIBS=ON
94   -DSUITESPARSE_ENABLE_PROJECTS=cholmod
95  )
96
97  if (WIN32)
```

```
98    list(APPEND SUITESPARSE_CMAKE_OPTIONS -
      DCMAKE_Fortran_COMPILER=${CMAKE_Fortran_COMPILER})
99  endif()
100
101 function(suitesparse_main)
102   list(APPEND SUITESPARSE_CMAKE_OPTIONS
103     -DCMAKE_INSTALL_PREFIX=${SUITESPARSE_MAIN_INSTALL_DIR}
104  )
105
106  if(APPLE)
107     list(APPEND SUITESPARSE_CMAKE_OPTIONS -DBLA_VENDOR=Apple -
      DSUITESPARSE_USE_64BIT_BLAS=OFF)
108    execute_process(
109      COMMAND brew --prefix libomp
110      OUTPUT_VARIABLE LIBOMP_PREFIX
111       OUTPUT_STRIP_TRAILING_WHITESPACE
112    )
113     message (STATUS "OpenMP prefix location " ${LIBOMP_PREFIX})
114     list(APPEND SUITESPARSE_CMAKE_OPTIONS
115       -DBLA_VENDOR=Apple
116      -DSUITESPARSE_USE_64BIT_BLAS=OFF
117      -DOpenMP_FOUND=TRUE
118       -DOpenMP_C_FOUND=C
119      -DOpenMP_C_FLAGS="-lomp"
120      -DOpenMP_C_INCLUDE_DIRS=${LIBOMP_PREFIX}/include
121      -DOpenMP_C_LIBRARIES=${LIBOMP_PREFIX}/lib
122       -DOpenMP_C_LIB_NAMES=libomp
123       -DOpenMP_libomp_LIBRARY=${LIBOMP_PREFIX}/lib/libomp.a
124     )
125   else()
126    list(APPEND SUITESPARSE_CMAKE_OPTIONS -DBLA_VENDOR=Intel10_64ilp)
127   endif()
128
129  if(LINUX)
130     get_target_property(MKL_INTEL_ILP64_PATH MKL::mkl_intel_ilp64
      IMPORTED_LOCATION)
131    get_target_property(MKL_INTEL_THREAD_PATH MKL::mkl_intel_thread
      IMPORTED_LOCATION)
132    get_target_property(MKL_CORE_PATH MKL::mkl_core IMPORTED_LOCATION)
133
```

```
134       set(MKL_LIBS "-m64 -Wl,--start-group ${MKL_INTEL_ILP64_PATH}
          ${MKL_INTEL_THREAD_PATH} ${MKL_CORE_PATH} -Wl,--end-group -liomp5 -lpthread -
          lm -ldl")
135     endif()
136
137   message(STATUS "SUITESPARSE_CMAKE_OPTIONS: ${SUITESPARSE_CMAKE_OPTIONS}")
138
139   if(LINUX)
140       ExternalProject_Add(
141         suitesparse_main
142         SOURCE_DIR "${SUITESPARSE_DIR}"
143          CONFIGURE_COMMAND ${CMAKE_COMMAND} -S "${SUITESPARSE_DIR}" -B
          "${CMAKE_BINARY_DIR}/suitesparse_main" ${SUITESPARSE_CMAKE_OPTIONS}
          ${SUITESPARSE_CMAKE_OPTIONS} -DBLAS_LIBRARIES=${MKL_LIBS} -
          DLAPACK_LIBRARIES=${MKL_LIBS}
144         BUILD_COMMAND ${CMAKE_COMMAND} --build "${CMAKE_BINARY_DIR}/suitesparse_main"
          --config Release
145         INSTALL_COMMAND ${CMAKE_COMMAND} --install ${CMAKE_BINARY_DIR}/
          suitesparse_main
146       )
147     else()
148       ExternalProject_Add(
149         suitesparse_main
150         SOURCE_DIR "${SUITESPARSE_DIR}"
151          CONFIGURE_COMMAND ${CMAKE_COMMAND} -S "${SUITESPARSE_DIR}" -B
          "${CMAKE_BINARY_DIR}/suitesparse_main" ${SUITESPARSE_CMAKE_OPTIONS}
152          BUILD_COMMAND ${CMAKE_COMMAND} --build "${CMAKE_BINARY_DIR}/
          suitesparse_main" --config Release
153         INSTALL_COMMAND ${CMAKE_COMMAND} --install ${CMAKE_BINARY_DIR}/
          suitesparse_main
154       )
155     endif()
156 endfunction()
157
158 suitesparse_main()
159
160 function(suitesparse_openblas)
161     list(APPEND SUITESPARSE_CMAKE_OPTIONS
162       -DCMAKE_INSTALL_PREFIX=${SUITESPARSE_OPENBLAS_INSTALL_DIR}
163       -DBLA_VENDOR=OpenBLAS
164     )
165
```

