UNIVERSITÀ DEGLI STUDI FIRENZE

DES Decryption

Parallel Programming for Machine Learning

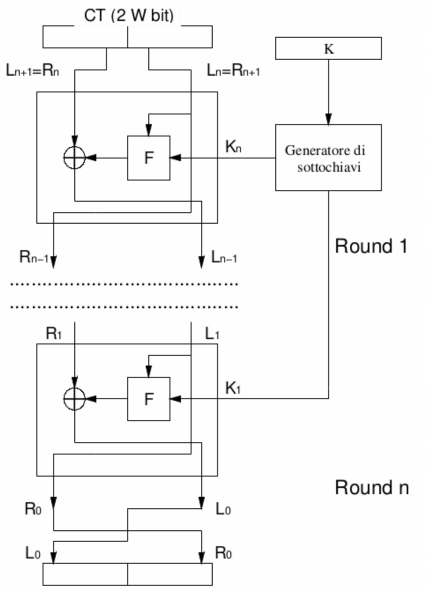
Luca Leuter

Introduction

One of the first encryption algorithms was DES (Data Encryption Standard) of 1972, based on the Feistel Cipher. 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.

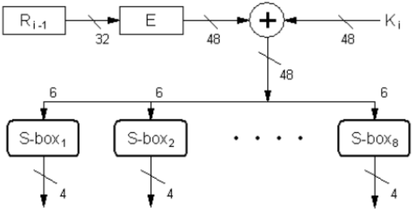
UNIVERSITÀ DEGLI STUDI FIRENZE

DES details



CT (2 W bit)
K
Ln+1=Rn
Ln=Rn+1
F
Kn
Generatore di sottochiavi
Rn−1
Ln−1
Round 1
R1
L1
F
K1
Round n
R0
L0
L0
R0

DES details



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

C++ Sequential Version

- *sequentialDecryption(vector[string] lines, int size)* which is the function that handles encryption and decryption of a vector of strings

```
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;
```

C++ Parallel Version

```
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";
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Python Sequential Version

```
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
```

Python Parallel Version

```
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
```

```
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
```

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.

Tests

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.

Test 1 - OpenMP

```
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;
```

Test 1 - OpenMP

```
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;
    }
```

Test 2 - OpenMP

```
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;
}
```

Test 2 - OpenMP

```
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;
}
```

Test 1 - Joblib

```
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')
```

Test 1 - Joblib

```
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')
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Test 2 - Joblib

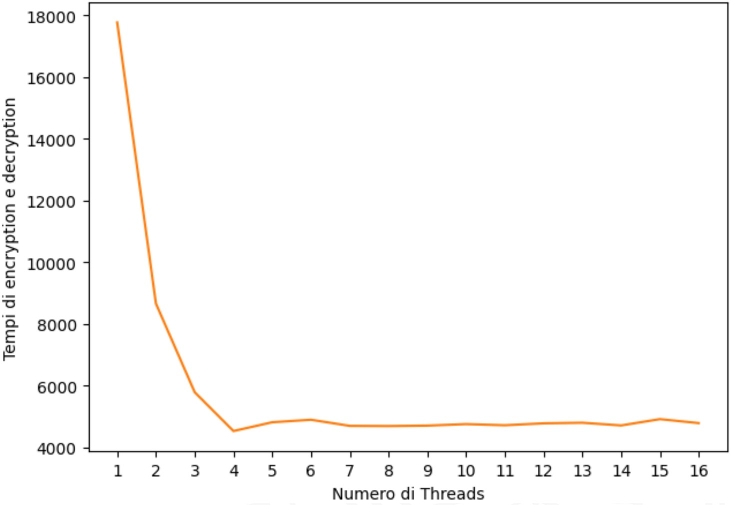
```
for i in range(5000, 11000, 1000):
    start_time = time.time()
    encrypt_and_decrypt_sequential(Lines[0:i])
    end_time = time.time()
    print(f'Tempo per decript sequenziale
    di {i}
        password:{end_time-start_time:}s')
```

