Password Decryption using DES Algorithm: Sequential vs Parallel version

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

Abstract

This paper will analyze the performance of an implementation of the Data Encryption Standard (DES) encryption algorithm in sequential version and two parallel versions

Future Distribution Permission

1. Introduction

In the modern world, cybersecurity is essential, as computer systems have become part of the daily life of almost every person on the planet. One of the first encryption algorithms was DES (Data Encryption Standard) of 1972, based on the Feistel Cipher. Over the years, DES has been improved, with Triple DES and AES (Advanced Encryption Standard) versions. In this project, we will see a sequential version of the DES algorithm implemented in C++ that will be compared with a parallel version implemented in C++ using the OPENMP library and another parallel version implemented in Python using the JOBLIB library.

2. Implementation

Before we look at the specific features of each implementation, let's examine the characteristics of the DES algorithm. To do this, we refer to Figure 1. The steps of encyption phase are:

- The plaintext is divided into two halves of W bits each (in our case W = 32), L_0 and R_0.
- These two halves pass through n rounds, each with the same structure, but using n sub-keys

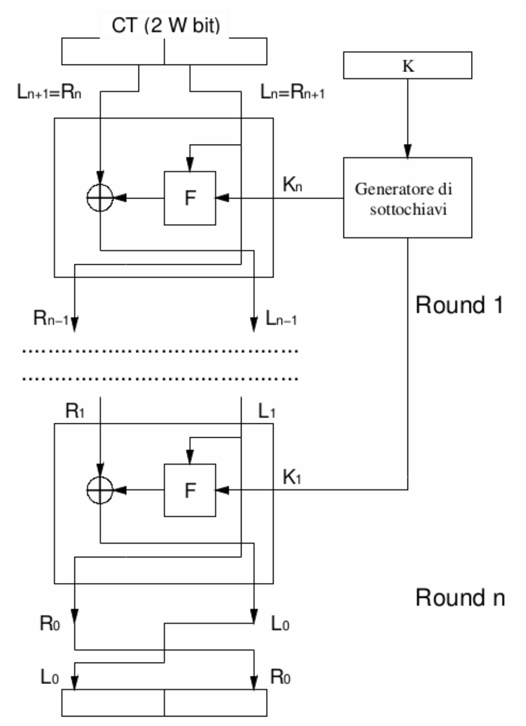

Figure 1. DES algorithm

K_1, ..., K_n, generated from the main key K. The sub-keys are all different from each other and different from K.

- The right half goes through the function F, which performs the substitution operation. It is then combined, via xor, with the left part: this becomes the right half of the output, while the left half will be formed by the input right half.
- The function F is the same for all rounds, and takes R_i and K_i as arguments.
- The two halves come out of the last round as

L_n and R_n, which, when combined, form the ciphertext.

To decrypt a ciphertext, simply pass it back through the algorithm, but be sure to use the n sub-keys in reverse order: K_n, K_{n-1}, ..., K_1. Now let's see how the function F is implemented. In the DES and AES ciphers, the function F uses the so-called Substitution-Box, or S-Box. An S-box is a table that associates each possible block of m input bits with a block of n output bits. In Figure 2, we see its implementation details.

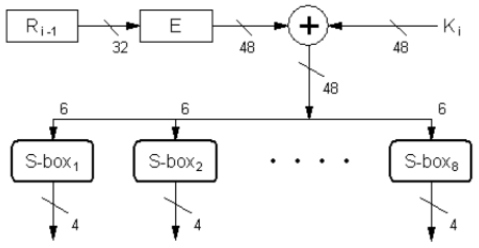

Figure 2. Implementation details of the F function

In the figure, R_{i-1} represents the input right half at the i-th round, while E is an expansion function that extends the bits from 32 to 48. The substitution involves a set of eight S-boxes, each of which accepts 6 bits as input and produces a 4-bit output. Each of them is a fixed 4×16 matrix: each of the 4 rows is a permutation of the integers between 0 and 31. The 6-bit input selects a row and a column, and the corresponding value, represented in binary, is the output. The row is selected by the binary number formed by the first and last of the 6 input bits, while the column is given by the central 4 bits. For example, for input 000111, the row is 01 (1 in decimal), so the second row is selected, while the column is 0011 (3 in decimal), corresponding to the fourth column.

