Metodi del Calcolo Scientifico

Progetto 1

Risoluzione di sistemi lineari tramite il metodo di Cholesky

Edoardo Silva 816560
Bryan Zhigui 816335
Davide Marchetti 815990

A.A. 2019/2020

Abstract

Questo progetto si prefigge lo scopo di studiare l'implementazione del metodo di Choleski per la risoluzione sistemi lineari per matrici simmetriche definite positive sparse in un ambiente di programmazione open source e di compararli con l'implementazione closed-source di MATLAB.

Il confronto avverrà in termini di tempo, accuratezza, impiego della memoria e anche facilità d'uso sia in ambiente Linux che Windows, eseguendo il codice su diverse matrici derivate da problemi reali e raccolte nella **SuiteSparse Matrix Collection**[1].

[1] https://sparse.tamu.edu/

1 Analisi dell'implementazione

1.1 MATLAB

Per la decomposizione di cholesky, MATLAB mette a disposizione il modulo `Symbolic Math Toolkit` contenente tutto il necessario. In particolare è stata utilizzata la funzione `chol`[2].

Utilizzo

`R = chol(A, [triangle])`: fattorizza la matrice A simmetrica definita positiva in una matrice triangolare superiore R tale che $A = R^{-1}R$.

Il parametro `triangle` permette di scegliere se attuare la decomposizione in una matrice triangolare superiore (default) o triangolare inferiore. In quest'ultimo caso, la matrice R risultante dall'equazione soddisferà l'uguaglianza $A = RR^{-1}$.

Manutenzione

La libreria è stata rilasciata per la prima volta nell'aggiornamento R2013a MATLAB. Attualmente, è ancora supportata e non presenta lacune o problemi riscontrati durante il suo utilizzo.

Documentazione

La documentazione ufficiale di MATLAB è ben strutturata e di facile consultazione. Vengono anche proposti molti esempi di utilizzo delle diverse funzioni in varie casistiche.

Licenza

Essendo MATLAB un software closed-source, non è possibile accedere al codice sorgente del modulo.

[2] https://www.mathworks.com/help/matlab/ref/chol.html

Problemi riscontrati

La criticità principale riguarda il supporto del modulo `memory` esclusivamente per sistemi operativi Windows. Difatti, è stato necessario utilizzare due metodi di profiling della memoria diversi a seconda del sistema operativo di esecuzione.

1.2 Open-Source (C++)

Dopo un'attenta analisi e comparazione di diverse opzioni, l'implementazione in C++ è stata costruita utilizzando **Eigen**, libreria che si pone l'obiettivo di essere leggera ed offrire supporto alle operazioni su vettori e matrici dense e sparse.

Utilizzo

`Eigen::loadMarket(A, filename)`: importa i valori di una matrice sparsa memorizzata in un file `.mtx` nella matrice fornita come primo argomento. Nel nostro programma, `A` è definita come `Eigen::SparseMatrix<double>`.

Il modulo `unsupported/Eigen/SparseExtra` che contiene queste funzionalità è attualmente deprecato.

Manutenzione

Eigen è in sviluppo attivo, tuttavia, alcuni moduli sono marcati come deprecati e non ne è garantito il loro pieno funzionamento. Un esempio di questi è la classe `MarketIO`, che permette di effettuare operazioni di input e ouput con file in formato `Matrix Market (.mtx)`.

Documentazione

La documentazione di Eigen è di facile consulto e abbastanza completa. Tuttavia, non si può fare la stessa considerazione rispetto ai moduli inseriti nel namespace `unsupported`, per i quali la documentazione è spesso mancante o riferita a versioni precedenti.

Licenza

Eigen è un software gratuito ed open-source rilasciato con licenza Mozilla Public License 2.0 (MPL2: simple weak copyleft license) dalla versione 3.1.1.

Problemi riscontrati

L'utilizzo di una classe deprecata ha inizialmente rallentato lo sviluppo. Infatti, delle matrici importate tramite `MarketIO` veniva ignorato il fatto che fossero salvate come simmetriche o meno.

Questo inconveniente è stato risolto modificando lo script di conversione MATLAB `mmwrite` per trasformare file `.mat` in `.mtx` e rigenerando le matrici assicurandosi che tutti gli elementi venissero salvati correttamente. Lo script utilizzato per la conversione è riportato nella sezione 8.

Tuttavia, questa trasformazione aggiuntiva comporta uno spreco di spazio su disco per memorizzare il doppio dei valori rispetto ad un semplice indicatore che specifichi qualora la matrice sia simmetrica, e di tempo per la conversione dei file e per la lettura del doppio delle righe da file durante l'esecuzione.

2 Dettagli implementativi

Riporteremo di seguito le sezioni semplificate delle parti principali di ciascun'implementazione. Lo schema di entrambi gli algoritmi coincide, le differenze sono date dalle peculiarità del linguaggio utilizzato.

