Password Decryption using DES Algorithm: Sequential vs Parallel version

Luca Leuter
E-mail address
luca.leuter@stud.unifi.it

Abstract

In questo paper verranno analizzate le performance di un'implementazione dell'algoritmo di cifratura DES (Data Encryption Standard) in versione sequenziale e in due versioni parallele

1. Introduction

Nel mondo moderno la sicurezza informatica è fondamentale, poiché i sistemi informatici sono entrati a far parte del quotidiano di quasi ogni persona sul pianeta. Uno dei primi algoritmi di cifratura è stato DES (Data Encryption Standard) del 1972, basato sul Cifrario di Feistel. Nel corso degli anni DES è stato migliorato, con le versioni Triple DES e con AES (Advanced Encryption Standard). In questo progetto si vedrà una versione sequenziale dell'algoritmo DES implementata in C++ che verrà messa a confronto con una versione parallela implementata in C++ usando la libreria OPENMP e un'altra versione parallela implementata in Python usando la libreria JOBLIB

2. Implementation

Prima di vedere le caratteristiche specifiche di ogni implementazione vediamo le caratteristiche dell'algoritmo DES. Per fare ciò facciamo riferimento alla Figura 1 I passi della fase di Encryption sono:

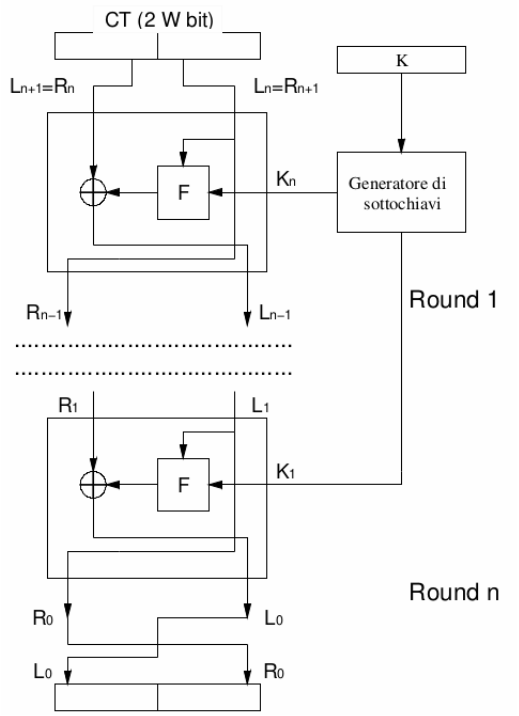

Figure 1. L'algoritmo DES

- Il plaintext è diviso in due metà di W bit ciascuna (nel nostro caso $W = 32$), L_0 ed R_0
- Queste due metà passano attraverso n round che hanno tutti la stessa struttura, ma usano n sotto-chiavi K_1, ..., K_n , generate a partire dalla chiave principale K. Le sottochiavi sono tutte diverse tra di loro e diverse da K.
- La metà destra passa attraverso la funzione F, che realizza l'operazione di sostituzione. Essa poi viene combinata, tramite xor, alla parte sinistra: questa va a costituire la metà destra dell'output, mentre la metà sinistra sarà costituita dalla metà destra in ingresso.

- La funzione F è la stessa per tutti i round, e prende come argomenti R_i e K_i
- Le due metà escono dall'ultimo round come Ln e Rn, che ricombinate formano il cyphertext

Per decifrare un ciphertext occorre semplicemente farlo passare di nuovo attraverso l'algoritmo, avendo cura però di usare le n sottochiavi in ordine inverso: K_n, K_{n-1}, ..., K_1 Vediamo ora come viene implementata la funzione F. Nei cifrari DES e AES la funzione F impiega al suo interno le cosiddette Substitution-Box, in breve S-Box. Una S-box è una tabella che associa ad ogni possibile blocco di m bit in ingresso un blocco di n bit in uscita. Nella Figura 2 vediamo i suoi dettagli implementativi.

