Comparison between sequential and parallel programming on Data Encryption Standard (DES) with OpenMP

Francesco Todino
E-mail address
francesco.todino@edu.unifi.it

Abstract

The aim of this paper is to compare the performance of sequential and parallel Data Encryption Standard (DES) algorithms using the OpenMP library.

1. Introduction

In this paper, we will analyze the performances obtained by sequential and parallel implementations of simulating a brute force attack of encrypted passwords with Data Encryption Standard (DES) algorithms. Our main goal is to understand the capabilities of the OpenMP library by comparing it with sequential algorithms. We will cover different tests, including benchmarking the execution time, measuring speedup and efficiency, and examining scalability across different input sizes.

1.1. Data Encryption Standard (DES)

To perform these tests we must first take a look at the theory behind DES. DES is a symmetric-key block cipher that was developed by IBM in the early 1970s and later adopted by the U.S. National Institute of Standards and Technology (NIST) as a federal standard in 1976. DES is widely known for its historical significance in the field of cryptography, although its security is considered outdated by modern standards.

As shown in Figure 1, the encryption process begins with an Initial Permutation (IP) of the 64-bit plaintext, reordering its bits according to a predefined permutation table. DES operates in a Feistel network structure, dividing the plaintext into two halves, labeled as L and R. In each of the 16 rounds of encryption, the right half (R) undergoes an expansion to 48 bits. This expanded half is then XORed with the round key, and the result passes through a series of S-boxes, performing a nonlinear substitution on the bits.

The output of the S-boxes undergoes further permutation, and the result is XORed with the left half (L). The left and right halves are then swapped, and the process repeats for each subsequent round. After 16 rounds, the left and right halves are combined, and a Final Permutation (FP) is applied, which is the inverse of the initial permutation. This produces the final ciphertext, ready for ”secure” transmission or storage.

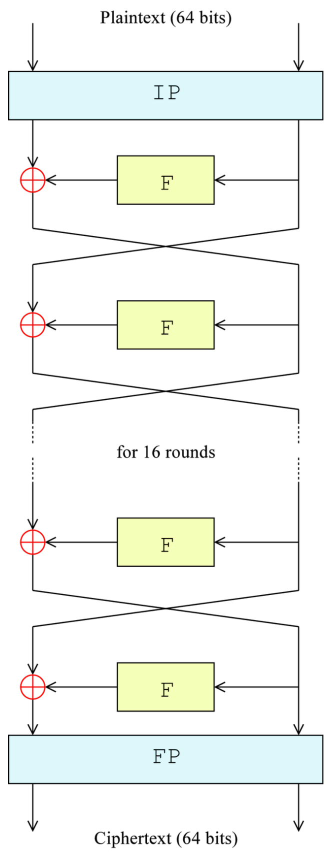

Figure 1. The overall Feistel structure of DES.

For decryption, the process is reversed. The ciphertext undergoes an initial permutation, and the same key schedule is applied in reverse order. The Feistel network structure is used again, but with the subkeys applied in the reverse sequence.

1.2. OpenMP

OpenMP, or Open Multi-Processing, is an API (Application Programming Interface) designed to facilitate parallel programming on shared-memory architectures. It supports C, C++, and Fortran languages and provides a set of compiler directives and library routines for parallel execution.

Developers use pragma directives to annotate code sections that should run concurrently. This shared-memory model simplifies parallel programming, abstracting complexities associated with thread management and data sharing.

1.3. Setup

All tests were developed in C++ and performed on a MacBook Pro (mid 2015) with an Intel Core i7-4770HQ with 4 cores and 8 threads. All plaintexts have a length of 8 and are composed of the characters included in [a-zA-Z0-9./].

2. Methods

In this section we will see all the code involved into this comparison. The objective was to find one or more encrypted passwords in a list of unencrypted passwords. To do this, we will follow these steps:

- We pick one or more plaintexts from a txt file containing a certain amount of them (varying from 10k to 1mln). We have created an utility in python to generate this file containing N random texts of a specific M length composed of the characters included in [a-zA-Z0-9./];
- One by one, we crypt all the selected plaintexts and then we proceed to iterate all the elements contained inside the file to find which one has the same ciphertext as them;
- We stop the iteration as soon as we find the same ciphertext, moving on to the next selected plaintext.

It is obvious that this can be a very demanding task for a sequential program, especially due to the possibility of selecting passwords at the end of the file.