Entrambi gli algoritmi accettano in input come parametro il percorso del file da analizzare ed effettuano le seguenti operazioni:

1. Carico la matrice M dal file specificato in input
2. Memorizzo l'occupazione attuale della memoria
3. Creo la soluzione esatta x_{es} come vettore di elementi con valore 1 pari alla dimensione della matrice M
4. Calcolo il vettore b dei termini noti
5. Applico la decomposizione di Cholesky con il metodo scelto fornendo in input la matrice M e il vettore b memorizzando il tempo di esecuzione ed ottenendo la soluzione approssimata x_{ap}
6. Calcolo la memoria utilizzata subito dopo la risoluzione del sistema
7. Calcolo l'errore relativo tra x_{ap} e x_{es}
8. Restituisco tutte le metriche registrate al chiamante che si occuperà della memorizzazione su file

2.1 MATLAB

In particolare, MATLAB accetta in input un file che si assume essere una matrice memorizzata in formato `.mat`. Questo permette di caricarla semplicemente tramite la funzione `load`.

Il listato 1 riporta la procedura descritta in precedenza utilizzata nella nostra implementazione.

Listing 1: MATLAB: Algoritmo principale

```
function [rows, memory_delta, solve_time, error] = chol_solve(name)
    load(fullfile('', 'matrix_mat', name), "Problem");

    [user] = memory;
    proc_memory_start = user.MemUsedMATLAB;

    rows = size(Problem.A, 1);
    x_es = ones(rows, 1);
    b = Problem.A * x_es;

    tic;
    R = chol(Problem.A);
    x_ap = R \ (R' \ b);
    solve_time = toc;

    [user] = memory;
    memory_delta = user.MemUsedMATLAB - proc_memory_start;

    error = norm(x_ap - x_es) / norm(x_es);
end
```

2.2 C++

Nell'implementazione C++ per la funzione `analyze_matrix` si è scelto di definire una `struct` contenente tutti i campi che era necessario ottenere in output (listato 2).

Questo permette di semplificare il codice evitando di utilizzare più parametri passati per indirizzo come valori di output.

Listing 2: C++: Struct per la memorizzazione del risultato

```
typedef unsigned long long ull;
struct result {
    unsigned int size;
    ull memory_delta;
    std::chrono::duration<double> solve_time;
    double relative_error;
};
```

L'implementazione del solutore ricalca gli step definiti precedentemente. La lettura della memoria utilizzata avviene utilizzando metodi di una classe personalizzata scritta ad-hoc per il progetto e riportata nella sezione 8.

Listing 3: C++: Algoritmo principale

```cpp
result analyze_matrix(std::string filename) {
    result r;
    SpMat A; // Eigen::SparseMatrix<double>
    ull start_mem, end_mem;

    Eigen::loadMarket(A, filename);
    r.size = A.rows();

    start_mem = memory::process_current_physical();

    Eigen::VectorXd x_es = Eigen::VectorXd::Ones(A.rows());
    Eigen::VectorXd b(A.rows());
    b = A*x_es;

    auto chol_start = std::chrono::high_resolution_clock::now();
    Eigen::SimplicialCholesky<SpMat> chol(A);
    Eigen::VectorXd x_ap = chol.solve(b);
    auto chol_finish = std::chrono::high_resolution_clock::now();

    end_mem = memory::process_current_physical();

    r.memory_delta = end_mem - start_mem;
    r.solve_time = chol_finish - chol_start;
    r.relative_error = (x_ap - x_es).norm() / x_es.norm();
}
```

3 Specifiche hardware

La piattaforma utilizzata per la produzione dei risultati riportati nelle sezioni 5 e 6, è composta come segue:

- **CPU**: AMD Ryzen 5 3600 - 6 Core / 12 Threads - 3.60Ghz /4.20Ghz
- **RAM**: Crucial Ballistix DDR4-3000C15 2*8Gb (16Gb) a 3800MHz
- **HDD**: Western Digital Green 1TB HDD
- **GPU**: Sapphire RX 580 Pulse (8GB VRam)

Il disco utilizzato per l'esecuzione in ambiente Windows è un **SSD Samsung 850 EVO (250Gb)**, mentre Linux è installato su un **NVMe Sabrent (256Gb)**.

4 Matrici analizzate

L'analisi si è concentrata sulle seguenti matrici:

Nome	Dimensione	NNZ
ex15	6.867	98.671
shallow_water1	81.920	327.680
cfd1	70.656	1.825.580
cfd2	123.440	3.085.406
parabolic_fem	525.825	3.674.625
apache2	715.176	4.817.870
G3_circuit	1.585.478	7.660.826
StocF-1465	1.465.137	21.005.389
Flan_1565	1.564.794	114.165.372

Tabella 1: Matrici analizzate

5 Risultati per sistema operativo

5.1 Windows

Tempo

All'incremento delle dimensioni della matrice di input non si manifesta una crescita lineare né costante in termini di tempo. Ciononostante, i tempi di esecuzione nell'implementazione in MATLAB risultano essere meno variabili rispetto alla controparte in C++.