```
166  if(WIN32)
167     list(APPEND SUITESPARSE_CMAKE_OPTIONS
168       -DBLAS_LIBRARIES=${OPENBLAS_DIR}/lib/libopenblas.lib
169      -DLAPACK_LIBRARIES=${OPENBLAS_DIR}/lib/libopenblas.lib
170    )
171   elseif(APPLE)
172     execute_process(
173      COMMAND brew --prefix openblas
174      OUTPUT_VARIABLE OPENBLAS_DIR
175      OUTPUT_STRIP_TRAILING_WHITESPACE
176    )
177     list(APPEND SUITESPARSE_CMAKE_OPTIONS
178       -DSUITESPARSE_USE_64BIT_BLAS=OFF
179      -DBLAS_LIBRARIES=${OPENBLAS_DIR}/lib/libopenblas.a
180      -DLAPACK_LIBRARIES=${OPENBLAS_DIR}/lib/libopenblas.a
181    )
182     execute_process(
183      COMMAND brew --prefix libomp
184      OUTPUT_VARIABLE LIBOMP_PREFIX
185       OUTPUT_STRIP_TRAILING_WHITESPACE
186    )
187     message (STATUS "OpenMP prefix location " ${LIBOMP_PREFIX})
188     list(APPEND SUITESPARSE_CMAKE_OPTIONS
189       -DBLA_VENDOR=Apple
190      -DSUITESPARSE_USE_64BIT_BLAS=OFF
191      -DOpenMP_FOUND=TRUE
192       -DOpenMP_C_FOUND=C
193      -DOpenMP_C_FLAGS="-lomp"
194      -DOpenMP_C_INCLUDE_DIRS=${LIBOMP_PREFIX}/include
195      -DOpenMP_C_LIBRARIES=${LIBOMP_PREFIX}/lib
196       -DOpenMP_C_LIB_NAMES=libomp
197       -DOpenMP_libomp_LIBRARY=${LIBOMP_PREFIX}/lib/libomp.a
198     )
199   endif()
200
201  message(STATUS "SUITESPARSE_CMAKE_OPTIONS: ${SUITESPARSE_CMAKE_OPTIONS}")
202
203  ExternalProject_Add(
204    suitesparse_openblas
205    SOURCE_DIR "${SUITESPARSE_DIR}"
```

