DES password decryption with OpenMP

Cosimo Giani
`cosimo.giani@stud.unifi.it`
University of Florence, Italy

Abstract

The following work is a mid term project for the Parallel Computing course held by professor Marco Bertini, University of Florence. The project implements a decryptor of hashes which are encrypted with the DES algorithm. The aim of this work is to compare the computational costs of both sequential and parallel executions: these comparisons will be made through the observation and the analysis of some graphs, using the information coming from some appropriate tests.

Future Distribution Permission

1. Introduction

This project proposes an implementation of a password decryptor encoded through **DES** (**D**ata **E**ncryption **S**tandard), a symmetric-key algorithm for the encryption of digital data that has been highly influential in the advancement of cryptography [2]. For educational reasons only passwords with 8 characters in length and belonging to the pattern [a-zA-Z0-9./] were considered.
The approach of this work is that of a brute force algorithm which provides, for decryption purposes, to compare an encoded string with the DES encoding of each word contained in the dictionary. If it is found that the encoding of a certain dictionary word is identical to the hash to be decrypted, this term is identified as the password sought and the decryption is therefore considered to have been successful. The project, written in C ++ language, foresees that this procedure is implemented in a sequential and in a parallel version that uses the **OpenMP** framework. The execution times required for the two methods for decryption were then compared through some graphs (created with some appropriate tests) that show the speedup.

2. Dictionary

Due to the type of approach used in this work, the dictionary supplied as input to the program assumes a certain importance: in fact this constitutes the reference dataset containing all and only the terms that can be decrypted.
The dataset of the *haveibeenpwned.com* [1] site was used for its creation: this contains more than 500 million of real world passwords previously exposed in data breaches, that is passwords that have been subject to a security breach which leads to an intentional or unintentional release of secure or private/confidential information to an untrusted environment.
To facilitate the procedure and not have extremely long waiting times, first of all only 15 million passwords were taken and, since these can be of variable length and sometimes contain special characters, to make them fall within the specifics of the problem, i.e. 8 length characters and belonging to the pattern [a-zA-Z0-9./], were further filtered in order to obtain as a final result a dataset of 1.826.532 words, which was used as a dictionary of this work.

3. Implementation

The project presents, in addition to the `main` class, an `Utils` class, containing some useful methods for the purpose of encrypting and print-

ing the speedup, and a `Decrypter` class, which provides the implementation of the methods that allow to execute the decryption of a vector of encoded strings received as a parameter. In particular, this class has two attributes necessary for the encryption of a word by applying the DES algorithm: the first consists of a vector of strings in which the words contained in the dictionary taken as input are stored, whose terms are read and inserted in the aforementioned vector within the constructor of the class; the second attribute is instead the so-called salt, i.e. random data that is used as an additional input to a function that hashes data and helps to safeguard passwords in storage.
Furthermore, in *Decrypter* there are two main methods that allow to decrypt the hashes received as a parameter both sequentially, using the classic brute force approach, and in parallel, with the help of the OpenMP framework. Both methods, in addition to the words to be decrypted, receive an additional parameter relating to the number of tests to be performed for each word: each term to be decoded is in fact decrypted for a number of times equal to this number and the computational time required for the relative decryption is calculated as the average of the times measured for each computation. By this mean, it is appropriate to specify that the `now()` method defined in the `steady_clock` class of the `chrono` library was used to fetch the time at the beginning and at the end of the execution of the two methods.

3.1. Sequential Implementation

The method for sequentially decrypting the desired input strings involves comparing each of these hashes with the DES encoding of each word in the dataset. If a match is found between the encoding to be decrypted and that of a dictionary word, then the term is identified as the one sought.To generate the hash of the passwords contained in the dataset, the `crypt()` function of the `crypt.h` library is used, which takes a password as a string and a salt character and returns a string which starts with the corresponding salt. For the DES-based algorithm, the salt should consist of two characters from the alphabet [./0-9A-Za-z], and the result of `crypt` will be those two characters followed by 11 more from the same alphabet, 13 in total [3].
For each word the decryption is performed for a number of times equals to the number of tests to be performed received in input and for each execution the computational time required is calculated as the time between the start of the for that runs through each word of the dataset and its conclusion.
At the end of the execution of all the tests, the average time for each of the required decryptions is calculated and returned in a vector for the future calculation of the speedup.