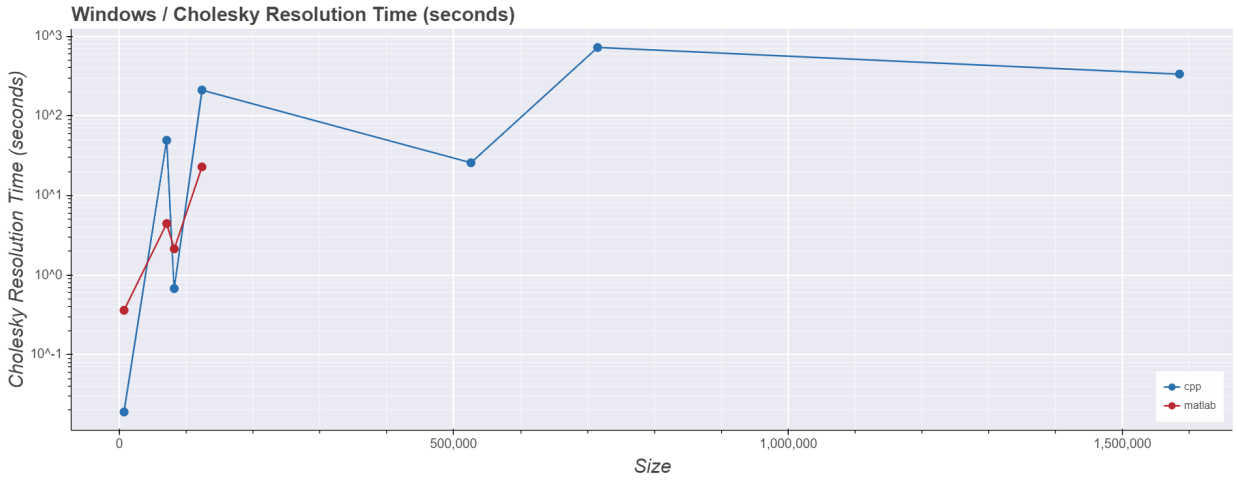

Figura 1: Tempo di risoluzione su Windows

Memoria

Come illustrato in fig. 2, l'implementazione in C++ occupa molta meno memoria rispetto a quella in MATLAB, in particolare al crescere della dimensione della matrice e del numero di elementi non nulli contenuti in essa.

Difatti, tutte le matrici analizzate in MATLAB per le quali non è presente un risultato nel grafico, durante la decomposizione di Cholesky hanno comportato un errore `Out of memory`.

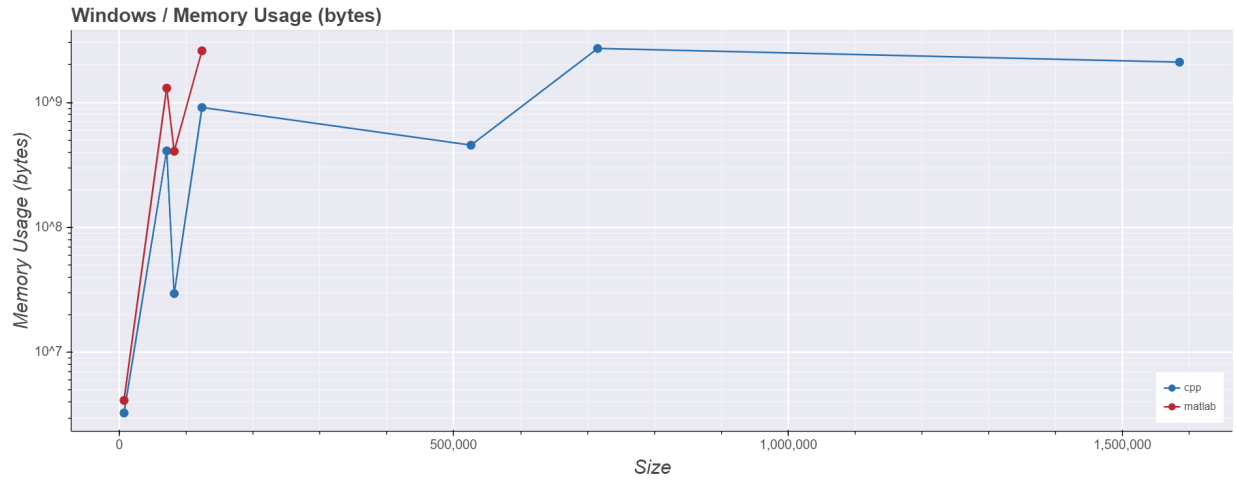

Figura 2: Utilizzo della memoria su Windows

Errore Relativo

Entrambe le implementazioni presentano un errore relativo molto simile. In fig. 3 si nota come al crescere della dimensione della matrice per C++ l'errore relativo sembra stabilizzarsi tra 10^{-10} e 10^{-12}.