```
206       CONFIGURE_COMMAND ${CMAKE_COMMAND} -S "${SUITESPARSE_DIR}" -B
          "${CMAKE_BINARY_DIR}/suitesparse_openblas" ${SUITESPARSE_CMAKE_OPTIONS}
207       BUILD_COMMAND ${CMAKE_COMMAND} --build "${CMAKE_BINARY_DIR}/
          suitesparse_openblas" --config Release
208      INSTALL_COMMAND ${CMAKE_COMMAND} --install ${CMAKE_BINARY_DIR}/
         suitesparse_openblas
209    )
210  endfunction()
211
212  suitesparse_openblas()
213
214  # Find all source files in the src directory
215  file(GLOB_RECURSE SRC_FILES "${SRC_DIR}/*.cpp")
216
217  if(WIN32)
218   set(EXE_SUFFIX "windows")
219    set(EXE_BLAS "mkl")
220  elseif(LINUX)
221    set(EXE_SUFFIX "linux")
222    set(EXE_BLAS "mkl")
223  elseif(APPLE)
224    set(EXE_SUFFIX "macos")
225    set(EXE_BLAS "accelerate")
226  else()
227    set(EXE_SUFFIX "unknown")
228  endif()
229
230  # Create the main executable target that uses MKL
231  add_executable(main ${SRC_FILES})
232  set_target_properties(main PROPERTIES
233   OUTPUT_NAME "main_${EXE_BLAS}_${EXE_SUFFIX}"
234   OUTPUT_NAME_RELEASE "main_${EXE_BLAS}_${EXE_SUFFIX}"
235   RUNTIME_OUTPUT_DIRECTORY "${BIN_DIR}"
236    RUNTIME_OUTPUT_DIRECTORY_RELEASE "${BIN_DIR}"
237  )
238
239  # Create the main executable target that uses OpenBLAS
240  add_executable(main_openblas ${SRC_FILES})
241  set_target_properties(main_openblas PROPERTIES
242    OUTPUT_NAME "main_openblas_${EXE_SUFFIX}"
243    OUTPUT_NAME_RELEASE "main_openblas_${EXE_SUFFIX}"
```

```
244   RUNTIME_OUTPUT_DIRECTORY "${BIN_DIR}"
245   RUNTIME_OUTPUT_DIRECTORY_RELEASE "${BIN_DIR}"
246 )
247
248 # Set C++ standard to C++20
249 target_compile_features(main PRIVATE
250   cxx_std_20
251 )
252 target_compile_features(main_openblas PRIVATE
253  cxx_std_20
254 )
255
256 target_link_directories(main PRIVATE ${SUITESPARSE_MAIN_INSTALL_DIR}/lib)
257 target_link_directories(main_openblas PRIVATE ${SUITESPARSE_OPENBLAS_INSTALL_DIR}/
    lib)
258
259 if(WIN32)
260   set(SUITESPARSE_LIB_SUFFIX "_static")
261 endif()
262
263 # Link against SuiteSparse libraries using imported targets
264 target_link_libraries(main PRIVATE
265   cholmod${SUITESPARSE_LIB_SUFFIX}
266  suitesparseconfig${SUITESPARSE_LIB_SUFFIX}
267  amd${SUITESPARSE_LIB_SUFFIX}
268  camd${SUITESPARSE_LIB_SUFFIX}
269   colamd${SUITESPARSE_LIB_SUFFIX}
270   ccolamd${SUITESPARSE_LIB_SUFFIX}
271 )
272 target_link_libraries(main_openblas PRIVATE
273  cholmod${SUITESPARSE_LIB_SUFFIX}
274  suitesparseconfig${SUITESPARSE_LIB_SUFFIX}
275  amd${SUITESPARSE_LIB_SUFFIX}
276  camd${SUITESPARSE_LIB_SUFFIX}
277   colamd${SUITESPARSE_LIB_SUFFIX}
278   ccolamd${SUITESPARSE_LIB_SUFFIX}
279 )
280
281 # Define include directories
282 set(INCLUDES
```

