PC-2019/20 Mid Term Project: Password Decryption DES, OpenMP

Marioemanuele Ghianni
E-mail address
marioemanuele.ghianni@stud.unifi.it

Abstract

This mid-term project is part of the exam of the course Parallel Computing at University of Florence.
The aim of this project is to introduce and analyze a sequential and parallel version of a decryption program that retrieve a plain text password crypted with DES algorithm. The project is developed in C++ and OpenMP is used to implement the parallel version.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

The problem is to find a password hash generated by **DES**; The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of electronic data.
DES is the archetypal block cipher—an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt.
Without going into too much detail, [3] DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General Structure of DES is depicted in the following illustration -

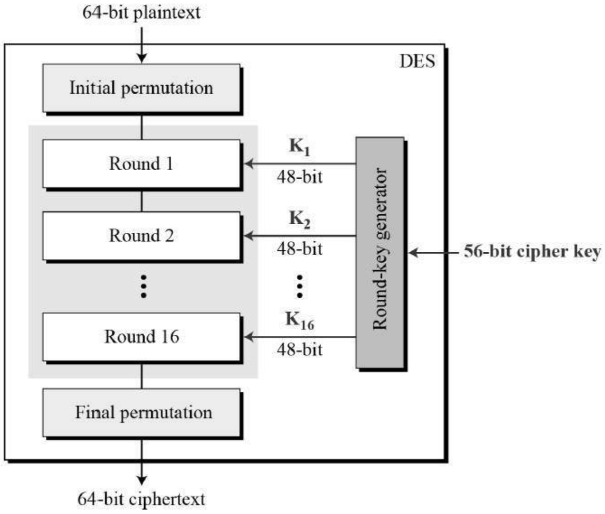

Figure 1. overall structure of des encryption.

Assuming to know the salt that generate the key and symplifying the problem taking into account only 8 characters passwords from the set (a-z,A-Z,0-9,./), we can perform a bruteforce attack dictionary based to retrieve the plain text password.
About the dictionary,it's derived through a python script from the **rockyou** dataset (see references [2]); a dataset that contains all the password hacked in 2009 from a site named Rock You.
The result is a custom dataset of **2750975** common passwords.

2. Structure

Summing up, the structure of the proposed algorithm is a sort of search and compare that:

- read the dictionary and store passwords into a vector.
- take one item at a time from vector and encrypt

it using an hypotetical known salt

- compare the encrypted element with a target password encrypted
- iterate until a match is found

2.1. Classes

In this project, there is a single important class called **Decrypter** that contains all the relevant functions.
Such as an initialization that read and store the dictionary, the sequential and parallel version of the bruteforce and a function to calculate speedups.
There are an initialization that read and store the dictionary, the sequential and parallel function of the bruteforce and a function to calculate speedups.
On top of the file .cpp there is an openMP import necessary to parallelize the code through simple compiler directives.

2.2. Main

The main file simply set the parameters to pass, istantiate a Decrypter object, invoke on him the decryption functions and retrieve the execution times.
It sets the number of runs per thread, the passwords to find and the number of threads to perform the parallel bruteforce ”attack”.

3. Sequential decryption

The sequential algorithm takes in input the number of runs and returns a vector of execution times.
Basically, for each target password encrypted there is a for-cycle that takes passwords from the vector that stores the dictionary, encrypts and compares them to the target.
When a correspondence is found, the cycle breaks and stops a chrono started at the beginning of the run cycle.
Finally, the algorithm stores the average of times of the runs for each target password and return them to the main.

4. OpenMP

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures [1].
The “MP” in OpenMP stands for “multiprocessing,” a term that is synonymous with shared-memory parallel computing.
It’s an implicit threading frameworks and libraries that take care of much of the minutiae needed to create, manage, and synchronize threads.
No need to restructure the serial program but only need to define directives to instruct the compiler on how to parallelize the code.