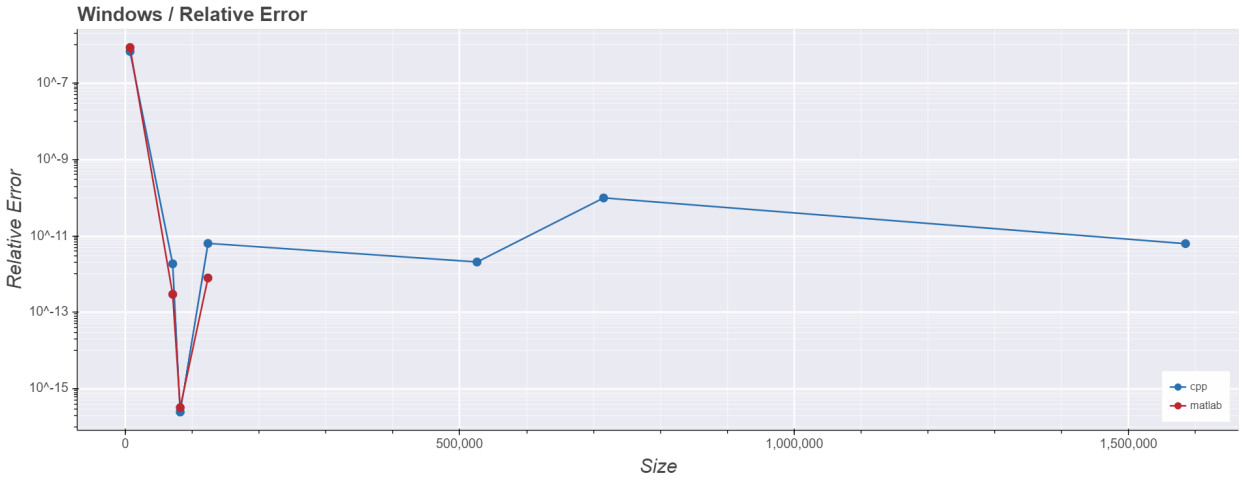

Figura 3: Errore relativo su Windows

5.2 Linux

Tempo

L'andamento del tempo di risoluzione del sistema non risulta strettamente dipendente dalla dimensione della matrice. Inoltre, l'implementazione in MATLAB riesce a mantenere tempi di completamento meno variabili.

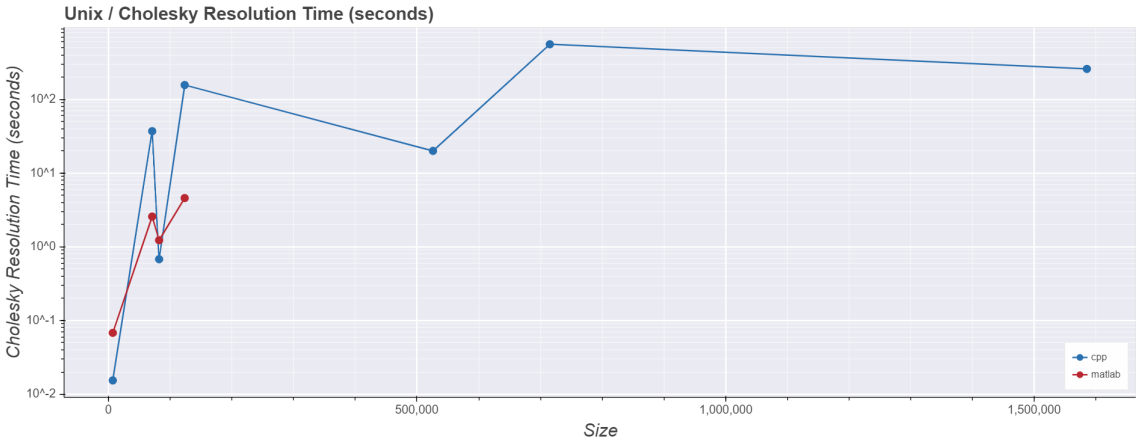

Figura 4: Tempo di risoluzione su Linux

Memoria

L'utilizzo della memoria, a differenza di quanto visto in precedenza nella fig. 2 presenta una lettura iniziale molto bassa nell'implementazione MATLAB.

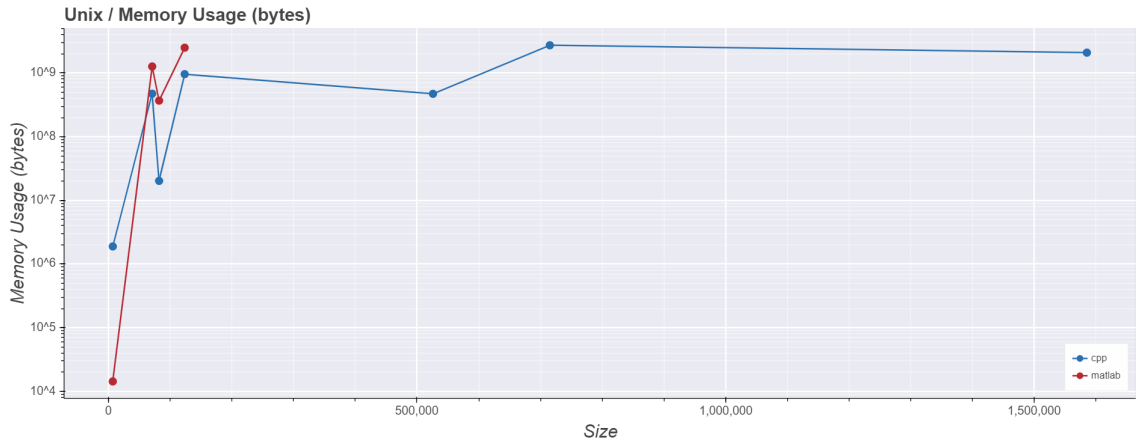

Figura 5: Utilizzo della memoria su Linux

Errore relativo

Si nota che, sia su c++ che con la libreria MatLab, si ha un errore comparabile per entrambi i metodi, e che dopo un massimo ed un minimo nelle matrici più piccole, l'errore si stabilizza intorno a 10^{-11}, indipendentemente dalle dimensioni delle matrici che il programma riesce a caricare.