```
283    ${LIB_DIR}/eigen
284   ${LIB_DIR}/fast_matrix_market/include
285  )
286  # Add include directories
287  target_include_directories(main PRIVATE ${INCLUDES}
     ${SUITESPARSE_MAIN_INSTALL_DIR}/include/suitesparse)
288  target_include_directories(main_openblas PRIVATE ${INCLUDES}
     ${SUITESPARSE_OPENBLAS_INSTALL_DIR}/include/suitesparse)
289
290  set(COMPILER_FLAGS "")
291
292  # Set up compiler flags for target main
293  if(MSVC)
294   list(APPEND COMPILER_FLAGS /external:W0 /W4 /EHsc /O2)
295
296    # MKL setup for Windows
297    target_compile_definitions(main PRIVATE -DEIGEN_USE_MKL_ALL -DMKL_ILP64)
298
299    target_include_directories(main PRIVATE $ENV{MKLROOT}/include)
300
301    target_link_directories(main PRIVATE $ENV{CMPLR_ROOT}/lib)
302   target_link_libraries(main PRIVATE
303     MKL::mkl_intel_ilp64
304     MKL::mkl_intel_thread
305      MKL::mkl_core
306      libiomp5md.lib
307   )
308  else()
309   list(APPEND COMPILER_FLAGS -Wall -march=native -O3)
310
311   if(APPLE)
312      execute_process(
313       COMMAND brew --prefix libomp
314       OUTPUT_VARIABLE LIBOMP_PREFIX
315        OUTPUT_STRIP_TRAILING_WHITESPACE
316     )
317      message (STATUS "OpenMP prefix location " ${LIBOMP_PREFIX})
318      target_compile_definitions(main PRIVATE -D_OPENMP -DUSING_ACCEL)
319     # find_package(OpenMP REQUIRED PATHS ${LIBOMP_PREFIX}/lib)
320     target_include_directories(main PRIVATE ${LIBOMP_PREFIX}/include)
```

```
321      # target_compile_definitions(main PRIVATE EIGEN_USE_BLAS EIGEN_USE_LAPACK)
322     target_link_directories(main PRIVATE ${LIBOMP_PREFIX}/lib)
323     target_link_libraries(main PRIVATE
324       "-framework Accelerate"
325        "-lomp"
326      )
327    else() # here i'm linux
328      list(APPEND COMPILER_FLAGS -m64)
329     target_compile_definitions(main PRIVATE -DEIGEN_USE_MKL_ALL -DMKL_ILP64)
330
331      # MKL setup for Linux
332      target_include_directories(main PRIVATE $ENV{MKLROOT}/include)
333
334      target_link_libraries(main PRIVATE
335       -m64 -Wl,--start-group
336       MKL::mkl_intel_ilp64
337       MKL::mkl_intel_thread
338        MKL::mkl_core
339        -Wl,--end-group
340        -liomp5 -lpthread -lm -ldl
341     )
342    endif()
343 endif()
344
345 target_compile_definitions(main_openblas PRIVATE -DEIGEN_USE_BLAS -
    DEIGEN_USE_LAPACK -DOPEN_BLAS)
346 if(APPLE)
347  execute_process(
348     COMMAND brew --prefix openblas
349     OUTPUT_VARIABLE OPENBLAS_DIR
350     OUTPUT_STRIP_TRAILING_WHITESPACE
351  )
352    target_include_directories(main_openblas PRIVATE ${OPENBLAS_DIR}/include)
353    target_link_directories(main_openblas PRIVATE ${OPENBLAS_DIR}/lib)
354 else()
355    target_include_directories(main_openblas PRIVATE ${OPENBLAS_DIR}/include)
356 endif()
357
358 if(WIN32)
359  target_link_libraries(main_openblas PRIVATE
```

```
    ${OPENBLAS_DIR}/lib/libopenblas.lib
  )
elseif(APPLE)
  target_link_libraries(main_openblas PRIVATE
    -lopenblas
 )
else() # here i'm linux
  target_link_libraries(main_openblas PRIVATE
    -fopenmp
   -lopenblas64
 )
endif()

# Finalize the target properties
target_compile_options(main PUBLIC ${COMPILER_FLAGS})
add_dependencies(main copy_override_files suitesparse_main)
install(TARGETS main)

target_compile_options(main_openblas PUBLIC ${COMPILER_FLAGS})
add_dependencies(main_openblas copy_override_files suitesparse_openblas)
install(TARGETS main_openblas)

message(STATUS "CMake configuration complete. Run 'make' or 'cmake --build . --config Release' to build the project.")
```

Codice 2

5.2. MATLAB

File: main.m