In Listing 1 we can see how we collect the plaintexts:

```
string line;
ifstream file("text_gen/words.txt");
vector<string> pwdList = {};
while (getline(file, line))
{
   pwdList.push_back(line);
}
file.close();
vector<string> pwdToCrack = {};
while (pwdToCrack.size() < nToCrack)
{
   auto newEl = pwdList[rand() %
       pwdList.size()];
   if (find(pwdToCrack.begin(),
       pwdToCrack.end(), newEl) ==
       pwdToCrack.end())
      pwdToCrack.push_back(newEl);
}

testCrack(pwdList, pwdToCrack);
```

Listing 1. Main content

To execute this analysis, we create the class DESAlgorithm that will be shared by both solutions, containing these methods:

- *generateKeys()*: generates the 16 keys for every round in DES.
- *reverseKeys()*: reverses the elements contained in roundKeys.
- *defineTables()*: utility to generate a random expansionTable and all the S-boxes involved. We use this method in the constructor to generate a new config each time.
- *XOR(string, string)*: performs a xor between the two binary inputs received.
- *DES(string)*: executes the DES algorithm, receiving as input the plaintext or the cyphertext and returning the related output.

We also have some utils to convert *binary* to *decimal*, *binary* to *alphanumerical* and viceversa.

2.1. Sequential implementation

As mentioned before, we iterate through all the *pwdToCrack* and then through all the passwords available. As soon as we find the same ciphertext, we stop and move on to the next password.

```
DESAlgorithm des;

for (string &encrypted : pwdToCrack)
{
  encrypted =
      des.DES(des.stringToBin(encrypted));
   for (string &pwd : pwdList)
   {
      string pwdEncrypted =
          des.DES(des.stringToBin(pwd));
      if (encrypted == pwdEncrypted)
      {
            cout << "Password found: " << pwd
                << endl;
            break;
        }
    }
}
```

Listing 2. Sequential implementation

2.2. Parallel implementation

In Listing 3 we can see the parallel implementation. At the beggining of the script, we set the number of threads that we are going to use with *omp_set_num_threads(nThreads)*. Then, we split the pwdList in equal parts, giving each part to each thread involved. In order to split the items correctly in case the division between threads has some rest, we decided to give the last thread its items plus the rest.

Regarding OpenMP, we use the directive *pragma omp parallel* to define the section of code where threads start to work simultaneously. Then, we assign each block to each thread based on its *thread_id* and use the *pragma omp cancel parallel* directive to stop parallel execution once the password is found.

It is important to set **OMP_CANCELLATION=true** within your computer's environment variables to enable cancellation directives otherwise threads will always run to the end of the list, achieving worse performance than the sequential version.

```
omp_set_num_threads(nThreads);
DESAlgorithm des;

for (string &encrypted : pwdToCrack)
{
    encrypted =
        des.DES(des.stringToBin(encrypted));
    vector<int> chunks;
    chunks.push_back(0);
    for (int i = 1; i < nThreads; i++)
    {
        chunks.push_back(
        static_cast<int>(pwdList.size() /
            nThreads) * i
        );
    }
    chunks.push_back(pwdList.size());

#pragma omp parallel shared(des,
    encrypted, chunks)
    {
        int tid = omp_get_thread_num();
        for (int i = chunks[tid]; i <
            chunks[tid + 1]; i++)
        {
            string pwdEncrypted =
                des.DES(des.stringToBin(
                    pwdList[i]
                    ));

                    if (encrypted == pwdEncrypted)
                    {
                        cout << "Password found: " <<
                            pwdList[i] << endl;
#pragma omp cancel parallel
                    }
#pragma omp cancellation point parallel
        }
    }
}
```

Listing 3. Parallel implementation

3. Tests

The tests done to evaluate the performance are the one suggested by the goal:

- In the first test we compared the performance by increasing the number of threads to solve a one password crack;
- In the second test we compared the performance by increasing the number of passwords searched, using the parallel version with increasing threads.

Both tests are performed with 100.000 available passwords ($\sim$ 900 kB). To avoid biased results, based on the positioning of the words selected, we tested everything more times and get the average results.

3.1. Fixed passwords number

The results of the first test, where the number of passwords is fixed to 1 and the number of threads is increased up to 64, are shown in Figure 2.

