Parallel Computing Mid-Term - Password decryption with C++ and OpenMP

Federico Palai
Universit degli Studi di Firenze
federico.palai1@stud.unifi.it

Federico Tammaro
Universit degli Studi di Firenze
federico.tammaro@stud.unifi.it

Abstract

The aim of this mid-term project is to find a plaintext password given its hash generated by the DES algorithm and assuming to know the salt used to generate the hash; also, a dictionary is given, from which new hashes of words are calculated and compared to the one to be decrypted.
Here will be presented the OpenMP version, including a sequential approach and two slightly different parallel approaches.

1. Introduction

The problem starts from a password hash generated by the Data Encryption Standard (DES) algorithm, which uses a symmetric key to encrypt and decrypt data, and the salt used to generate it; to simplify the problem, only passwords containing 8 characters from the set [a-zA-z0-9./] are considered and we assume to known the salt that has been used. Another input of the problem is a .txt file used as dictionary, with each line containing a common password, date or even a random generated combination from the allowed set.
Then, the search and compare algorithm that has been used is the following:

1. Read an element from the vector containing the dictionary
2. Calculate the hash of the element, using the known salt
3. Compare the hash to the one to be decrypted
   (a) If the two hashes are identical, the plaintext password has been found
   (b) Otherwise, repeat from step 2

2. Implementation

The implementation of this particular problem revolves around a class, **DictionaryDecrypter**, used to implement sequential and both parallel decryption methods.
The OpenMP framework has been used for the parallel implementations of DES decryption, using two slightly different approaches. Both have in common the same method arguments:

- *int runsPerPassword*, number of runs to get a mean time of execution, which may vary depending on the current CPU utilization of other processes running in the background; so, each implementation runs exactly a given number of times and a mean time is calculated at the end of the execution. This is also used to get a mean time of the sequential execution and is passed as an argument while launching the program from command line.
- *int numberOfThreads*, number that specifies how many threads to use in each parallel execution. While using only 1 thread makes the method sequential, every other number of threads will make the OpenMP framework execute the parallel code concurrently.

The *chrono* library has been used to measure the execution time of each method, starting in each method just before entering the loop which iterates along the dictionary vector containing all the words; more specifically, the

std::chrono::steady_clock class has been used to have a monotonic clock, intended to measure intervals of time.

2.1. First parallel method - Parallel for directive

The first method uses a parallel region to create and initialize a *crypt_data* struct for each thread, used in the next *crypt_r* function call, before entering the **parallel for** directive; also, the number of threads to be used is set at the start of the parallel region using the *num_threads* clause.
Then, the parallel for loop execution starts, doing an automatic division of the dictionary into chunks and handing them to each thread; then, a boolean volatile variable has been used to signal each thread whether the encrypted hash has been found in the dictionary.
Once that variable has been set to true, each thread cycles the for loop without doing any work but increasing the indexes, since the for loop can not be terminated using a *break* statement; this introduces a delay, significant as the number of threads grows, after the password has been found, but can not be avoided when using a *parallel for* OpenMP directive.

2.2. Second method - Without parallel for directive

This alternative method creates a parallel region with a given number of threads, but instead of using an OpenMP for directive, it simply creates a for loop inside this region: to achieve this, an equal size for all the chunks is calculated before starting the loop. Each thread is given all the indexes from $thread\_number * chunk\_size$ to $((thread\_number + 1) * chunk\_size) - 1$, where $chunk\_size = dictionary\_size / n\_threads$.
A boolean volatile variable is still used to signal each thread when the encrypted hash has been found in the dictionary, but unlike the previous method, every thread does a check before executing any work inside the loop and can break the loop instead of having to cycle through the remaining indexes. This saves a lot of execution time, making this method always perform better than the first one.
Also, while the first method becomes barely executable with 200+ threads, this method can still run even with 10000 threads and probably even more.