2.1. C++ Sequential Version

In the sequential version of the algorithm we can find the following functions:

- *convertDecimalToBinary(int decimal)* and *convertBinaryToDecimal(string binary)* to be used for finding numbers in the S-Boxes and vice versa
- *xor(string a, string b)* that performs the xor operation between two binary strings
- *generateKeys()* that generates the K keys
- *tablesFiller()* that generates the values for the Expansion Table (used to expand the bits from 32 to 48) and for the Substitution Boxes
- *reverseKeys()* that reverses the order of the keys to alternate between encryption and decryption
- *DES(string plaintext)* that implements the actual algorithm
- *sequentialDecryption(vector[string] lines, int size)* which is the function that handles encryption and decryption of a vector of strings

The code for the function is reported below. *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

To implement the parallel version of the algorithm, the *OpenMP* library was used. The functions seen in the sequential version remain substantially the same, with the only change regarding the *generateKeys()*, *reverseKeys()*, and *DES(string plaintext)* functions. For these functions, it is necessary to pass a reference to *roundkeys* as an argument. This variable was a global variable in the sequential version, but it became a local variable in the parallel version. This is because, in the parallel version of the algorithm, it was decided to parallelize on the data, meaning that each thread takes an element of the password vector and performs encryption, decryption, and checks that they have been executed correctly (all of which is enclosed in the *pragma omp for* directive of *OpenMP*). To do this, each thread needs to have a private key vector on which to work because, as seen previously, the key vector must be inverted between encryption and decryption. The expansion table and the substitution boxes, on the other hand, can be shared among the threads because we only read data from them. The new function for handling encryption and decryption using *OpenMP* directives is now reported:

```
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

The sequential version in Python is the same as the sequential version implemented in C++. The code for the main function is reported below.

```
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

The other parallel version of the algorithm was implemented in Python using the Joblib library. It provides the *Parallel* function to write embarrassingly parallel code, allowing the use of the same function on multiple Python processes.

In this version, the functions are the same. However, to allow the use of the *Parallel* function of the Joblib library, the single encryption/decryption operation has been encapsulated in a function instead of being left in a for loop. In this way, it can be passed as an argument to the *Parallel* function as follows:

```
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
```

The *Parallel* function has as parameters the number of threads used, the function to parallelize, and the arguments that will be passed to that function. In this way, each Python process invoked will call the *single-en-dec* function using a password from the list as an argument. The code of the function is:

```
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

To evaluate performance, 2 types of tests were implemented for each version: the first where the number of threads used in the parallel implementation increases, while keeping the number of passwords fixed; the second where the number of passwords to be encrypted and decrypted increases.

3.1. Test 1 - OpenMP

For the first test, the first 5000 passwords and the C++ code version using OpenMP are considered. The various lines of the file are saved in a vector and a sequential encryption and decryption is performed for each password. This is repeated *nTest* times, and finally, the average values are calculated.

```
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;
```

Subsequently, tests were performed by increasing the number of threads. We go up to a maximum of 16 threads, and for each one, we perform *nTest*, calculating the execution times and taking the average. The speedup compared to the sequential version is also calculated.

```
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

In the second test a subset of passwords is considered each time, until reaching all 10000 passwords. For the sequential version of the test, 5000 passwords are initially considered, and the number is increased by 1000 each time while calculating the times

```
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;
}
```

For the parallel version of the test, we set the number of threads used to the maximum possible (4 in this case) using the function *omp_set_num_threads(maxThreads)*, and the same loop as the sequential test is executed: starting from 5000 passwords, it increases by 1000 each time while calculating the times.

```
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

For the first test, the first 5000 passwords and the Python code version using Joblib are considered. The various lines of the file are saved in a vector and a encryption and decryption is performed for each password sequentially, calculating the times.

```
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')
```

Next, tests are run by increasing the number of threads. It goes up to a maximum of 16 threads and for each one, *n-test* is executed, calculating the execution times and taking the average.

```
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

For the second test, instead, a subset of passwords is considered each time until reaching all 10000 passwords. For the sequential version of the test, starting from 5000 passwords, the number of passwords is increased by 1000 each time while calculating the times.

```
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')
```