Test 2 - Joblib

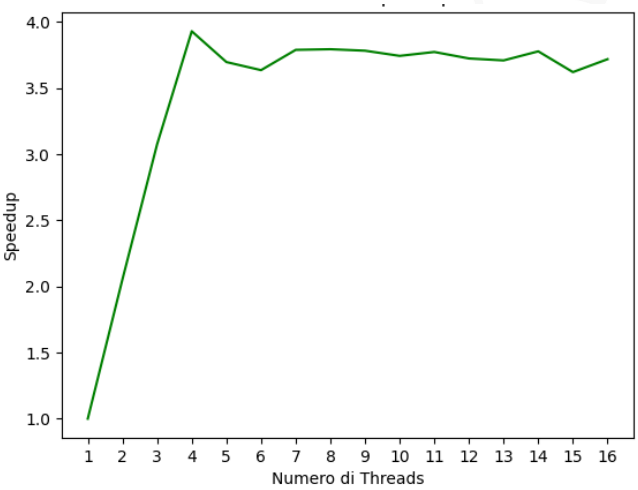
```
for i in range(5000, 11000, 1000):
    start_time = time.time()
    encrypt_and_decrypt_parallel(Lines[0:i],
        4)
    end_time = time.time()
    print(f'Tempo per decriptazione
        parallela con 4 threads di {i}
        password: {end_time -
        start_time:} s')
```