3. The dataset

The dataset used for the experiments below has been extracted from a bigger one containing 10 million passwords [1]: only the ones matching the [a-zA-z0-9./] pattern and 8 characters long have been extracted using a simple Python script. This led to custom dataset containing 304198 common passwords.

3.1. Choosing passwords to test

For a more accurate comparison of sequential and parallel executions, rather than picking random passwords from the dataset, 4 passwords have been chosen to carry out the tests: first word of the dictionary (*password*), last word of the dictionary (*vjht1051*) and two words from around the middle of the dictionary (*mara1992* and *Maple800*).
These passwords have been chosen because of their position when dividing the dataset into chunks:

- *password* is always going to be at the start of the first chunk (first because of its position, not necessarily because it will finish first!).
- *vjht1051* is always going to be at the end of the last chunk (as before, last because of its position).
- *mara1992* is going to vary between the end of a chunk, if the dictionary has been divided into an even number of chunks, and the middle of a chunk, if the dictionary has been divided into an odd number of chunks. This word is necessary since, in the parallel version with manually-split chunks, the chunk where *vjht1051* is located can be smaller than the other ones; this is because indexes are rounded at the highest integer index when dividing the size of the dictionary by the number of threads. On the contrary, the word

mara1992 will always be around the end of a chunk of equal size as all the other ones.

- *Maple800* is going to vary between the start of a chunk, if the dictionary has been divided into an even number of chunks, and the middle of a chunk (the same as *mara1992*), if the dictionary has been divided into an odd number of chunks. This word is necessary since *password*, while being at the start of a chunk, in a parallel execution will always perform worse than the sequential execution (as presented in the results section); so, another word at the start of a chunk was necessary to calculate the speedup of a word in that position.

Their position in the dataset and inside the chunk from 1 to 8 threads can be seen in Figure 1, illustrating the position of the chosen passwords inside the chunks.

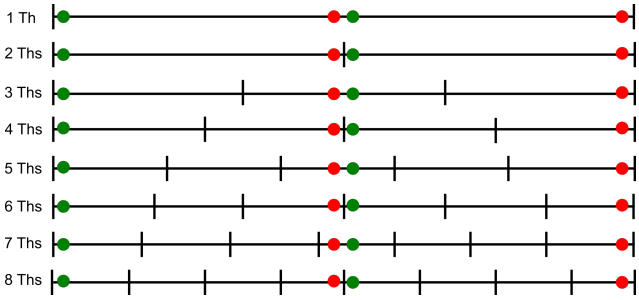

Figure 1. Chosen passwords and chunks when increasing from 1 to 8 threads

4. Experiments

The CPU used for these experiments is an *AMD Phenom II X6 1090T*, clocked at 3.2GHz with 6 physical cores and no hyper-threading (one thread per core).
Using the dataset described in the previous section, parallel experiments were carried out using an increasing number of threads (2-10, 20, 30, 40, 50, 100, 200, 500, 1000, 5000, 10000); for each password decryption, both sequential and parallel, 50 runs were made to suppress outliers caused by the utilization of cores by background tasks.

5. Results

All of the experiments focused on the execution time of each method and the speedup of parallel methods compared to the sequential version.

Predictably, to decrypt the first word of the dictionary, *password*, a sequential method is much faster than a parallel method: while the sequential method iterates only one time then stops, parallel methods have to create each thread, initialize them with the indexes of the chunk to iterate, create the *crypt_data* structures required by the *crypt_r* function and then start. All these actions cause an overhead that is not irrelevant since it is bigger than the search time for the first word of the dictionary; as soon as the password to decrypt slides forward in the dictionary, the overhead becomes irrelevant in comparison with the search time.
So, as shown in Figure 2, there is no speedup at all ($speedup < 1$) in both parallel methods, even though the second method performs better than the first because it immediately stops every thread.

