DES decoding

Johan Andrey Bosso
johan.bosso@stud.unifi.it

Paolo Innocenti
paolo.innocenti@stud.unifi.it

Abstract

With the rise of concerns related to cyber security, an increasing attention has been focused on achieving better and more secure encryption standards. This article wants to show how, by using parallel programming techniques, it is possible to significantly lower the time needed for a brute force attack to recover a password encrypted with the DES algorithm.

1. Introduction

The goal of this paper is to show the benefits, in terms of performance, deriving from a parallelization of a brute force attack in order to decode an encrypted password.
In section 2 we will first describe the task and the basic idea to solve it and present a sequential solution both with C++ and Java. Then, in section 3, we will focus on how we achieved the parallelization of the process using the OpenMP [1] library in C++ and the Thread class in Java. Finally, in section 4, we will present the tests performed in order to evaluate the performance improvements and discuss the achieved results.

2. DES decryption

Data Encryption Standard[2] is an algorithm based on 56-bits encryption keys. Its use over the time has revealed many frailties. Even so, a brute force approach for its decryption still requires a great amount of time as, when considering 8-digit passwords on the [a-zA-Z0-9./] char set in the worst case scenario, it requires encrypting 64^8 passwords with 2^{56} encryption keys.
In order to make it easier to test our implementations, we decided to restrict the domain of the available chars to numbers, lowering the password domain to only a part of the original one. This was considered a reasonable limitation, as it is common to use dates when writing 8-digits passwords. We then converted the brute force approach for password recovery into a research task by encrypting each password with every key and comparing each encryption with a target encrypted password.
The two implementations however handle the task differently. The C++ implementation encrypts all the passwords with a key, before moving to the following key, while the Java implementation encrypts a password with all the keys before moving to the following password.

3. Implementations

3.1. C++ e OpenMP

The C++ version of the program was made limiting the passwords to a dataset made of dates with GGMMYY or MMGGYY format, with its part separated by either a **/** or a **.**.
As explained before, in this implementation we visit the entire password set with a key before moving to the following key and repeating the process, following the algorithm 1. At each iteration of the inner *for* loop we perform encryption on a password-key pair and we compare the result with a given target, ending the process in case of positive match.
The OpenMP API allows to parallelize the process by simply using #pragma directives, which

denote the parallelizable sections. For our implementation, they were easily used to parallelize the computation of the two *for* loops, being each analysis of a password independent from the others and allowing so the preservation of the sequential consistency.
OpenMP parallelization is implicitly run at compile level. When the compiler finds a section denoted as parallel by pragma, it handles it with a fork-join policy, generating threads to execute the section. To manage a *for* loop, OpenMP splits it in chunks, using environment variables to set how to chunk the data and to set the amount of threads. A particular focus has been put on deciding how to chunk the data and, for our case, we decided to make it dynamic, relative to both the data and the key set dimension. We first split the outer *for* loop in 4 chunks by defining it as
#pragma omp parallel for schedule (dynamic, keysetdim/4)
Then, we used a similar pragma directive to split also the inner *for* loop in 4 chunks.
As for the number of threads, we let OpenMP set it since, unless otherwise indicated, it creates a number of threads equal to the amount of cores on the computer, making it the best choice by default. Since each thread can not be influenced by others while executing, OpenMP does not provide synchronization flags and so we used an exit call to interrupt all the threads when achieving a positive match on the target password.

Algorithm 1: Sequential implementation in C++

```
Result: Plain Password
initialization;
for i ← 0 to keysetdim do
    for j ← 0 to datasetdim do
        cand = encrypt(pswset[j],keyset[i]);
        if cand == target then
            plainpsw = pswset[j];
            break;
        end
    end
end
```

3.2. Java

Instead, in the Java implementation, the problem has been extended by not limiting the amount of passwords to a specific dataset and considering in its place the entire space of passwords achievable with the [0-9] char set.
After setting this condition, we first developed a sequential version. To generate the passwords we created a password generator with a *public* method which returned a new password. Then we used the generator within the main loop of the program to generate a new password after comparing the previous one, encoded with every available key, with the target password.
This process can be summed up using the following algorithm:

Algorithm 2: Sequential implementation in Java

```
Result: Decoded Password
initialization;
while !found do
    currentPsw = generatePsw();
    reset key to 0;
    while !found and key ¡ maxKey do
        if testPwd(currentPsw, key) then
            found = true;
        end
        key++;
    end
end
```

To parallelize this process, we used the Thread class, which we extended to create threads that would test a password on the entire key space. We then shared the password generator among the threads to prevent testing the same password more than once and we marked its *generate()* method as *synchronized*, allowing the threads to enter it one at a time. The time overhead caused by this was considered negligible, being many orders of magnitude smaller than the one needed to test the password with every key.
Finally, to set a common condition, we used a shared *volatile* variable in order to prevent the threads from reading a copy of a traditional boolean attribute in its cache.