3.2. Parallel Implementation

What has just been described has also been implemented with a parallel approach using the OpenMP framework. This version is based on the subdivision of the data to be computed into chunks: the words of the dataset to be analyzed, i.e. whose encryption is to be compared with the hash to be decoded, are divided into groups of the same size; each group is then assigned to the computation of a certain thread among those generated, where the number of total threads has been specified as an input parameter to the method. The operation of subdivision into chunks is carried out on the basis of the number of each thread, retrieved thanks to the `omp_get_thread_num()` function of the `omp.h` library, and the number of words that each thread must compute, calculated as the ratio between the total number of words of the dataset and the total number of threads.
The code is then parallelized with the `#pragma omp parallel` clause with the necessaries directives, including the one that defines the number of threads to generate. In the body of this construct an approach similar to the sequential one is proposed, but with the particularity that each generated thread computes only the words of the dataset belonging to the chunk assigned to it in the manner described above.

In order for the threads to communicate with each other, thus allowing each of these to verify if some other thread had found the decoding sought, a boolean variable `found` was introduced with the role of a *shared* flag among all threads; it has also been made *volatile* to ensure consistency in the reading of its value by the various threads, initialized to false and set to true only if the decoding is found for the hash in question. In particular, as long as `found` is set to false and the dictionary words have not all been "consumed", the current encrypted word is compared with the hash to be decoded. In the event that the two strings coincide, the variable `found` is set to true and the search is interrupted, otherwise in the case in which the word has already been found by another thread or the dictionary words are finished, the threads are interrupted from their work.
From the description of the execution flow just provided it can be seen how the implementation presented allows to save a considerable computational cost compared to a variant that uses the `#pragma omp parallel for` directive: the approach of this work in fact has the particularity of allowing a thread to exit from the *for* with the *break* statement if it detects that the hash has already been found by another thread, which allows, compared to other variants, to prevent the useless execution of some iterations of the loop, whose number increases as the position of the searched word, stored in the respective chunk, decreases.
Another aspect to underline is the encoding phase of each string of the dataset, operation necessary for the comparison with the hash to be decrypted. The `crypt()` function used for the sequential approach is in fact unsuitable for this parallel approach as it does not allow to define the memory space in which to allocate the coding result: this problem is in fact fatal during the computation, since it can happen that two or more threads try to allocate an encrypted in the same memory location with the consequent loss of the first allocated hash. To overcome this problem, the `DES_fcrypt()` function of the `openssl/des.h` library was used for encryption, which receives not only the string to be encoded and the salt characters, but also the address of the memory location in which to allocate the result. It was therefore sufficient to allocate a memory space for each thread within the *pragma* directive and pass the location pointer to the encryption function.

4. Experimental Results

In order to compare the computational times required to decrypt certain passwords using the different approaches presented in this project, different types of tests were performed.
For each term to be decrypted, the experiments resulted in the calculation of the **speedup**, a metric obtained with the ratio between the time required for the sequential decoding and the time required for the parallel one, depending on the number of generated threads. In particular, each test for the calculation of the speedup was performed with a number of threads equal to `2,3,4,5,6,12` and the results were graphed through a Python script to facilitate the analysis of what was obtained.
The program includes two testing phases: the first consists in evaluating the speedup for words placed in specific positions in the dictionary, the second one calculates the speedup obtained on average, i.e. decrypting words chosen at random.
The terms chosen on the basis of their position in the input file are the following:

- ***lzd9bs07***: first word in the dictionary
- ***roxy1440***: word placed at position `(N/2)+1`
- ***tri3ple6***: last word in the dictionary

where `N` is obviously the total number of words contained in the dictionary.
The experiments took place on a **Intel Core i7-8750H CPU** which has 12 cores (6 physical + 6 logical), but the project was build inside a virtual environment through *Oracle VM VirtualBox* [4], a free and open source hosted hypervisor for virtualization. Given the limitations of the virtual machine, which does not allow hyperthreading, the tests were performed on the 6 physical cores.