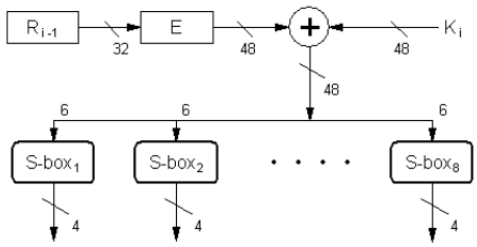

Figure 2. Dettaglio implementativo della funzione F

Nella figura, R_{i-1} rappresenta la metà destra in ingresso al round i-esimo, mentre E è una funzione di espansione che porta i bit da 32 a 48. La sostituzione consiste in un insieme di otto S-box, ognuna delle quali accetta 6 bit come input e produce un output di 4 bit. Ognuna di esse è una fissata matrice 4×16: ciascuna delle 4 righe è una permutazione degli interi compresi fra 0 e 31. L'input di 6 bit seleziona una riga ed una colonna, e il relativo valore, rappresentato in binario, è l'output. La riga viene selezionata dal numero binario formato dal primo e dall'ultimo dei 6 bit di input, mentre la colonna è data dai 4 bit centrali. Ad esempio, per l'input 000111, la riga è 01 (1 in decimale), per cui viene selezionata la seconda riga, mentre la colonna è 0011 (cioè 3 in decimale), corrispondente alla quarta colonna.

2.1. C++ Sequential Version

Nella versione sequenziale dell'algoritmo troviamo le seguenti funzioni:

- *convertDecimalToBinary(int decimal)* e *convertBinaryToDecimal(string binary)* da usare per trovare i numeri nelle S-Box e viceversa
- *xor(string a, string b)* che esegue l'operazione di xor tra due stringhe binarie
- *generateKeys()* che genera le chiavi K
- *tablesFiller()* che genera i valori per l'Expansion Table (usata per espandere i bit da 32 a 48) e per le Substitution Boxes
- *reverseKeys()* che inverte l'ordine delle chiavi per alternare tra encryption e decryption
- *DES(string plaintext)* che implementa l'algoritmo vero e proprio
- *sequentialDecryption(vector[string] lines, int size)* che è la funzione che gestisce encryption e decryption di un vettore di stringhe

Si riporta il codice della funzione *sequentialDecryption(vector[string] lines, int size)*:

```
bool sequentialDecryption(vector<string>
    lines, int size){
   bool isCorrect = true;
   generateKeys();
   reverseKeys();
   tablesFiller();
   for (int j = 0; j < size; j++) {
      reverseKeys();
      string pt =
          convertStringToBinary(lines[j]);
      string ct = DES(pt);
      reverseKeys();
      string decrypted = DES(ct);
      string x =
          convertBinaryToString(decrypted);
      if (x != lines[j]){
         isCorrect = false;
         cout<<"FAILED";
         break;
      }
   }
   return isCorrect;
}
```

2.2. C++ Parallel Version with OpenMP

Per implementare la versione parallela dell'algoritmo è stato fatto uso della libreria *OpenMP*. Le funzioni viste nella versione sequenziale rimangono sostanzialmente le stesse, con l'unico cambiamento che riguarda le funzioni *generateKeys()*, *reverseKeys()*, *DES(string plaintext)*. Per loro infatti è necessario passare come argomento un riferimento a *round-keys*, la quale da variabile globale quale era nella versione sequenziale è diventata una variabile locale. Questo perché nella versione parallela dell'algoritmo è stato deciso di parallelizzare sui dati, cioè ogni thread prende un elemento del vettore di password ed esegue encryption, decryption e controllo che siano state eseguite correttamente (tutto ciò è racchiuso nella direttiva *pragma omp for* di *OpenMP*. Per fare ciò ogni thread ha bisogno di avere un vettore di chiavi privato su cui poter lavorare, poiché, come visto precedentemente, il vettore delle chiavi deve essere invertito tra encryption e decryption. La tabella di espansione e le substitution boxes invece possono essere condivise tra i thread, poiché da esse si leggono solo i valori che servono. Si riporta ora la nuova funzione di gestione di encryption e decryption con l'uso delle direttive di OpenMP:

```
bool parallelDecryption(vector<string>
    lines, int size, int nThreads){
   bool isCorrect = true;
   string round_keys[16];
   tablesFillerParallel();
   omp_set_num_threads(nThreads);
#pragma omp parallel private(round_keys)
    shared(expansionTable2,
    substitutionBoxes2)
   {
      generateKeysParallel(round_keys);
      reverseKeysParallel(round_keys);
#pragma omp for
      for (int j = 0; j < size; j++) {
         reverseKeysParallel(round_keys);
         string pt =
             convertStringToBinary2(lines[j]);
         string ct = DESParallel(pt,
             round_keys);
         reverseKeysParallel(round_keys);
         string decrypted =
             DESParallel(ct, round_keys);
         string x =
             convertBinaryToString2(decrypted);
         if (x != lines[j]) {
            cout << x << "-" << lines[j]
                << endl;
            cout << "FAILED";
            isCorrect = false;
         }
      }
   }
   return isCorrect;
}
```