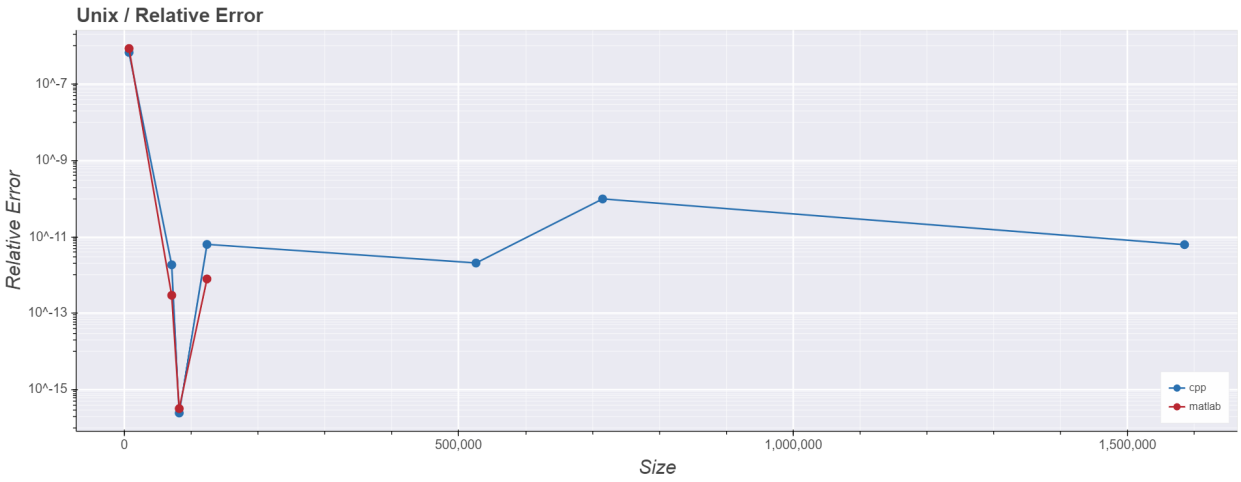

Figura 6: Errore relativo su Linux

6 Risultati per implementazione

6.1 C++

Tempo

Come illustrato in fig. 7, Linux risulta più efficiente di Windows nell'esecuzione dell'implementazione C++ per tutte le matrici analizzate.

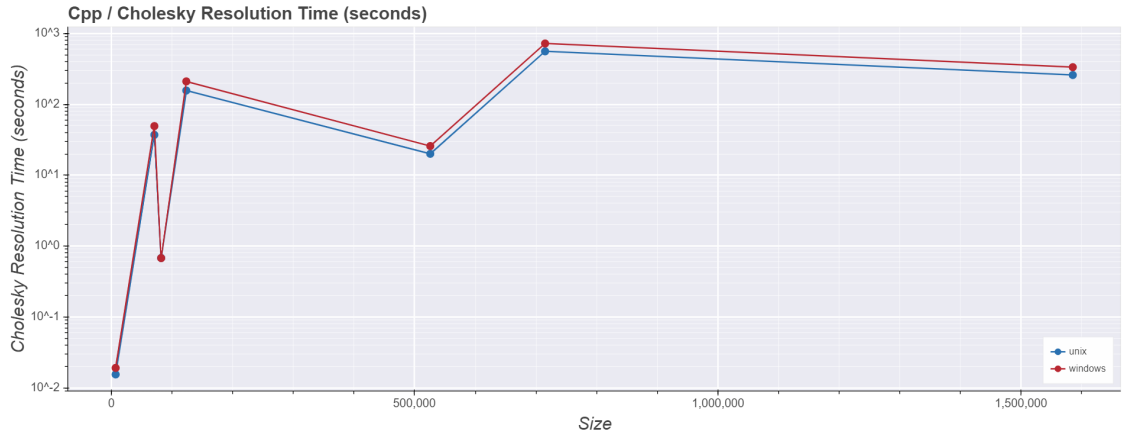

Figura 7: Tempo di risoluzione nell'implementazione C++

Errore Relativo

L'errore relativo non risulta influenzato dal diverso ambiente sul quale viene eseguita l'implementazione.

