DES decryption with OpenMP

Pietro Zarri
pietro.zarri@stud.unifi.it

Abstract

The aim of this project is to implement a simple version of a decryption algorithm, having password hash value generated through the DES (Data Encryption Standard) algorithm. The search is executed within a dictionary of 8-characters passwords. The main focus is on measuring how a parallel approach with OpenMP library can improve the performance of the execution.

Future Distribution Permission

1. Introduction

The problem starts with a password's hash value generated by the DES algorithm. Using a 100.000 items dictionary, every word is encrypted and compared with the start value.
To improve the performance is implemented a parallel strategy with the OpenMP library. Sequential and parallel approach are compared in term of speedup, changing the number of threads in the parallelization process.

2. Decryption

2.1. DES algorithm

The **Data Encryption Standard (DES)**[4] is a symmetric-key algorithm for the encryption of digital-data. It was developed in 1970 by IBM.
The classic algorithm aims to crypt a 64 bits block data with a 64 bits key of which only 56 bits are used (the other 8 bits may be used for error detection). Generally [a-zA-Z0-9./] is supposed as the key characters set. A block, to be enciphered, is subjected to an initial permutation, then to a complex key-dependent computation and finally to a permutation which is inverse of the initial one. Actually DES algorithm is considered unsecure due to its relatively short key size.

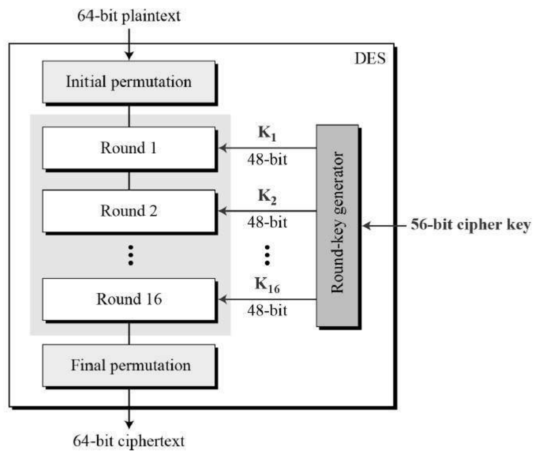


Figure 1. DES algorithm scheme

2.2. Dictionary

The dictionary used for this experiment is a subset obtained from a bigger one, after retrieving, with a python script all 8-characters words[3]. All characters are in the [a-zA-Z0-9./] set.

2.3. Words to test

In order to obtain a more accurate result in term of performance improvement and speed up all types of decryption is executed in four different cases. The words used are the first one *"password"*, the last one *"63286328"*, *"kanukanu"* which is approximately at $\frac{1}{3}$ of the dictionary and *"boston44"* at $\frac{2}{3}$.

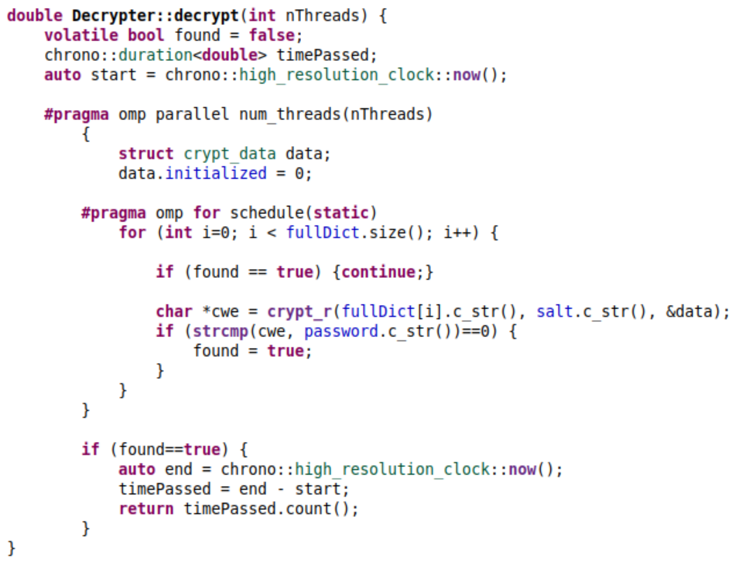
3. Implementation

The implementation of the dictionary attack is based on the *Decrypter* class. The constructor receives the path of the dictionary file (*.txt*), and a 2-characther string named *salt* used for the word encryption[1]. The *.txt* file is processed and for facility of use all the words are inserted into a string vector. Within the class there is also a method which recives the hash value of the word to find in the dictionary and store it in the class ad private attribute. The main features are implemented in two methods:

- **decrypt**: implements the sequential version of the decryption algorithm.
- **decryptSeq**: implement the parallel version of the algorithm using the OpenMP library.

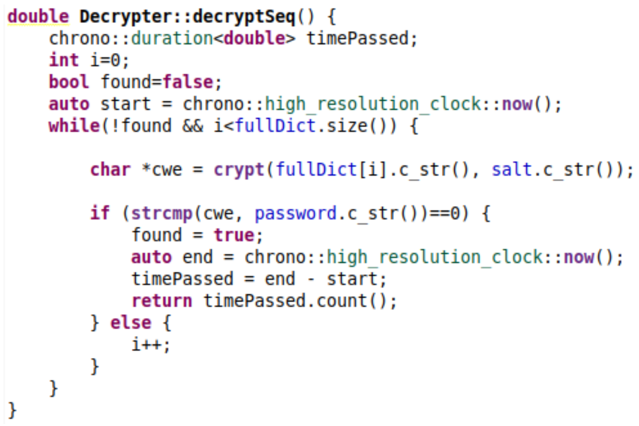
Basically the routine of the two methods is the same and for clearness below is reported the code:

Its executed a loop on the variable `fullDict` and from every word is calculated its has value. Then this is compare with the initial value, if this happens the value ”found” is set to true and the flow exits from the loop.
To have a measure of the improvement with a parallel approach is used the `chrono` class. Thanks to this is possible to measure the time occured to find the word in the dictionary correspondent to the initial hash value.



```
double Decrypter::decrypt(int nThreads) {
    volatile bool found = false;
    chrono::duration<double> timePassed;
    auto start = chrono::high_resolution_clock::now();

    #pragma omp parallel num_threads(nThreads)
        {
            struct crypt_data data;
            data.initialized = 0;

        #pragma omp for schedule(static)
            for (int i=0; i < fullDict.size(); i++) {

                if (found == true) {continue;}

                char *cwe = crypt_r(fullDict[i].c_str(), salt.c_str(), &data);
                if (strcmp(cwe, password.c_str())==0) {
                    found = true;
                }
            }
        }

        if (found==true) {
            auto end = chrono::high_resolution_clock::now();
            timePassed = end - start;
            return timePassed.count();
        }
}
```

Figure 2. Algorithm Parallelization with OpenMP



```
double Decrypter::decryptSeq() {
    chrono::duration<double> timePassed;
    int i=0;
    bool found=false;
    auto start = chrono::high_resolution_clock::now();
    while(!found && i<fullDict.size()) {

        char *cwe = crypt(fullDict[i].c_str(), salt.c_str());

        if (strcmp(cwe, password.c_str())==0) {
            found = true;
            auto end = chrono::high_resolution_clock::now();
            timePassed = end - start;
            return timePassed.count();
        } else {
            i++;
        }
    }
}
```

Figure 3. Sequential Algorithm

4. Parallel Implementation

To implement the parallelization of the execution are used the PRAGMA directives of OpenMP[2]. They're preprocessor directives and they're declared with a hash.
`PRAGMA omp parallel num_thread()` creates a parallel region. When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code. There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point. *nThreads* specifies the number of threads.
`PRAGMA omp for schedule(static)` divides the execution of the *for* loop among all the threads that have been created. Work-sharing constructs like this do not launch new threads. There are five different loop scheduling clauses. They describes how iterations of the loop are divided among the threads in the team. In this case *static* clause is used: loop iterations are divided into blocks of size chunk and then statically assigned to threads.
In order to have consistency during the reading by threads, the *found* variable is marked with ***volatile*** keyword.

5. Experiments and results

All the tests are executed on a computer with the following specifications:

- Intel Core i5-2450M @2.50GHz Quad-Core.
- 8GB RAM.
- NVIDIA GeForce GT 630M.
- Ubuntu 20.04 LTS.

Every experiment is run ten times and then is calculated the arithmetic mean between all execution times.
At this stage particular emphasis was given to the speedup concept to figure out if the taken multi-threading approach brought benefits. The speedup Sp is defined as $Sp = ts/tp$ where, p is the number of processors, ts is the completion time of the sequential algorithm and tp is the completion time of the parallel algorithm. The results of comparison between the sequential and the parallel approach with different number of threads are shown in the following table and graph:

Threads	Time(s)	Speedup
1	0.6394	1
2	0.3437	1.71
4	0.2422	2.63
8	0.2301	2.77
16	0.2323	2.75

Table 1. Password matching with ”boston44” password

The tables 1 shows that an optimal speedup is reached with a number of threads equal to 8. It is reasonable due to the Hyper-Threading technology that makes possible to host resources of two threads in a single core.
In the following graph is shown the speedup values with four different words as explained in 2.3.

Observing Figure 4 is clear that the parallelization of the code is more efficient when the words are at the bottom of the dictionary. In particular we can see that in the case of ”password” word using OpenMP directives is counterproductive. The cause of this can be attributed to threads overhead.

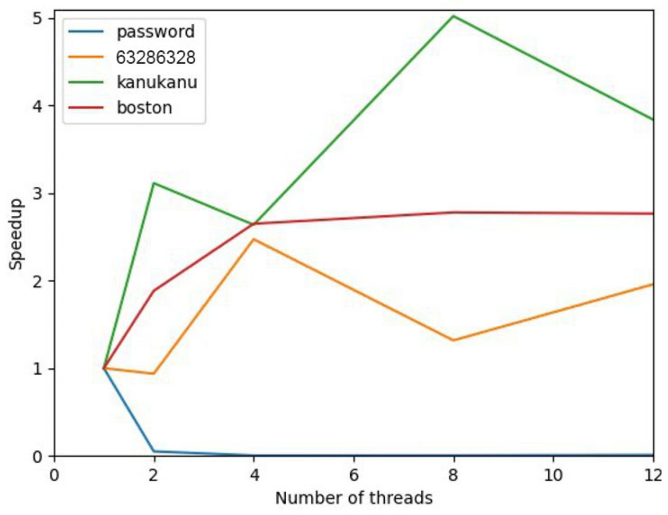


Figure 4. Comparison between speedups

We can also see that the speed up assumes an irregular trend with the other words. In fact part of the result depends also on the position that the target words has in his *chuck*. The split in chunks is done automatically by OpenMP.

5.1. Conclusion

How its obvious the parallel approach allows an increase of the performance especially when the word to find is farther from the start.
The OpenMP library provides an easy implementation of a parallel region and the framework automatically calculates each starting index for a chunk division. On the other side this behaviour doesn't allows the programmer flexibility on how to split tasks between threads.

References

[1] G. C. L. Manual”. Encrypting passwords. https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_650.html.

[2] ”OpenMP”. https://www.openmp.org.

[3] ”SecList”. https://github.com/danielmiessler/SecLists/tree/master/Passwords.

[4] ”Wikipedia”. Data encryption standard. https://it.wikipedia.org/wiki/Data_Encryption_Standard.