```
files = dir("matrices/*.mat"); % Get all .mat files in the matrices folder

matricesNum = length(files);

structArray = repmat(struct('os', string(computer('arch')), 'timestamp', "", 'exception', "None", 'matrixName', "", ...
    'matrixSize', 0, 'rows', 0, 'cols', 0, 'nonZeros', 0, ...
    'loadTime', 0.0, 'loadMem', 0.0, ...
    'decompTime', 0.0, 'decompMem', 0.0, ...
    'solveTime', 0.0, 'solveMem', 0.0, 'relativeError', 0.0), matricesNum, 1);

```

```
11  fprintf("- Processing %d matrices...\n", matricesNum);
12
13  for i = 1:matricesNum
14      name = files(i).name;
15      matrixName = erase(name, ".mat");
16      fprintf("  - Processing matrix %d/%d: %s\n", i, matricesNum, matrixName);
17      structArray(i).timestamp = datetime;
18      structArray(i).matrixName = matrixName;
19
20      try
21          java.lang.System.gc();
22          pause(10);
23
24          % Read matrix
25          fprintf("    1. Loading matrix...\n");
26          profile clear;
27          profile -memory on;
28          matData = load(fullfile("matrices", name));
29          profile off;
30
31          [loadTime, loadMemAlloc, loadMemFreed] = getProfileResults();
32          profile clear;
33
34          A = matData.Problem.A;
35          clear matData;
36          [rows, cols] = size(A);
37          nonZeros = nnz(A);
38          aBytes = whos('A').bytes;
39
40          fprintf("      ✓ Matrix loaded (%.2f ms), Memory used (%d) \n", loadTime,
            loadMemAlloc + loadMemFreed);
41          fprintf("      ✓ Matrix properties:\n");
42          fprintf("        - Matrix type: %s\n", class(A));
43          fprintf("        - Matrix Memory Usage: %d\n", aBytes);
44          fprintf("        - Matrix size: %d x %d\n", rows, cols);
45          fprintf("        - Non-zero entries: %d\n", nonZeros);
46
47          structArray(i).loadTime = loadTime;
48          structArray(i).loadMem = loadMemAlloc + loadMemFreed;
49          structArray(i).matrixSize = aBytes;
```

```
50          structArray(i).rows = rows;
51          structArray(i).cols = cols;
52          structArray(i).nonZeros = nonZeros;
53
54          % Cholesky decomposition
55          fprintf("    2. Performing Cholesky decomposition with AMD Ordering ...
            \n");
56
57          profile clear;
58          profile -memory on;
59          [R, flag, perm] = chol(A, 'vector');
60          profile off;
61
62          [decompTime, decompMemAlloc, decompMemFreed] = getProfileResults();
63          profile clear;
64
65          % Check if the matrix is not symmetric positive definite
66          if flag ~= 0
67              structArray(i).exception = "The matrix is not symmetric positive
                definite";
68              fprintf("      ⚠ Cholesky decomposition failed (%.2f ms), Memory used
                (%d)\n", decompTime, decompMemAlloc + decompMemFreed);
69
70              clear A;
71              clear perm;
72              clear R;
73              continue;
74          end
75
76          structArray(i).decompTime = decompTime;
77          structArray(i).decompMem = decompMemAlloc + decompMemFreed;
78
79          fprintf("      ✓ Cholesky decomposition completed (%.2f ms), Memory used
            (%d)\n", decompTime, decompMemAlloc + decompMemFreed);
80
81          % Define expected solution
82          xe = ones(cols, 1);
83          b = A(perm, perm) * xe;
84
85          clear A;
86          clear perm;
```

```
87
88          % Solve system
89          fprintf("    3. Solving system...\n");
90          profile clear;
91          profile -memory on;
92          x = R\(R'\b);
93          profile off;
94
95          [solveTime, solveMemAlloc, solveMemFreed] = getProfileResults();
96          profile clear;
97
98          fprintf("      ✓ System solved (%.2f ms)\n", solveTime);
99
100         structArray(i).solveTime = solveTime;
101         structArray(i).solveMem = solveMemAlloc + solveMemFreed;
102
103         clear R;
104         clear b;
105
106         % Relative error
107         fprintf("    4. Calculating relative error...\n");
108         relativeError = norm(x - xe) / norm(xe);
109         fprintf("      ✓ Relative error: %.2e\n", relativeError);
110
111         clear x;
112         clear xe;
113
114         structArray(i).relativeError = relativeError;
115         structArray(i).exception = "";
116     catch exception
117         structArray(i).exception = replace(exception.message, newline, ' - ');
118         warning("  ⚠ Error processing %s: %s\n", matrixName, exception.message);
119
120         clear A;
121         clear perm;
122         clear R;
123         clear b;
124         clear x;
125         clear xe;
126
```