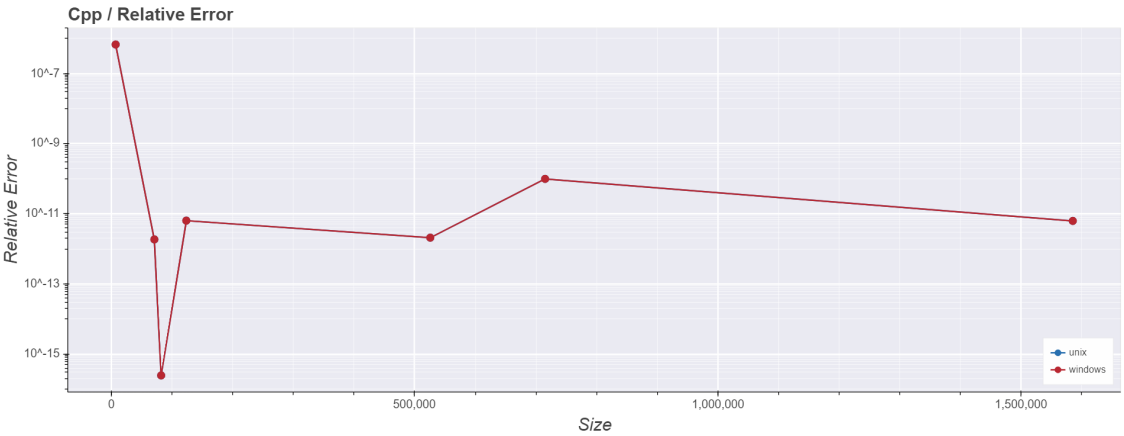

Figura 8: Errore relativo nell'implementazione C++

Memoria

Al crescere della dimensione della matrice, la memoria impiegata per la risoluzione tende ad essere pressoché identica nei due ambienti.

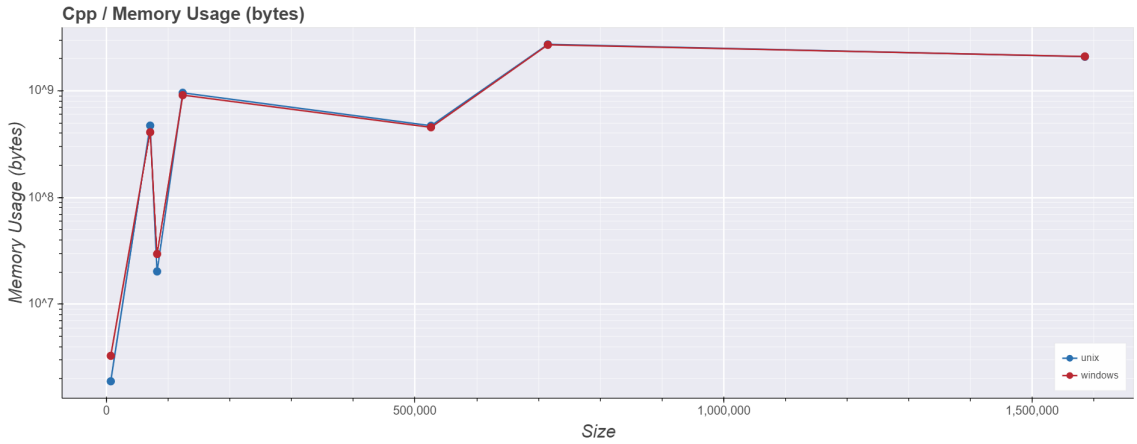

Figura 9: Utilizzo della memoria nell'implementazione C++

6.2 Matlab

Tempo

L'esecuzione dello script MATLAB in ambiente Linux risulta essere più efficiente. In questo caso, la differenza risulta ancora più marcata rispetto all'implementazione C++ illustrata in fig. 7.

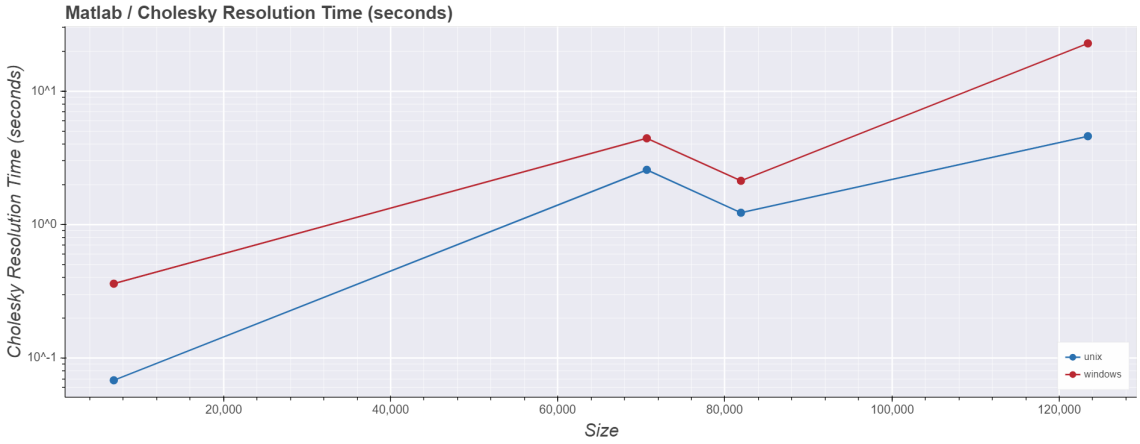

Figura 10: Tempo di risoluzione nell'implementazione MATLAB

Errore Relativo

L'errore relativo dei risultati della libreria Matlab non cambia a seconda delle architettetture.