5. Parallel decryption

The overall structure of the parallel version is very similar to the sequential one except for the openMP directives and the partition of the dictionary vector in **workloads** to assign to the various threads.
In fact, the parallel version takes in input the number of threads and, through a parallel section of openMP and the function **omp_get_thread_num()** equally divides the dictionary password to be processed ($workload = dictionarySize/nThreads$).
The parallel section shares the dictionary vector and a volatile bool variable **found** that tells the other threads to stop cycling and break when assigned to true.
In fact, when there’s a correspondence between the encrypted word and target, found is assigned to true and the if-control redirects the other threads to the else-break section.
Below a significant extract of the code.

```
#pragma omp parallel default(none) num_threads(
    nThreads) shared(found, workloadThread,
    pswToCrack, dictPSW)
{
    int threadID = omp_get_thread_num();
    struct crypt_data data;
    data.initialized = 0;
    for (int pswID = threadID * workloadThread;
        pswID < (threadID + 1) * workloadThread;
        pswID++) {
        if (pswID<dictPSW.size() &&!found) {
            char *pswEncrypted = crypt_r(dictPSW[
                pswID].c_str(), saltpsw.c_str(),
                &data);
            if (pswToCrack ==string(pswEncrypted)
                ) {
                //found flag assigned to true to
                    redirect other threads in the
                     else-break section
                found = true;
                break;
            }
        }else {
            break;
        }
    }
}
```

6. Tests and comparison

6.1. Target passwords

Choosing the right passwords to test is an important step; the first password of the dictionary (**warbec14**)and the last one (**sauske00**) are chosen, to start.
Furthermore, to give a more practical measure of the speedup, it was also measured on 100 random words extracted from the dictionary using the shuf linux command.

6.2. Threads

Another important parameter to test is the speedup varying the number of threads and that can be very relevant because it define the actual number of cores exploited and the size of the workloads/chunks.
So, the various tests are performed varying the number of threads from 2 to 32 with a gradual increase.

6.3. Test configuration

The tests were performed on a Ubuntu virtual machine with 2 physical core and 1 virtual core allocated on top of an Intel i57200U clocked at 2.50GHz.

6.4. First password: top of the dictionary

As we can easly expect, decrypting the first password using a parallel method is absolutely much slower compared to the sequential one (see figure 3 below for detailed speedups).
There is a massive overhead in the parallel method when creates threads, initializes them and assigns them a workload while the sequential method only needs one cycle iteration .
So, generally speaking, a password wich belongs to the first elements of the dictionary it is faster to decrypt with a sequential method.

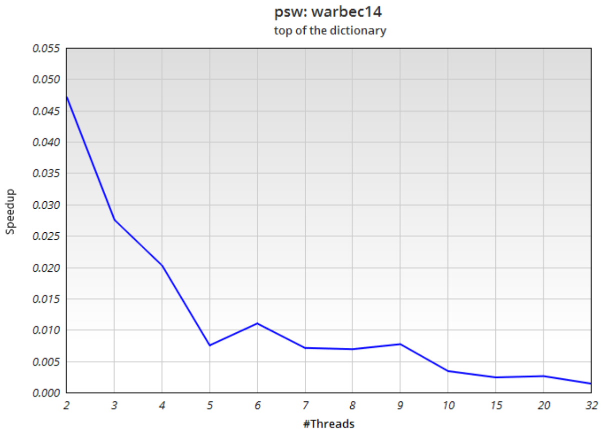

Figure 2. Speedup graph of the password warbec14.

6.5. Second password: end of the dictionary

Much more interesting from the point of view of parallelism, is the word placed at the end of the dictionary.
As we can see from the graph below, the speedup increases approximately to the number of threads that can be managed in parallel and then it remains stationary as the threads change for various iterations.

It does not exactly reach 3 due to the virtual core and the fact that the program runs on a virtual machine.
Concluding, in the case of a word at the bottom of the dictionary, the parallel approach is no doubt better but we cannot consider this as a general case.

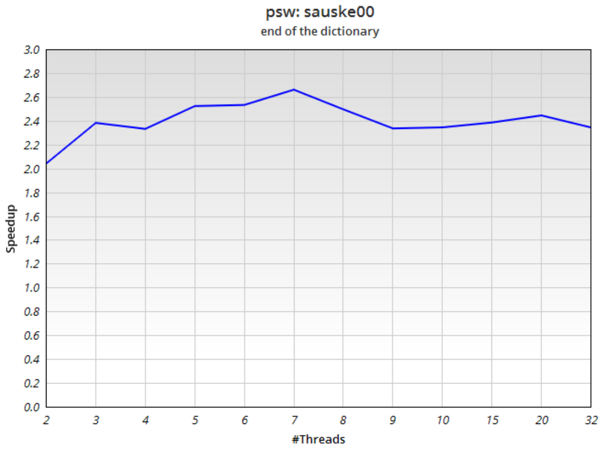

Figure 3. Speedup graph of the password sauske00.