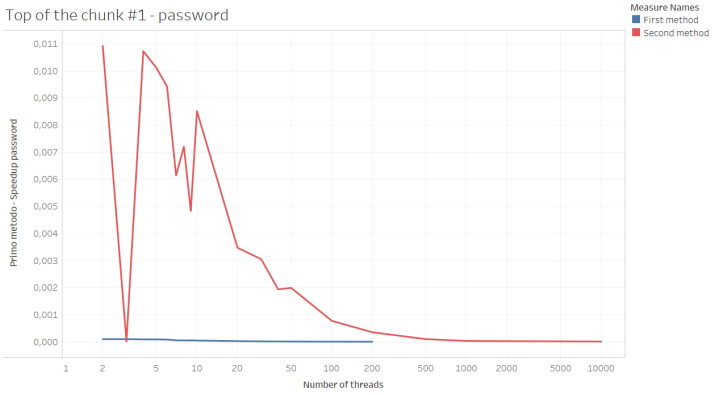

Figure 2. Speedup decrypting the hash of the word ”password”.

Moving to the center of the dictionary, the word *mara1992* will vary between the end and the middle of a chunk, and as we can see in Figure 3, it performs better with an odd number of threads (being approximately in the middle of a chunk) than with an even number of threads (being at the

end of a chunk).
Though irregular, the speedups of both parallel methods can be approximated with a linear or slightly sub-linear speedup until 10 threads, where it starts decreasing for both methods; the linear behaviour can be seen better by plotting separately odd and even number of threads, by comparing speedups when the word is always in the same position inside the chunk.
As before, the second method outperforms the first, stopping each thread right after finding the word in the dictionary.

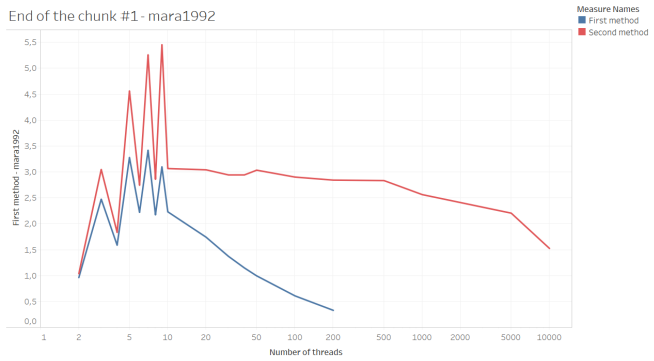

Figure 3. Speedup decrypting the hash of the word ”mara1992”.

The results of the second word placed in the middle of the dictionary, *Maple800*, are highly method-dependent and vary, as for the previous word, because of its position in relation of the number of threads: with an even number of threads the word is at the start of a chunk, while with an odd number of threads is placed approximately in the middle of a chunk.
An interesting result, in this case, is the comparison between the two parallel methods, presented in Figure 4: when the word is at the beginning of a chunk, the second parallel method heavily outperforms the first one, since threads are terminated right away without the need to cycle through all the remaining indexes; on the contrary, when the word is in the middle of the chunk the two methods perform similarly, without a significant difference in speedup.
Both methods have a decreasing speedup when the number of threads increases, because the word slides forward in the chunk due to a rounded split of chunks; after 10 threads performances degrade noticeably, with the first method having no speedup at all after 50 threads, seen in Figure 5.

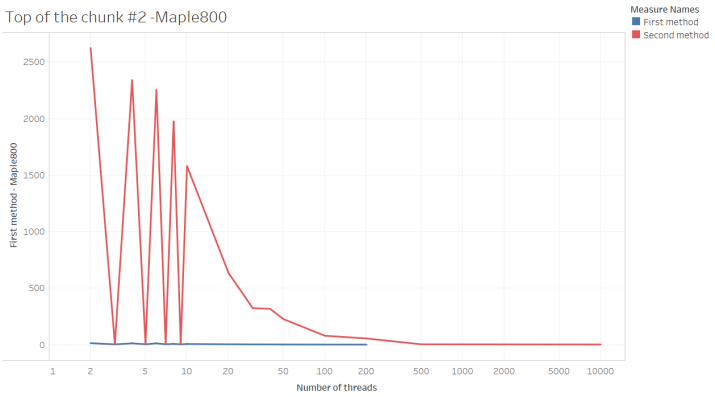

Figure 4. Speedup decrypting the hash of the word ”Maple800”.