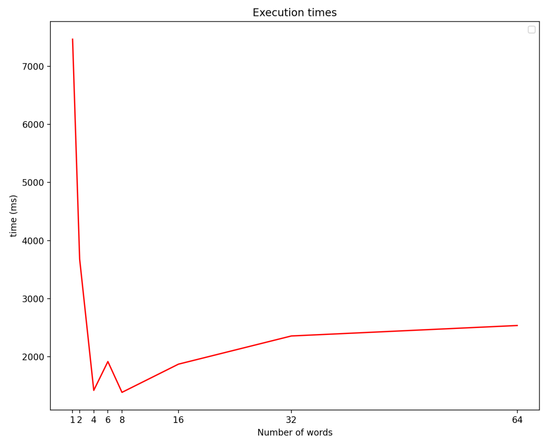

Figure 2. Times solving 1 password, increasing threads.

As expected, the time decreases from 7465 ms in the sequential version down to 1389 ms in the parallel version with 8 threads (expected as best performance in this setup), then increasing linearly as the number of threads increases.

Then we can see related speedups shown in Figure 3.

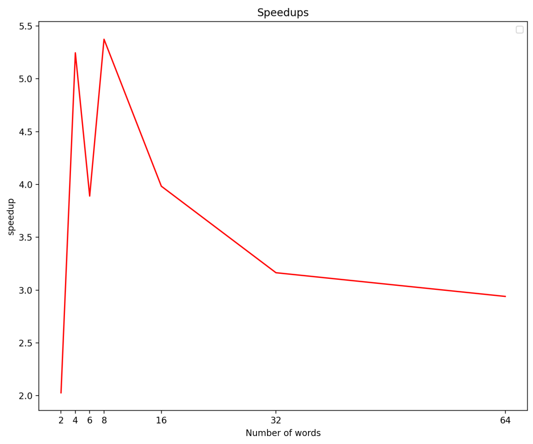

Figure 3. Speedups solving 1 password, increasing threads.

Going by the times, the speedups achieved are promising, having achieved speedup up to 5.37437x on 8 threads.

3.2. Fixed threads number

The results of the second test, where the number of threads increases and the number of passwords searched is increased up to 20, are shown in Figure 4.

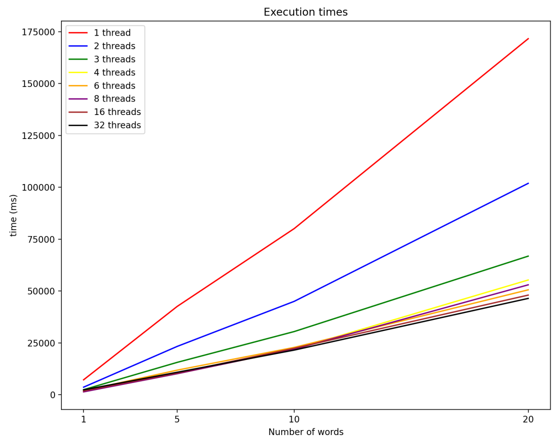

Figure 4. Times solving N passwords, increasing threads.

As expected, the time increases from 7201 ms to 171658 ms in the sequential version while in the 8-thread parallel version, as example, it starts from 1395 ms up to 52993 ms, confirming the performance increase.

It can be seen from the Figure 4 that the best performance is obtained with 32 threads: we think that this is because increasing the number of threads increases the probability of finding the correct password in the first iterations.

Then we can see related speedups shown in Figure 5.

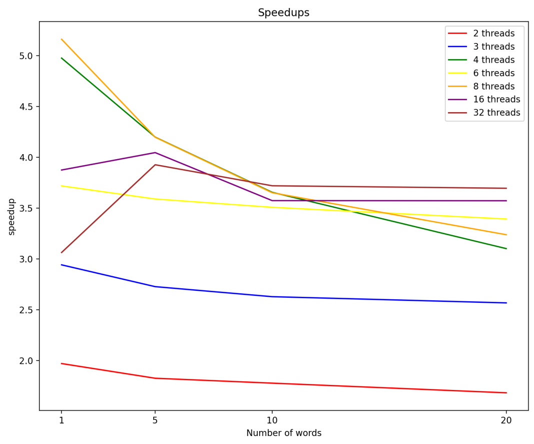

Figure 5. Speedups solving N passwords, with 8 threads.

In this test we expected speedups to improve over time but this does not happen. As said before, it may be due to the randomness of password positions not always favoring the parallel version. To confim that, the increase in threads led to increased performance in multi-threaded tests due to the fact that the set of words is split into multiple sets, increasing the probability of finding the searched item in the first few iterations.

4. Conclusion

In conclusion, the comparison between sequential and parallel programming in the context of the Data Encryption Standard (DES) using OpenMP confirms that the best approach is generally the parallel one, having obtained increasingly better performance in all tests carried out.