2.3. Python Sequential Version

La versione sequenziale è la stessa della versione sequenziale implementata in C++. Si riporta il codice della funzione principale.

```
   def
       encrypt_and_decrypt_sequential(lines):
      is_correct = True
      tables_filler()
      round_keys = generate_keys()
      reverse_keys(round_keys)
      for line in lines:
         reverse_keys(round_keys)
         pt =
             convert_string_to_binary(line)
         ct = DES(pt, round_keys)
         reverse_keys(round_keys)
         decrypted = DES(ct, round_keys)
         x =
             convert_binary_to_string(decrypted)
         if x != line.strip():
            print(x)
            print(line)
            print("DECRIPTAZIONE FALLITA")
            print()
            is_correct = False
      return is_correct
```

2.4. Python Parallel Version with Joblib

L'altra versione parallela dell'algoritmo è stata implementata in Python facendo uso della libreria Joblib. Essa fornisce la funzione *Parallel* per scrivere codice imbarazzantemente parallelo, permettendo l'uso di una stessa funzione su più processi Python.
Anche in questa versione le funzioni sono le stesse. Per permettere però l'uso della funzione

Parallel della libreria *Joblib* la singola operazione di encryption/decryption è stata incapsulata in una funzione piuttosto che essere lasciata in un ciclo for. In questo modo essa può essere passato come argomento alla funzione *Parallel* nel seguente modo:

```
def encrypt_and_decrypt_parallel(lines,
    job):
   is_correct = True
   tables_filler()
   round_keys = generate_keys()
   Parallel(n_jobs=job)(delayed(single_en_dec)
      (line, round_keys) for line in
         lines)
   return is_correct
```

La funzione *Parallel* ha come parametri il numero di thread usati, la funzione da parallelizzare e gli argomenti che verranno passati a tale funzione. In questo modo ogni processo Python invocato chiamerà la funzione *single-en-dec* usando come argomento una password della lista. Il codice di quest'ultima funzione è il seguente:

```
def single_en_dec(line, round_keys):
reverse_keys(round_keys)
   pt = convert_string_to_binary(line)
   ct = DES(pt, round_keys)
   reverse_keys(round_keys)
   decrypted = DES(ct, round_keys)
   x=convert_binary_to_string(decrypted)
   if x != line.strip():
      print("DECRIPTAZIONE FALLITA")
      return False
return True
```

3. Performance and Test

Per valutare le performance sono stati eseguiti 2 tipologie di test per ogni versione: il primo all'aumentare del numero di thread utilizzati nell'implementazione parallela, tenendo fisso il numero di password; il secondo all'aumentare del numero di password da cifrare e decifrare.

3.1. Test 1 - OpenMP

Per il primo test vengono considerate le prime 5000 password e la versione del codice in C++ usando OpenMP. Le varie righe del file vengono salvate all'interno di un vettore e viene eseguita per ogni password una cifratura e una decifratura in modo sequenziale. Questo viene ripetuto *nTest* volte e infine viene fatta una media dei valori.

```
   ifstream file("password.txt");
   vector<string> lines(nLines);
   for (int j = 0; j < nLines; ++j)
      getline(file, lines[j]);
   int sequentialTime = 0;
   for(int i = 0; i<nTest; i++) {
      auto start = system_clock::now();
      if(!sequentialDecryption(lines,
          testLines)){
         break;
      }
      auto end = system_clock::now();
      auto elapsed =
         duration_cast<milliseconds>(end-start);
      sequentialTime += elapsed.count();
   }
   cout << "Tempo Decriptazione
       Sequenziale: " <<
       sequentialTime/nTest;
```