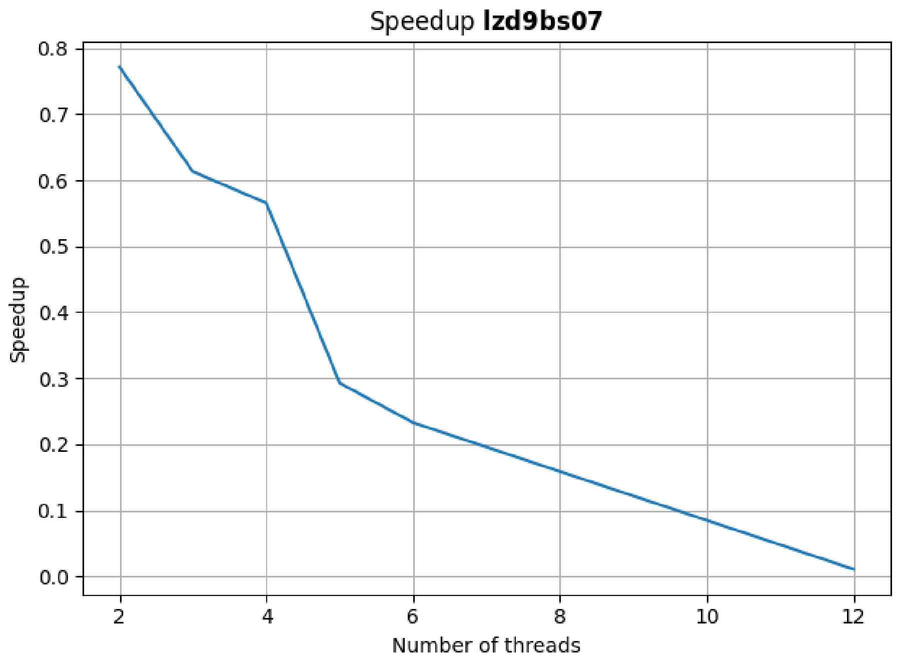

Figure 1. Speedup for the word `lzd9bs07`

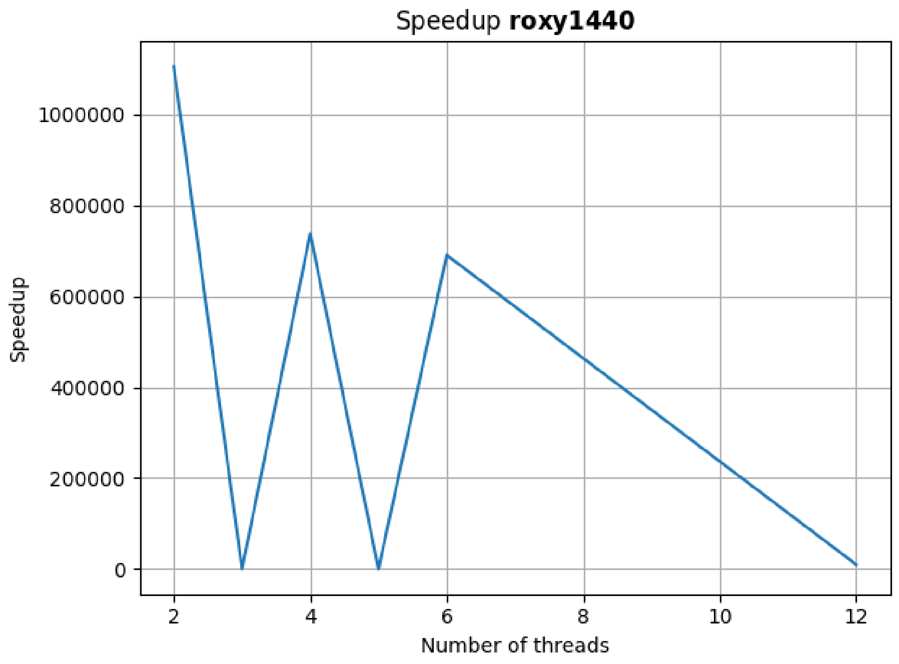

Figure 2. Speedup for the word `roxy1440`

4.1. Specific Passwords Experiments

For the performance analysis of what has been described so far, 30 tests were performed for each of the selected words, both in the sequential and parallel approach, and the execution time was considered as the average computational time de-