```
127         continue;
128     end
129     fprintf("  - Processed matrix %d/%d: %s\n", i, matricesNum, matrixName);
130 end
131
132 % Write results to CSV
133 resultFile = "bench_" + computer('arch') + ".csv";
134 fprintf("- Writing results to " + resultFile + "...\n");
135
136 results = struct2table(structArray);
137 writetable(results, resultFile);
138
139 fprintf("✓ Results saved.\n");
140 fprintf("✓ All matrices processed!\n");
```

Codice 3: main

File: getProfileResults.m

```
1  function [execTime, memAllocated, memFreed] = getProfileResults()
2  % GETPROFILERESULTS Get the profiling results for the last profile session.
3  %
4  %   [execTime, memAllocated, memFreed] = getProfileResults()
5  %
6  %   Returns:
7  %     execTime     - Execution time in milliseconds
8  %     memAllocated - Memory allocated in bytes
9  %     memFreed     - Memory freed in bytes
10 %
11 %   Example:
12 %     [time, memAlloc, memFreed] = getProfileResults();
13
14     p = profile('info');
15     profileData = p.FunctionTable;
16
17     % Extract both timing and memory information
18     execTime = sum([profileData.TotalTime]) * 1000;
19     memAllocated = sum([profileData.TotalMemAllocated]);
20     memFreed = sum([profileData.TotalMemFreed]);
21 end
```

Codice 4: helper function

File: sparse_lib_versions.m

MATLAB

```
function sparse_lib_versions
    spparms('spumoni',2)

    load west0479
    C = west0479;
    A = C * C';
    n = size(A,1);
    b = rand(n,1);

    filename = fullfile(tempdir, 'sparse_info.txt');
    fclose(fopen(filename,'wt'));  % erase existing file
    diary(filename);

    chol(A);
    p = amd(A);
    p = colamd(A);
    x = C \ b;
    x = A \ b;

    S = C (:, 1:n-1);
    x = S \ b;

    diary off
    spparms('spumoni',0)

    hyperlink = ['<a href="matlab:edit(''' filename ''')">' filename '</a>'];
    fprintf('\n=> sparse matrix library versions (from %s):\n', hyperlink);
    fprintf(  '============================================================\n');

    try
        % Open the file for reading
        fileID = fopen(filename, 'r');
        if fileID == -1
            error('File could not be opened.');
        end

        % Read the file line by line
        while ~feof(fileID)
            line = fgets(fileID); % Get the current line

```

```
41              % Check for the desired patterns
42              if contains(line, 'version ')
43                  disp(['Found: ', line]); % Display the matching line
44              end
45              if contains(line, 'UMFPACK V')
46                  disp(['Found: ', line]); % Display the matching line
47              end
48          end
49
50          % Close the file after reading
51          fclose(fileID);
52      catch ME
53          % Handle errors gracefully
54          disp(['Error: ', ME.message]);
55      end
56  end
```

Codice 5: sparse lib versions

Bibliografia

[1] T. Davis, «DrTimothyAldenDavis/SuiteSparse». Consultato: 9 aprile 2025. [Online]. Disponibile su: https://github.com/DrTimothyAldenDavis/SuiteSparse

[2] «BLAS (Basic Linear Algebra Subprograms)». Consultato: 8 maggio 2025. [Online]. Disponibile su: https://netlib.org/blas/

[3] «LAPACK — Linear Algebra PACKage». Consultato: 8 maggio 2025. [Online]. Disponibile su: https://netlib.org/lapack/