UNIVERSITÀ DEGLI STUDI FIRENZE

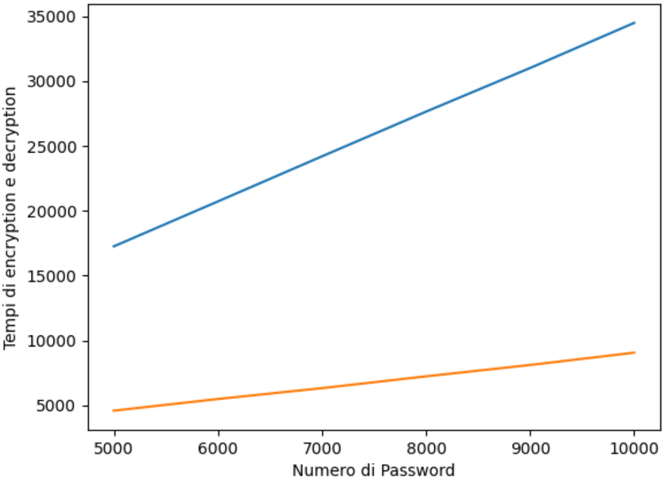
Results of test 1 with OpenMP



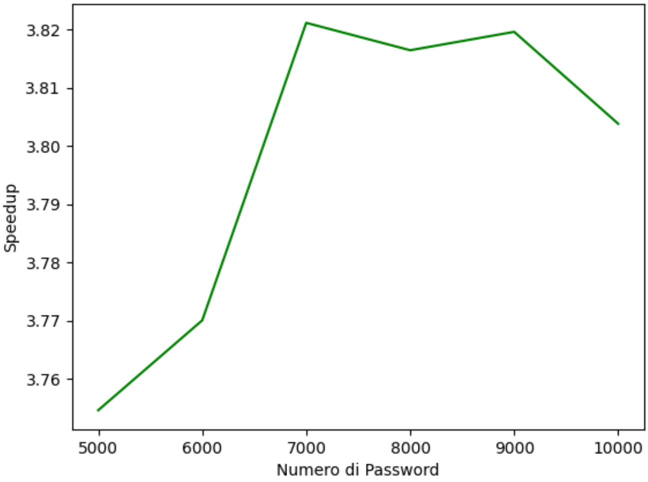
Results of test 1 with OpenMP



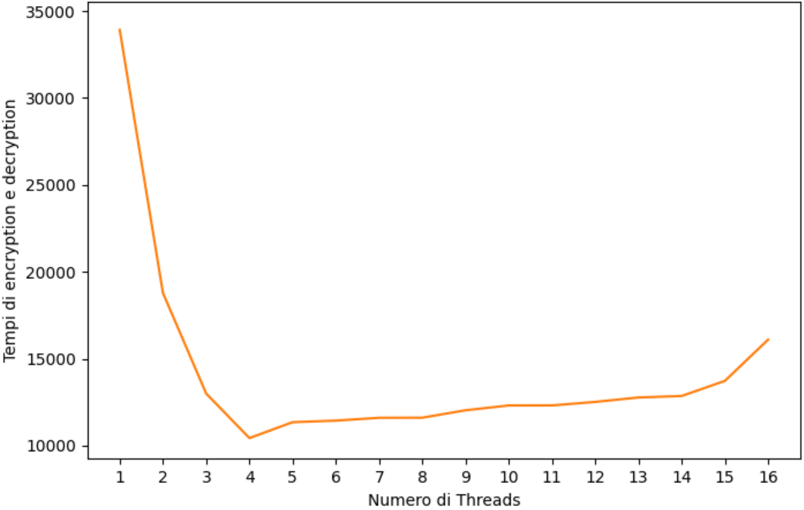
Results of test 2 with OpenMP



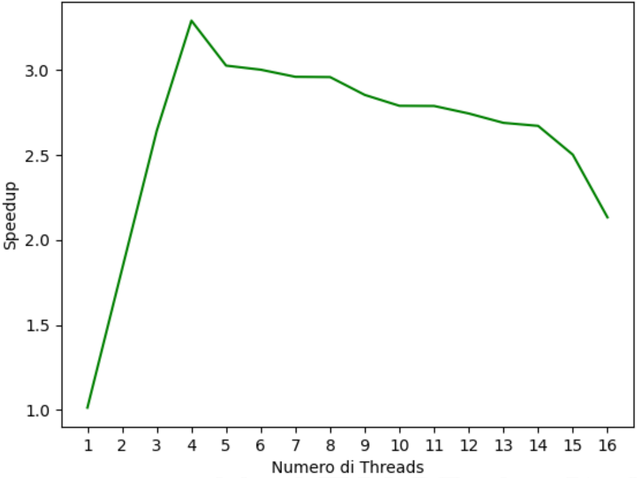
Results of test 2 with OpenMP



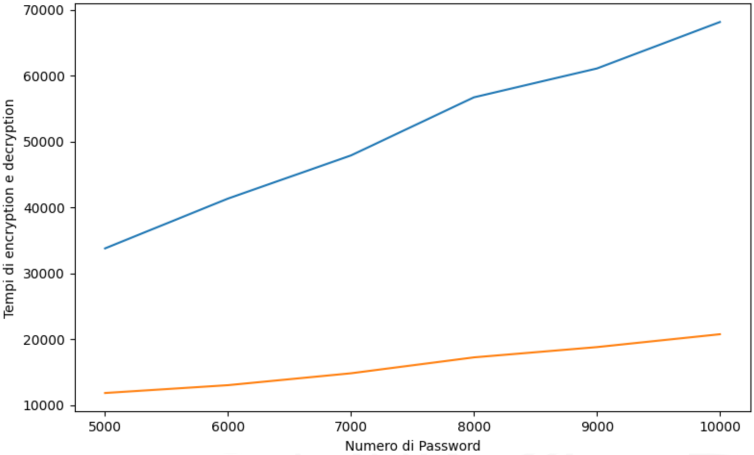
Results of test 1 with Joblib



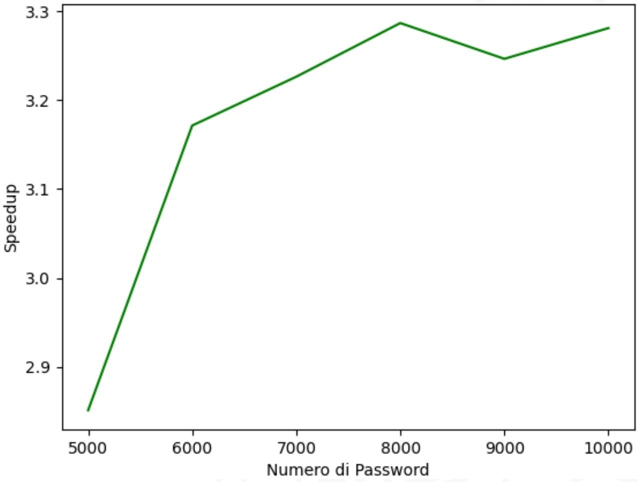
Results of test 1 with Joblib



Results of test 2 with Joblib



Results of test 2 with Joblib



Conclusion

From the results, we notice that the C++ version takes less time than the Python version, as can be seen from the comparison graph. 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.