Successivamente vengono eseguiti i test all'aumentare dei thread. Si arriva fino a un massimo di 16 thread e per ognuno si eseguono *nTest*, calcolando i tempi di esecuzione e facendo la media. Viene anche calcolato lo speedup rispetto alla versione sequenziale

```
int maxThreads = omp_get_max_threads();
   for (int nThreads = 2; nThreads <
       4*maxThreads+1; nThreads++) {
      int parallelTime = 0;
      for (int i = 0; i < nTest; i++) {
         auto start = system_clock::now();
         if(!parallelDecryption(lines,
             testLines, nThreads)){
            break;
         }
         auto end=system_clock::now();
         auto elapsed
            =duration_cast<milliseconds>(end-start);
         parallelTime+=elapsed.count();
      }
      cout << "Tempo Decriptazione
          Parallela usando " << nThreads <<
          " threads: " << parallelTime /
          nTest << endl;
      float speedup = (float)
          sequentialTime/parallelTime;
      cout<<"SpeedUp:
          "<<speedup<<endl<<endl;
   }
```

3.2. Test 2 - OpenMP

Per il secondo test invece viene considerato ogni volta un sottoinsieme delle password fino ad arrivare a tutte le 10000 password. Per la versione sequenziale del test si parte da 5000 password e si aumenta ogni volta di 1000 calcolando i tempi.

```
for(int n=testLines; n < nLines+1000;
    n+=1000) {
   auto start = system_clock::now();
   if(!sequentialDecryption(lines, n)){
      break;
   }
   auto end = system_clock::now();
   auto elapsed =
      duration_cast<milliseconds>(end-start)
   cout << "Tempo Decriptazione
       Sequenziale con "<<n<<" password:
       "<<elapsed.count()<<endl;
}
```

Per la versione parallela del test si setta il numero di thread usati al massimo possibile (4 nel caso in esame) usando la funzione *omp_set_num_threads(maxThreads)* e si esegue la stesso ciclo del test sequenziale: si parte da 5000 password e si aumenta ogni volta di 1000 calcolando i tempi.

```
for (int n=testLines; n < nLines+1000;
    n+=1000) {
   auto start = system_clock::now();
   if(!parallelDecryption(lines, n,
       maxThreads)){
      break;
   }
   auto end=system_clock::now();
   auto elapsed
     =duration_cast<milliseconds>(end-start)
   cout << "Tempo Decriptazione
       Parallela con " << n << "
       password: " << elapsed.count() <<
       endl;
}
```

3.3. Test 1 - Joblib

Per il primo test vengono considerate le prime 5000 password e la versione del codice in Python usando Joblib. Le varie righe del file vengono salvate all'interno di un vettore e viene eseguita per ogni password una cifratura e una decifratura in modo sequenziale, calcolando i tempi.

```
n_test = 5
file = open('password.txt', 'r')
Lines = file.readlines()
start_time = time.time()
encrypt_and_decrypt_sequential(Lines[0:5000])
end_time = time.time()
print(f'Tempo per decriptazione
    sequenziale: {end_time -
    start_time:.3f} s')
```

Successivamente vengono eseguiti i test all'aumentare dei thread. Si arriva fino a un massimo di 16 thread e per ognuno si eseguono *n-test*, calcolando i tempi di esecuzione e facendo la media

```
for i in range(2, 17, 1):
    test_time = 0
    for j in range(n_test):
       start_time = time.time()
       encrypt_and_decrypt_parallel(Lines[0:5000],
           i)
       end_time = time.time()
       test_time += end_time - start_time
    print(f'Tempo per decriptazione
        parallela
    con {i} jobs: {test_time/n_test:.3f}
        s')
```

3.4. Test 2 - Joblib

Per il secondo test invece viene considerato ogni volta un sottoinsieme delle password fino ad arrivare a tutte le 10000 password. Per la versione sequenziale del test si parte da 5000 password e si aumenta ogni volta di 1000 calcolando i tempi.

```
for i in range(5000, 11000, 1000):
   start_time = time.time()
   encrypt_and_decrypt_sequential(Lines[0:i])
   end_time = time.time()
   print(f'Tempo per decript sequenziale
   di {i}
       password:{end_time-start_time:}s')
```

Per la versione parallela del test si setta il numero di thread usati al massimo possibile (4 nel caso in esame) e si esegue la stesso ciclo del test sequenziale: si parte da 5000 password e si aumenta ogni volta di 1000 calcolando i tempi.

```
for i in range(5000, 11000, 1000):
    start_time = time.time()
    encrypt_and_decrypt_parallel(Lines[0:i],
        4)
    end_time = time.time()
    print(f'Tempo per decriptazione
        parallela con 4 threads di {i}
        password: {end_time -
        start_time:} s')
```

3.5. Setup

I test sono stati eseguiti su un pc con un processore AMD A8-64101 avente 4 core e 4 thread. Il file usato per le password è un txt contenente 10000 password di lunghezza 8 caratteri. Di esse le prime 128 sono una raccolta di password più usate in tutto il mondo, mentre le altre sono state generate casualmente.