For the parallel version of the test, the maximum number of threads is set (4 in this case) and the same cycle as the sequential test is executed: starting from 5000 passwords, the number of passwords is increased by 1000 each time, and the execution times are calculated.

```
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

The tests were performed on a PC with an AMD A8-6410 processor that has 4 cores and 4 threads. The file used for passwords is a txt file containing 10000 passwords with a length of 8 characters. The first 128 passwords are a collection of the most commonly used passwords in the world, while the rest were generated randomly.

3.6. Results OpenMP

The results of the first test with OpenMP, where the number of passwords is fixed at 5000 and the number of threads is gradually increased, are shown in Figure 3.

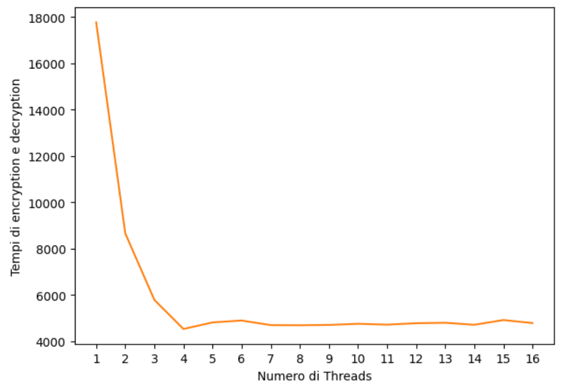

Figure 3. Comparison between loading times of 5000 passwords with OpenMP

The test produced various speedups, which are visible in Figure 4.
It can be seen from the graph that the speedup initially increases as the number of threads increases, until it reaches the maximum when there are 4 threads (which is the maximum number for the machine). Afterward, it decreases, oscillating between various values, always better than the initial ones but inferior to when the number of threads is 4.

The results of the second test, in which the sequential loading and loading with the number of threads set to the maximum (in this case 4) are compared by increasing the number of passwords, are visible in Figure 5.

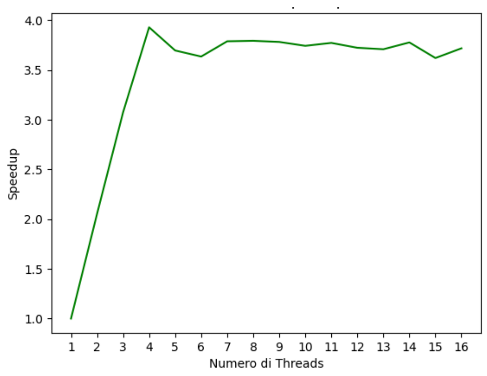

Figure 4. Speedup of the test with OpenMP increasing the number of threads

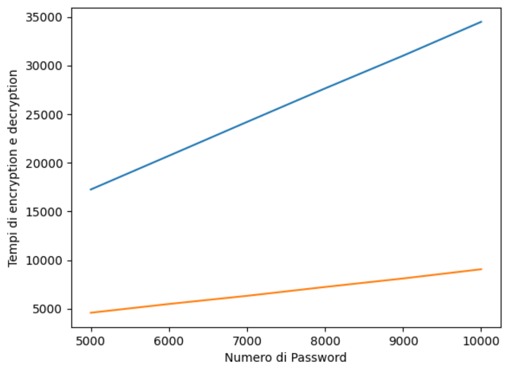

Figure 5. Comparison between the times for encryption and decryption of an increasing number of passwords with OpenMP

The test produced various speedups, visible in Figure 6.

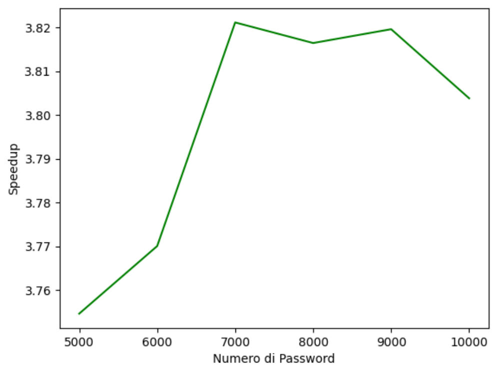

Figure 6. Speedup of the test by increasing the number of passwords