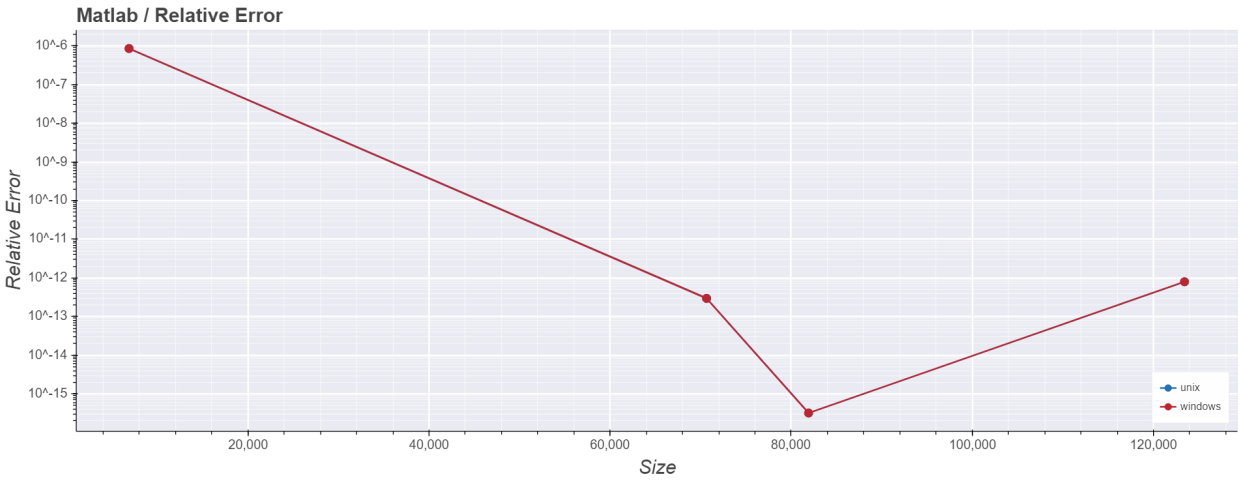

Figura 11: Errore relativo nell'implementazione MATLAB

Memoria

La differenza di memoria utilizzata nelle due architetture è visibile solo sull'esecuzionen della prima matrice, le restanti impiegano pressoché lo stesso quantitativo di memoria.

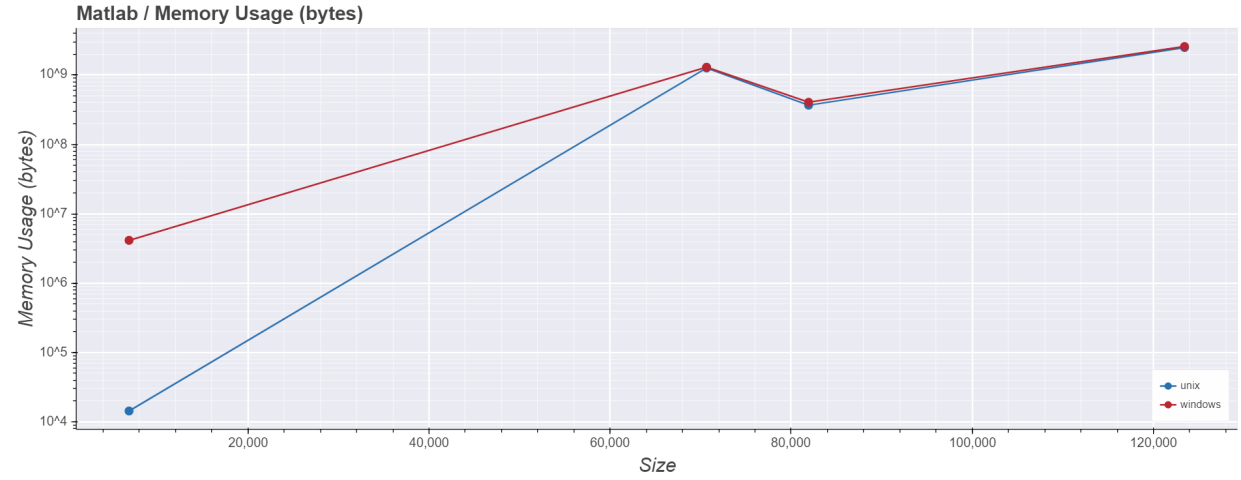

Figura 12: Utilizzo della memoria nell'implementazione MATLAB

7 Conclusioni

Dai risultati ottenuti, il problema principale di MATLAB risulta l'eccessivo utilizzo di memoria RAM. Al contrario, C++ riesce a risolvere matrici di dimensioni più grandi grazie ad una gestione più efficiente ed un minor overhead del linguaggio.

Analizzando i tempi di esecuzione invece, la situazione si ribalta. MATLAB riesce a ridurre notevolmente i tempi di attesa per la soluzione del sistema soprattutto al crescere della dimensione della matrice. Invece, C++ impiega un tempo maggiore nell'ordine di alcuni minuti (comparabile solo sulle matrici per le quali si è riusciti ad avere una soluzione in entrambe le implementazioni).