4. Test and Results

4.1. Hardware Specification

The tests for the C++ implementations were carried out on a machine with a Quad-Core CPU Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, with HyperThreading enabled, 16GB DDR4 RAM and Ubuntu 18.10 as operative system while those for the Java version were performed on a Intel(R) Core(TM) i5-6600K @ 3.6 GHz, with HyperThreading disabled, 16GB DDR4 RAM and Ubuntu 19.10 as operative system.

4.2. C++

While the execution time of the sequential implementation is influenced only by the position of both the password and the key in the entire set, for a password and a key set both split in chunks the parallel implementation one is influenced by their membership chunk and their position in it. To test such influence, a first test on C++ implementation was done by fixing a target password and varying progressively both the dataset and the keyset size to evaluate their impact on the execution time.
A second test to compare sequential and parallel implementations was then performed by varying progressively the dataset and the keyset and reserching the password in various positions of the sets using encryption keys both at the beginning and at the end of the set.

4.3. Java

In order to allow an easier testing of the bigger password domain, another limitation was set in the Java implementation by reducing the number of bits used for the DES keys to an arbitrary value given in input to the program. More precisely all the tests were performed 4 times by encrypting the original password with a random key, chosen within the range of keys delimited by the given bits, and then computing the average.
We first performed tests aimed at analysing the impact of the amount of threads on the execution time. During this part, we set the password and the DES key bits and changed the number of threads.
We then moved onto a second test by increasing progressively the number of bits used for the encryption keys in order to evaluate how the size of the key domain affected the performance.
We then carried out a final test to analyse the difference in time caused by the position of the password, using first a password(14041994) from the first quarter of the password space and then one chosen in the final quarter of the same space(85476948).

4.4. Result

The first tests on the C++ implementations showed that for the sequential implementation the time needed to research a fixed password, by encoding it with one key at a time, was influenced by the key's position in the entire key set, while, for the parallel version, such time depended on the position of the key inside the chunk to which it belonged.

keysetdim	datadim	time(par)(s)	time(seq)(s)	SpeedUp
1000	1000	13.50	50.73	3.73
1000	2000	26.80	109.37	4.06
1000	3000	40.37	161.52	4.00
1000	4000	53.62	190.50	3.55
1000	5000	67.00	238.62	3.56
2000	1000	27.00	95.50	3.53
2000	2000	53.87	191.50	3.55
2000	3000	80.37	287.34	3.57
2000	4000	107.37	382.00	3.55
2000	5000	134.62	477.87	3.54

Table 1. Average time with fixed password

Subsequent tests showed how execution times depended on the size of the dataset and keyset and on the position of the password and the key within their sets. Observing table 4.4, it is clear that the relation between the size and the execution time can be considered linear for both implementations, as the time grows with the same rate at which the size does. We were also able to notice how the position of the target key and the target password within the chunks of the parallel version affected the performance. While researches involving keys and passwords belonging to the first chunk did not highlight any dif-

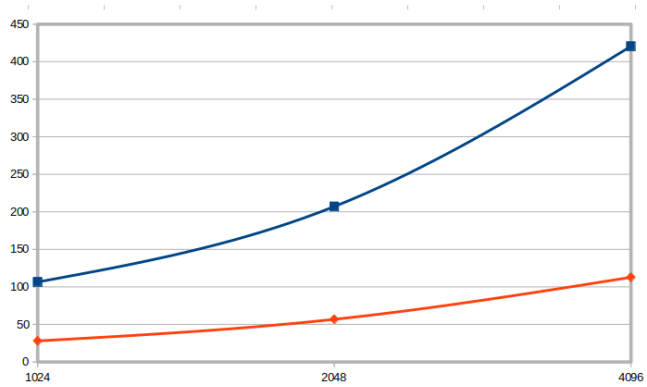

Figure 1. Time trend of sequential and parallel version

ference between the two versions, the execution time reduction could be already noticed considering a key and a password belonging to the second chunk, as in the parallel implementation we do not have to wait to complete the first chunk before analysing the second.
Finally, by considering the gathered results and the speedup as

$$S = \frac{t_s}{t_p}$$

, with t_s the average execution time for the sequential implementation with a given configuration, and t_p the average execution time for the parallel implementation with the same configuration, we computed all the speedups and achieved an average 3.6x speedup with a 0.2s σ.