It can be noticed from the graph that the speedup does not have a precise trend but only varies between the values of 3.75 and 3.82.

3.7. Results Joblib

The results of the first test with Joblib, in which the number of passwords is fixed at 5000 and the number of threads is increased gradually, are visible in Figure 7.

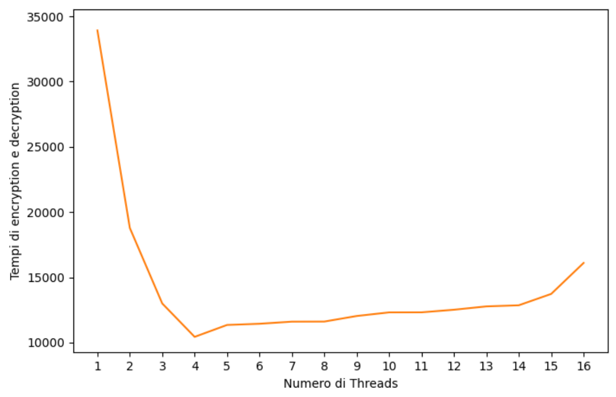

Figure 7. Comparison of encryption/decryption times of 5000 passwords using Joblib

The test produced several speedups that are visible in Figure 8.

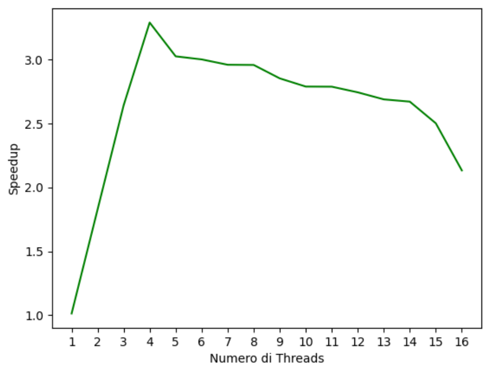

Figure 8. SpeedUp of the test with Joblib increasing the number of threads

It can be observed from the graph that the speedup initially increases as the number of threads increases, until it reaches its maximum when there are 4 threads (which is the maximum number for the machine). Subsequently, it decreases as the number of threads increases.

The results of the second test, in which the sequential loading and the loading with the maximum number of threads set (in this case 4) are compared by increasing the number of passwords, are visible in Figure 9.

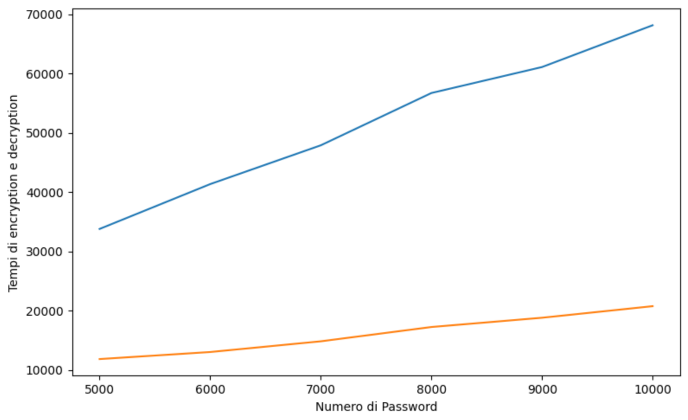

Figure 9. Comparison between the encryption and decryption times of an increasing number of passwords with Joblib

The test produced various speedups, visible in Figure 10.

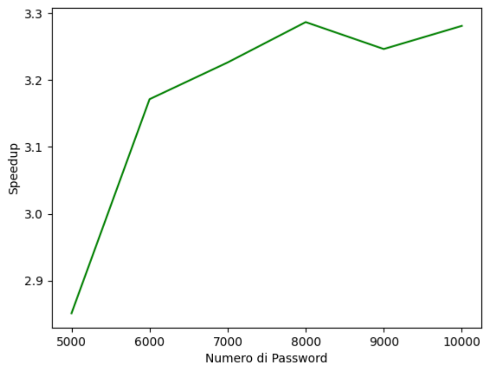

Figure 10. Speedup of the test increasing the number of passwords

It can be observed from the graph that the speedup increases with the increase in the number of passwords.

4. Conclusion

We have seen an implementation of the DES encryption algorithm in two different versions. From the results, we notice that the C++ version takes less time than the Python version, as can be seen from the comparison graph in Figure 11. Therefore, we can conclude that if the goal of an application is speed of execution, it is better to implement the C++ version using the OpenMP library.
The project is available at this link: `https://github.com/Luca0079CM/DES_Decryption`

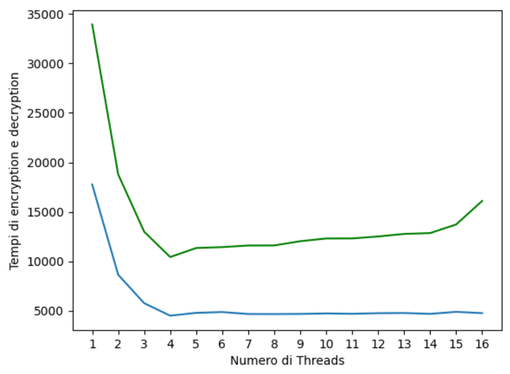

Figure 11. Comparison between the 2 versions on the decrypt of 5000 passwords