Questa differenza così marcata è spiegabile facilmente: MATLAB è un software che esegue calcoli in parallelo e sfrutta tutti i core presenti sulla sistema utilizzato, inoltre le operazioni più comuni in ambito matematico ci si aspetta che siano ottimizzate a dovere. D'altro canto, l'implementazione C++ al momento è un processo single core, non riuscendo quindi a sfruttare la totale potenza del processore ed impiegando più tempo.

Questa ipotesi è rafforzata dal confronto tra fig. 4 e 7, infatti MATLAB riesce a beneficiare dell'esecuzione multicore rimarcando ancora di più la differenza nei tempi di esecuzione rispetto a C++.

Analizzando le due implementazioni sui diversi sistemi operativi possiamo affermare con sicurezza che Linux riesca ad eseguire in modo più prestante le due implementazioni.

In conclusione, i risultati mostrano come la libreria Eigen per C++ prediliga un utilizzo più efficiente della memoria, penalizzando il tempo di esecuzione. Questo, a parità di hardware, permette di lavorare su matrici di dimensioni maggiori a discapito di un codice più complesso e meno intuitivo.

8 Listati di codice

Listing 4: Main dell'implementazione C++

```cpp
int main() {
    result _result;
    std::ofstream output(OUTPUT_FILE, std::ofstream::out);

    // Write headers
    output << "filename" << ","
        << "size" << ","
        << "memory_delta" << ","
        << "solve_time" << ","
        << "relative_error" << CSV_EOL;

    // Look for matrix in ../matlab/matrix_mtx folder
    std::string path = "../matlab/matrix_mtx";
    for (const auto& entry : std::filesystem::directory_iterator(path)) {
        if (entry.path().extension() == ".mtx") {
            if (! output.is_open()) {
                output.open(OUTPUT_FILE, std::ofstream::app);
            }

            std::cout << entry.path() << std::endl;
            _result = analyze_matrix(entry.path().string());

            output << entry.path().stem() << "," << _result << CSV_EOL;
                ↪ output.close();
        }
    }

    return 0;
}
```

Listing 5: Lettura della memoria in C++

```cpp
// Adapted from: https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-consumption-from-inside-a-process
unsigned long long memory::process_current_physical() {
#ifdef OS_WIN
    PROCESS_MEMORY_COUNTERS_EX pmc;
    GetProcessMemoryInfo(GetCurrentProcess(), (PROCESS_MEMORY_COUNTERS*)&pmc, sizeof(pmc));

    return pmc.WorkingSetSize;
#elif defined(OS_NIX)
    long rss = 0L;
    FILE* fp = NULL;

    if ((fp = fopen("/proc/self/statm", "r")) == NULL) {
        return (size_t)0L;
    }

    if (fscanf(fp, "%*s%ld", &rss) != 1) {
        fclose(fp);
        return (size_t) 0L;
    }

    fclose(fp);
    return (size_t) rss * (size_t) sysconf(_SC_PAGESIZE);
#endif
}
```

Listing 6: Main dell'implementazione MATLAB

```
path = fullfile('matrix_mat', '*.mat');
files = dir(path);
filesCount = length(files);

results = ["filename", "size", "memory_delta", "solve_time", "relative_error
    ↪ "];

for K = 1 : filesCount
  filename = convertCharsToStrings(files(K).name);
  [size, memory_delta, solve_time, relative_error] = chol_solve(filename, 1)
      ↪ ;

  results = [results(1:K,:); [strtok(filename, '.'),size,memory_delta,
      ↪ solve_time,relative_error]];
  clearvars filename size memory_delta solve_time relative_error;
end

if isunix
  writematrix(results, 'unix-output.csv');
elseif ispc
  writematrix(results, 'windows-output.csv');
end
```

Listing 7: Funzione di conversione da `.mat` a `.mtx` per matrici simmetriche

```
function [ err ] = mmwrite_symmetric(filename,A)
mattype = 'real';
precision = 16;
comment = '';

mmfile = fopen([filename],'w');
if (mmfile == -1)
  error('Cannot open file for output');
end;

[M,N] = size(A);
if (issparse(A))
  [I,J,V] = find(A);

  issymm = 0;
  symm = 'general';
  [I,J,V] = find(A);
  NZ = nnz(A);

  rep = 'coordinate';

  fprintf(mmfile, '%%%%MatrixMarket matrix %s %s %s\n', 'coordinate',
      ↪ mattype, symm);

  [MC,NC] = size(comment);
  if (MC == 0)
    fprintf(mmfile, '%% Generated %s\n', [date]);
  else
    for i = 1:MC,
      fprintf(mmfile, '%%%s\n', comment(i, :));
    end
  end

  fprintf(mmfile, '%d %d %d\n', M, N, NZ);
  realformat = sprintf('%%d %%d %% .%dg\n', precision);

  for i = 1:NZ
    fprintf(mmfile, realformat, I(i), J(i), V(i));
  end;
end
fclose(mmfile);
```