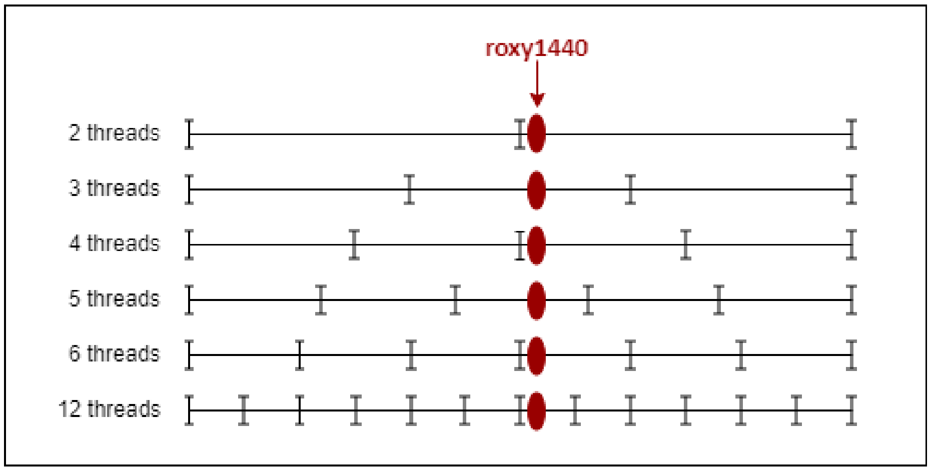

Figure 3. Position of the word `roxy1440` in the respective chunk as the number of threads changes

tected by the individual experiments.
The first term selected is the word **lzd9bs07** located in the first position in the dictionary of this project. As can be seen from the graph in Figure [1], which shows the speedup calculated as the ratio between sequential time and parallel time as the number of threads varies, the parallel approach turns out to be totally inefficient. The best speedup is in fact obtained with 2 threads, which value is in any case lower than 1. By applying a parallel approach for the decryption of terms placed at the beginning of the dictionary provided as input, a significant deterioration of performances is highlighted compared to its sequential variant. This is due to the fact that the latter finds in a negligible time the decoding of terms placed near the beginning of the dataset, a time that cannot in any way improve in parallel but that indeed worsens due to the *overhead* required for the creation and the management of the threads throwed.
As concern the speedup of the term **roxy1440**, which is placed at the position `(N/2)+1`, from the observation of the relative graph in Figure [2] it can be seen how the performances of the sequential approach are outperformed by applying the parallel method. In this case the best speedup is recorded with the execution of only 2 threads as well, in which it reaches a value above one million.
This speedup, apparently too high, is justified by the fact that to find a word placed near the middle of the dictionary, the sequential method must go through almost one million words before finding its decryption.
As shown by the diagram in Figure [3], however, if the term considered is placed at the beginning of a chunk, as it happens for the password *roxy1440* when the number of threads is an even value, the thread in charge of computing the chunk will find the decryption immediately, with the consequence that the ratio between the time required by the sequential variant and the execution time for the parallel one will return an extremely large value. In fact such high values are obtained only if the parallel method is applied generating an even number of threads, while it can be generally *linear* or *superlinear* for an odd number of threads.
Obviously, as the number of threads generated increases, the speedup value decreases due to both the overhead time required by the multiple threads throwed and, in the case of an even number of threads, to the fact that the begin of a chunk progressively moves away from the word considered.
Finally we analyzed the word **tri3ple6**, which is located at the bottom of the dictionary. Precisely, due to the position in which it is found, this is the password that among all those within the dataset requires the greatest computational time for sequential decryption.