6.6. Third test: 100 random words

This is the most significant test and that allows us to better appreciate the benefits of parallelism.
As previously said, 100 random words are extracted from the dictionary with the shuf command that generates random permutations from input lines to standard output.
Then, the average and the median of the speedups values per number of threads are calculated.
It was chosen to use the median in addition to the average because, in the lucky case where the word is positioned on top of the chunk, the speedup value becomes very large and largely determines the relative value of the average.
This is endeed visible in the graph below relative to the average (Figure 4) where the speedups values are very variable with even very high peaks.
On the other hand, by using the median the outliers are excluded and a more realistic measure of the speedup is obtained as we can see in the second graph (Figure 6).
In this case, we can conclude that the sequential approach is not the best choice since, however, on average there are speedups much greater than 1 with a parallel approach.

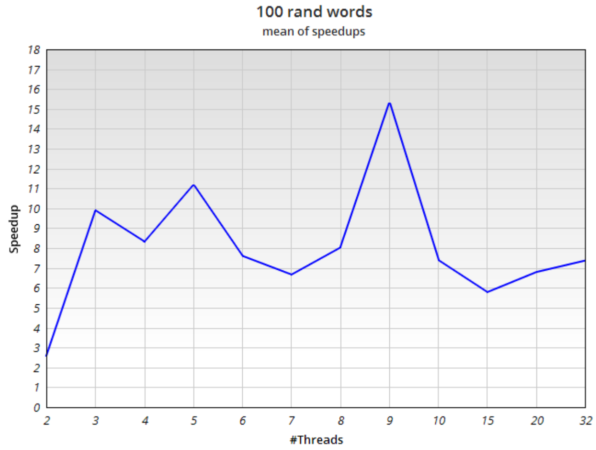

Figure 4. Speedups mean graph based on 100 random words.

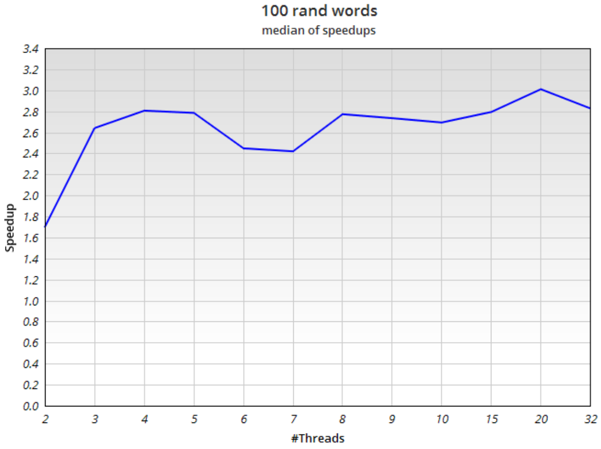

Figure 5. Speedups median graph based on 100 random words.

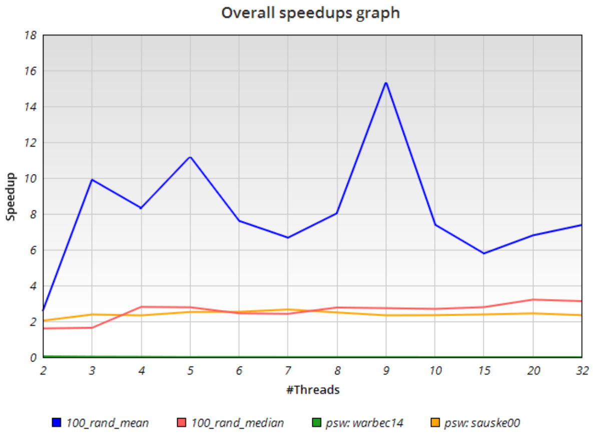

Figure 6. Speedups summary chart.

7. Conclusions

Concluding, we can say, as was predictable, that in this problem the best approach is the parallel approach and it is even more so with a substantial number of cores available because, of course, the speedup values obtained are however determined by the CPU cores and therefore these values have wide scalability.

References

[1] Lawrence Livermore National Laboratory Blaise Barney. *OpenMP*. 2020. URL: `https://computing.llnl.gov/tutorials/openMP/`.

[2] William J. Burns. *Common Password List (rockyou.txt). Built-in Kali Linux wordlist rockyou.txt*. 2018. URL: `https://www.kaggle.com/wjburns/common-password-list-rockyoutxt`.

[3] *Data Encryption Standard*. URL: `https://www.tutorialspoint.com/cryptography/data_encryption_standard.htm`.