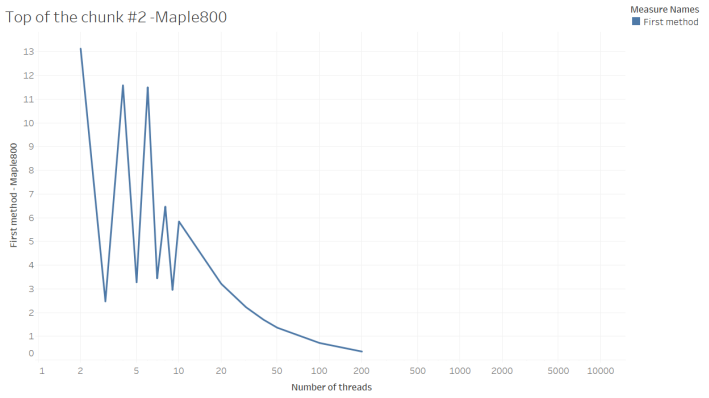

Figure 5. Detail of the speedup decrypting the hash of the word ”Maple800”.

Last but not least, the word at the very end of the dictionary, *vjht1051*, is always at the end of the chunk and its speedup can be seen as linear up until the maximum number of cores available (6, in this case), then a region of constant speedup starts. Speedup values with 500 and 1000 threads, reaching two times those with 6 threads, can be seen as an optimal value of the trade-off between overhead and searching time: chunks are very small and quick to search, making the time to create and synchronize each thread irrelevant. But, as the number of threads grows, the cost

of creating and synchronizing threads overcomes the searching time and the speedup drops significantly.

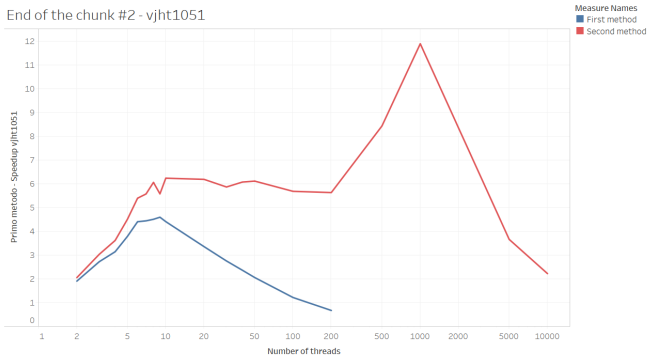

Figure 6. Speedup decrypting the hash of the word "vjht1051".

5.1. Amdahl's law

Since the theoretical speedup is limited to at most 20 times, the speedup values of the password *Maple800* may seem impossible. This is because Amdahl's law considers the execution of the whole task, while the second parallel method presented in this project stops all the threads (and the work) as soon as a password is found. Being at the beginning of a chunk with an even number of threads, the parallel method stops without having done all the work required to comply with Amhdal's law.

6. Conclusions

Unsurprisingly, in most cases the best approach for a dictionary-based password decryption is a parallel approach: then, the problem shifts to the search of the optimal number of threads to get the best speedup.

6.1. Omp parallel for vs manual for parallelization

Using a *parallel for* OpenMP directive is less error-prone and more programmer-friendly, since the framework automatically calculates each starting index for a chunk division, but the for loop can not be interrupted until each thread runs out of indexes to cycle on; so, each thread will continue to cycle without doing any work even when the password has been found.
The best way to improve the parallel execution is by manually handling the indexes splitting and using a normal *for* loop inside a parallel region: this way, the for loop can be broken as soon as the password is found and no empty cycling is required, leading to a significant increase in performances.

References

[1] Daniel Miessler. 10 million passwords list, 2019. *https://tinyurl.com/10MDataset*.