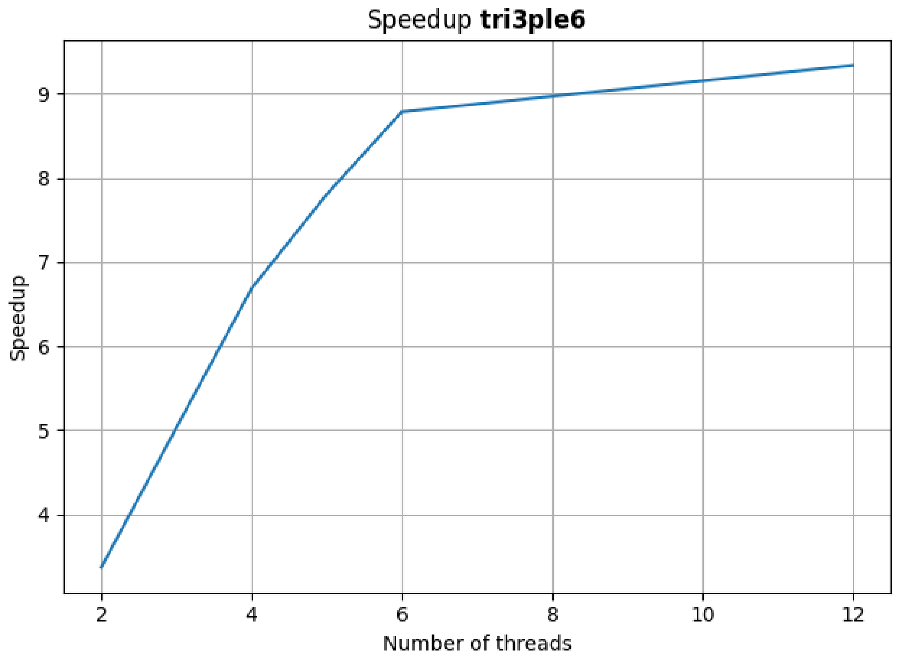

Figure 4. Speedup for the word `tri3ple6`

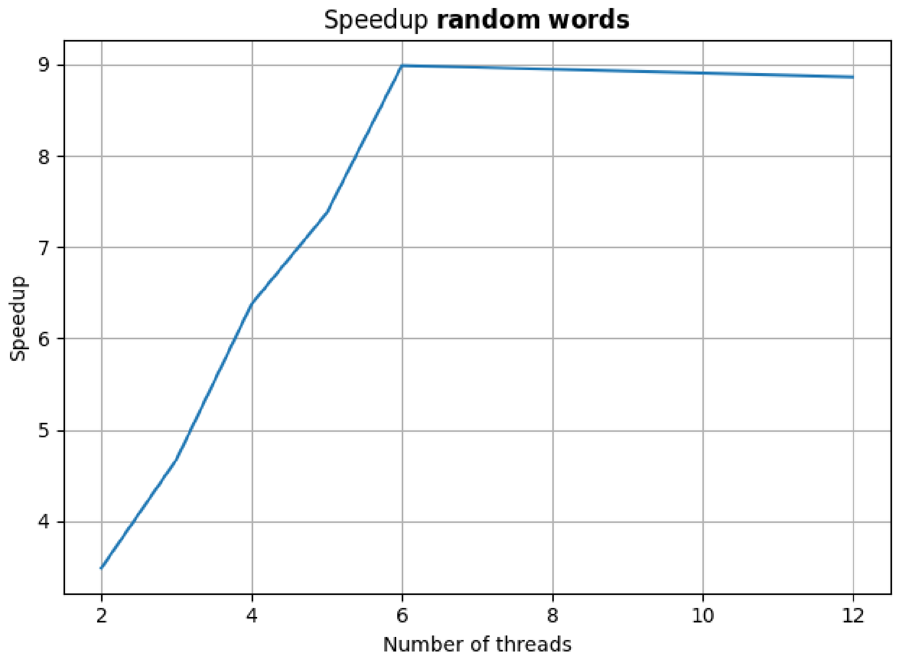

Figure 5. Speedup for random words

From the analysis of the results shown in Figure [4] it can be seen how the growth is almost linear up to 6 threads throwed, reaching a speedup value close to 9, i.e. *superlinear*. After 6 threads, however, the speedup changes its behaviour, since yet doubling the number of threads (12) its value does not seem to increase significantly and still remains around 9.

4.2. Random Passwords Experiments

To complete the experimental phase of this project, the aforementioned random tests were performed: in this case, for reasons related to the high computational time required, it was chosen to perform tests to decrypt a number of hashes equal to 100 and in particular 3 tests for each word were performed.
As can be seen from Figure [5], the analysis of the results produced highlights that the average performances calculated are extremely similar to those relating to the decryption of the word located at the bottom of the dictionary. In fact, in a similar way to the latter, the speedup increases approximately linearly up to 6 threads, reaching a *superlinear* speedup in this case as well, and then remains close to a value slightly below 9 for a doubled number of threads.

5. Conclusions

As can be understood from the analysis of the results presented in this paper, the evaluation of the performance of the parallel approach proposed in this work depends on the different case studies considered. For words positioned at the beginning of the dictionary, the parallel method is found to be totally inefficient, although this is obvious since parallelism induces considerable advantages when the computation takes place on a large number of data, indeed the performance of the method improves for words positioned in the second half of the dataset.
It is also interesting to see a close correlation between the results obtained on average by decoding randomly chosen words and the results obtained by decrypting the word positioned at the bottom of the dataset, which allows us to understand how the parallel approach induces significant advantages in terms of computational time required. In general, however, as can be seen from the results shown in the graphs (in particular Figure 5), having a number of threads higher than the cores present on the machine does not lead to major advantages, that is for the project of this paper, in terms of performances, having more than one thread per core does not introduce significant changes.

References

[1] Dataset passwords
. `https://haveibeenpwned.com/Passwords`.

[2] Des algorithm
. `https://en.wikipedia.org/wiki/Data_Encryption_Standard`.

[3] Useful link to *crypt* function
. `https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/ html_node/libc_650.html`.

[4] Virtual box
. `https://www.virtualbox.org/`.