3.6. Results OpenMP

I risultati del primo test con OpenMP, in cui si tiene fisso il numero di password a 5000 e si aumenta via via il numero di thread è visibile nella Figura 1.

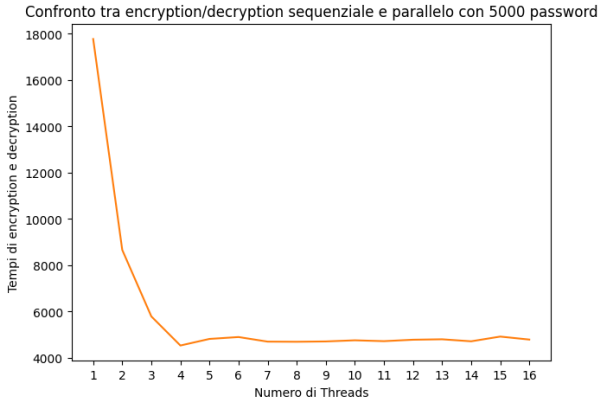

Figure 3. Confronto tra i tempi di caricamento di 5000 password con OpenMP

Il test ha prodotto diversi speedup che sono visibili nella Figura 2.
Si nota dal grafico che lo speedup inizialmente cresce aumentando il numero di thread, fino a raggiungere il massimo quando essi sono 4 (che risulta essere il numero massimo per la macchina). Successivamente decresce oscillando

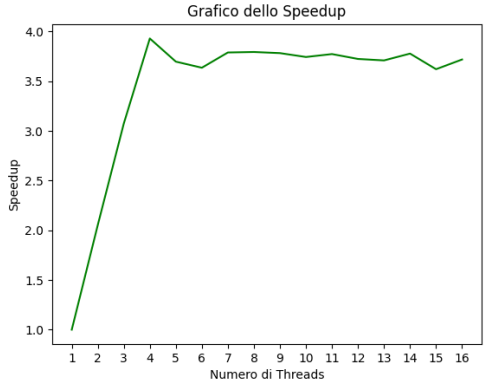

Figure 4. SpeedUp del test con OpenMP aumentando il numero di thread

tra vari valori, sempre migliori dei primi ma inferiori rispetto a quando il numero di thread è 4.

I risultati del secondo test, in cui si confronta il caricamento sequenziale e il caricamento con il numero di thread settato al massimo (in questo caso 4) aumentando il numero di password è visibile in Figura 3.

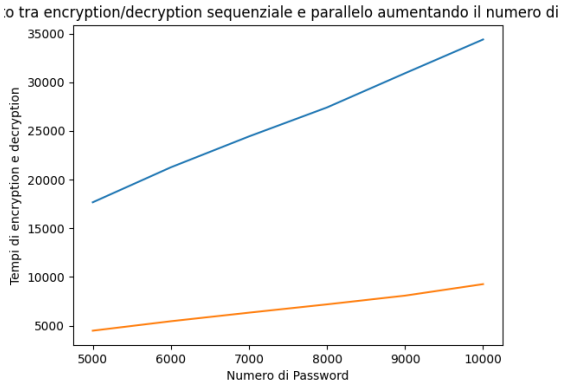

Figure 5. Confronto tra i tempi di cryptazione e decrypt di un numero crescente di password

Il test ha prodotto diversi speedup, visibili in Figura 4.
Si nota dal grafico che lo speedup diminuisce con l'aumentare del numero di password

3.7. Results Joblib

I risultati del primo test con Joblib, in cui si tiene fisso il numero di password a 5000 e si aumenta via via il numero di thread è visibile nella Figura 5.