As for the Java version, the first test, carried out to evaluate the impact of the number of threads on the execution time, showed that using 4 threads was the best choice. In fact, even if the execution times for 8 and 16 threads were smaller than the sequential version, they were higher than those for 4 threads because of the context switching required to run the various threads.

	Time	
	Avg(hh:mm:ss)	σ(s)
Sequential	02:06:53	-
4 Threads	**00:40:17**	40
8 Threads	00:43:25	38
16 Threads	00:46:11	44

Table 2. Results obtained changing the amount of threads using 10 bits and fixed password

After fixing the number of threads, the other two tests were carried out changing the other parameters.
In the first, the results, obtained by varying the number of bits on which to compose the encryption key between 4 and 12, pointed out the high implementation's sensibility to this parameter. This is justified by the need to encrypt the tested password with each key in the keyset, the size of which depends on the bits chosen at the start of the program. It is to be noted that, for these tests, the standard deviation increased together with the bits and this was linked to the choice of using a random key from the key space. While for our tests this resulted in a smaller impact in the execution time when compared to that necessary to test a password with all the keys, it would not have been the case for a 56-bit key space. In such case, using a key closer to 0x00000000000000 would have shortened the execution time significantly compared to one closer to 0xFFFFFFFFFFFFF.

Bits	Sequential		Parallel	
	Avg(hh:mm:ss)	σ(s)	Avg(hh:mm:ss)	σ(s)
4	00:01:56	10	00:00:37	3
5	00:04:05	14	00:01:19	3
6	00:09:02	33	00:02:34	2
7	00:19:07	45	00:05:10	6
8	00:38:43	10	00:11:39	15
9	01:17:32	179	00:21:18	18
10	02:06:53	-	00:40:17	40
11	04:10:36	-	01:18:47	32
12	08:30:32	-	02:31:54	55

Table 3. Results for key-space bits

In the second one, instead, to evaluate the impact of the password position, we repeated the previous tests while using a password belonging to the last quarter of the space of the generated passwords. The results are summed up in table 4.

	14041994		85476948	
	Avg(hh:mm:ss)	σ(s)	Avg(hh:mm:ss)	σ(s)
Sequential	02:06:53	-	$>$ 12h	-
Parallel	00:40:17	40	04:07:22	37

Table 4. Comparison between the two different passwords using 10-bit key space

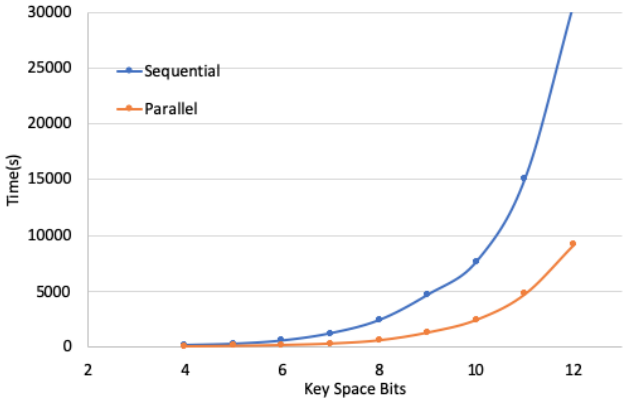

Figure 2. Time trend based on key space bits

Based on the data obtained and estimating the execution times for a higher number of bits, we measured an average 3.3x speedup with a 0.2s σ. Unfortunately, for a higher value of bits, it would still require a very long time for to successfully complete a brute force attack using a standard computer, making the use of dedicated hardware the only option available nowadays for this task.

References

[1] OpenMP API for parallel programming, version 201511. `http://openmp.org/wp/`.

[2] Wikipedia. Data encryption standard, 2019. [Online; Checked on 11/02/2020].