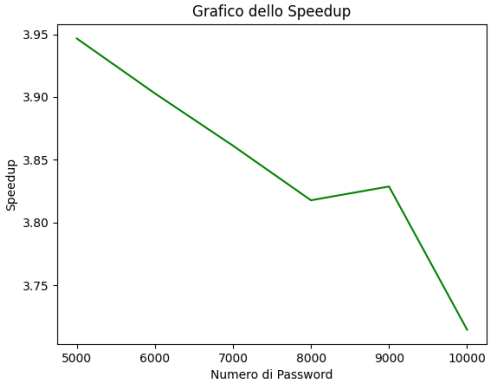

Figure 6. SpeedUp del test aumentando il numero di password

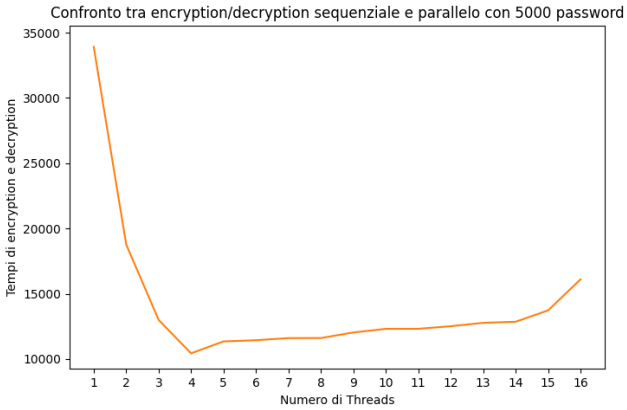

Figure 7. Confronto tra i tempi di caricamento di 5000 password con Joblib

Il test ha prodotto diversi speedup che sono visibili nella Figura 2.

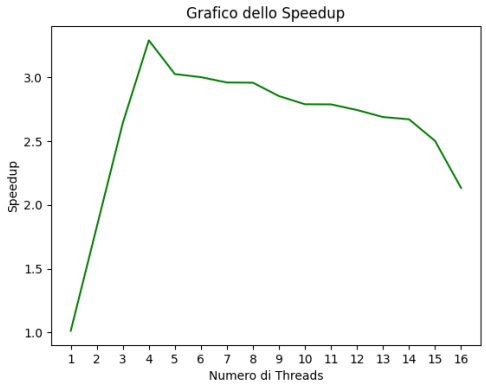

Figure 8. SpeedUp del test con Joblib aumentando il numero di thread

Si nota dal grafico che lo speedup inizialmente cresce aumentando il numero di thread, fino a raggiungere il massimo quando essi sono 4 (che risulta essere il numero massimo per la macchina). Successivamente decresce con l'aumentare del numero di thread.

I risultati del secondo test, in cui si confronta il caricamento sequenziale e il caricamento con il numero di thread settato al massimo (in questo caso 4) aumentando il numero di password è visibile in Figura 3.

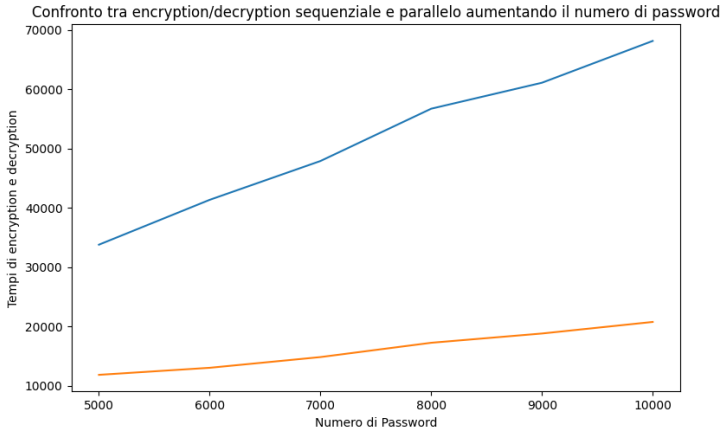

Figure 9. Confronto tra i tempi di cryptazione e decrypt di un numero crescente di password

Il test ha prodotto diversi speedup, visibili in Figura 4.

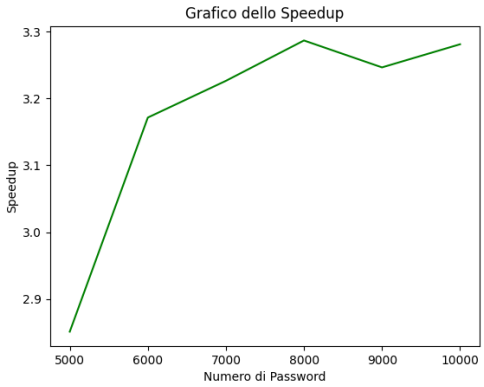

Figure 10. SpeedUp del test aumentando il numero di password

Si nota dal grafico che lo speedup aumenta con l'aumentare del numero di password

4. Conclusion