Parallel Computing Mid-Term: Password decryption with OpenMP

Lorenzo Giannella
lorenzo.giannella@stud.unifi.it
https://github.com/lore1379/DESdecrypter

Abstract

The aim of this mid term project is the implementation of a password decrypter, in order to find a plaintext encrypted using crypt (DES Algorithm). We will consider 8 characters lenght plaintexts in the set [a-zA-Z0-9./] and try to decrypt a list of password: this is the so called "dictionary attack". There are 3 implementations, one sequential and two parallel versions, obtained with OpenMP. A simple performance study on speed up has been made, in order to analyze the benefits of parallelism. The developed code is in C++ language, tests were made on a famous list of password called "rockyou", filtered in the set [a-zA-Z0-9./].

1. Introduction

In cryptography, encryption is the process of encoding information. This process converts the original representation of the information, known as plaintext, into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext back to plaintext and access the original information. Encryption does not itself prevent interference but denies the intelligible content to a would-be interceptor. In cryptanalysis and computer security, a dictionary attack is an attack using a restricted subset of a keyspace to defeat a cipher or authentication mechanism by trying to determine its decryption key or passphrase, sometimes trying thousands or millions of likely possibilities often obtained from lists of past security breaches.

2. Algorithm Description

Given a set of 4 plaintexts and a salt, random data that is used as an additional input to a one-way function that hashes a password, the algorithm generates 4 password hashes. For the DES-based algorithm, the salt should consist of two characters from the alphabet [a-zA-Z0-9./], and the result of crypt will be those two characters followed by 11 more from the same alphabet, 13 in total. Only the first 8 characters in the key are significant [1]. Then the algorithm proceeds according to the following phases:

1. Read an element from the dictionary vector
2. Calculate the hash of the element, with the known salt
3. Compare the hash to the one we want to decrypt
   (a) Hashes are identical so the plaintext has been found
   (b) Hashes are not identical so we go back to phase 1 with the next element.

3. Algorithm Implementation

Initially, I implemented a sequential version of the Decrypter. Afterwards, 2 parallel OpenMP implementations are compared to evaluate the benefit of parallelization. Tests were made on the hardware outlined in Table 1.

Two method arguments are used to help with the test phase:

[1] https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_650.html

CPU	Intel(R) Core(TM) i5-7300HQ CPU@2.50GHz
GPU	NVIDIA GeForce GTX 950M
RAM	8,0 GB
Hard Disk	SSD - 64 GB

Table 1. Hardware used to test sequential and parallel code performance

- experimentRuns: numbers of runs performed in order to have a more accurate estimation of time needed to complete operations. Then a mean is calculated with collected times.

- numberOfThreads: how many threads to use in each parallel execution.

3.1. Sequential Version

The sequential version of the DES Decrypter was implemented in C++ programming language [2]. It operates utilizing two vectors to record execution times and compute the mean. Then it follows the steps for every encrypted password (wantedPassword) in the vector initially created:

1. clear the run times vector for a fresh timer start

2. performs the operation the number of desired times (experimentRuns) and for every plaintext in the dictionary:

   (a) encrypt the plaintext using crypt fuction and the salt.

   (b) compare the encryption attempt with the password we are looking for.

3. when the password has been found, it breaks from the loop and save the execution time in the proper vector.

4. after the number of desired runs, it proceeds to compute the mean of run times in the vector and saves it in the one containing execution times.

[2] C++ implementation is available at https://github.com/lore1379/

3.2. Parallel Version

OpenMP [2] (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

In this project [3] two versions of OpenMP code have been produced taking into account Step 1 and Step 2 [1] optimizations:

- Step 1: "omp parallel" directive is used to create a parallel region, then "omp for" is used to do an automatic division of the region. Even if it is not fully exploited, this is used to reduce threads startup (overhead) and save time in partitioning and handing chunks to threads in subsequent operations.

- Step 2: calculate and manually partition the loops across the threads, which allows for removal of barriers and required synchronization. The impact of the manual partitioning of the arrays is that it reduces cache thrashing and race conditions by not allowing threads to share the same space in memory.

Step 2 is a further optimization made in order to avoid the work sharing of for loops by explicitly computing the work assignment based on thread ids, ideally speeding up the calculations.

3.2.1 Automatic division

The first method creates a parallel region, thanks to the omp parallel directive, set all variables to shared and use the clause to specify the number of threads to be used. Then a crypt data struct is created and initialized for each thread, this will be used by the crypt_r fuction. This is needed because it makes the function reentrant and in particular thread safe.

When the parallel for loop execution starts, an automatic partition of the dictionary is handed

[3]

in private copy to each thread. With the aid of a boolean volatile variable we are able to signal whether the password has been found.

Unfortunately this approach introduces a delay as the number of threads increase because once the variable is set to true, each thread must complete the loop even if it's not doing any work aside increasing the index. This can't be avoided when using a parallel for directive since the loop can't be terminated using a break statement.

3.2.2 Manual division

In this second method instead of using an OpenMP for directive, we calculate and manually partition the dictionary by simply creating a for loop inside this region. To achieve this, we use the variable chunckSize, that is the size of dictionary divided for the number of threads. Then each thread is given all the indexes from numThread * chunkSize to ((numThread + 1) * chunkSize). 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, so this method always perform better than the first one.

3.3. Experiment Data

The dataset used for the experiments below has been extracted from a bigger one containing 14,341,564 unique passwords, used in 32,603,388 accounts from the company RockYou. [4] Back in 2009, a company named RockYou was hacked. This wouldn't have been too much of a problem if they hadn't stored all of their passwords unencrypted, in plain text for an attacker to see. They downloaded a list of all the passwords and made it publically available. For this experiment only the ones matching the [a-zA-z0-9./] pattern and 8 characters long have been extracted. This led to custom dataset containing 1,591,813 common passwords.

[4] https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt

3.4. Passwords chosen

Four passwords have been chosen from this dataset to perform the tests:

1. password: this is the first word of the dictionary and, regarding position, it is always going to be at the start of the first chunk.
2. sky_surf: this password is at the end of first half so it will vary from the end and the middle of a chunk, depending if the dictionary is divided into an even or odd number of chunks. This is useful to compare a word that will be at the end of a full size chunk, taking in consideration that the last chunk, containing hotel333, could be smaller. This is because we use the function *ceil* that rounds the division of the dictionary upward, returning the smallest integral value that is not less than the size of dictionary divided for the number of threads.
3. skutterr: this password is at the beginning of the second half so it will vary from the beginning and the middle of a chunk, depending if the dictionary is divided into an even or odd number of chunks. This is useful because password, at the beginning of dictionary will always perform better in sequential version than in the parallel one so we needed a word at the start of a chunk to value the benefit of parallelism.
4. hotel333: this is the last word of the dictionary and, regarding position, it is always going to be at the end of the last chunk.

3.5. Performance Analysis

In order to understand if these parallel implementations could improve the performances with respect to the sequential version, a simple speedup analysis has been performed. For each password both the sequential and the parallel executions have been recorded 10 times and then averaged, in order to obtain a better estimation.

Speedup $Sp(n)$, is defined as the ratio of the execution time $T(l)$ in a sequential manner to the corresponding execution time $T(n)$ in the parallel version for some class of tasks of interest:

$$Sp(n) = \frac{T(l)}{T(n)} \quad (1)$$

Tests were made on the hardware outlined in Table 1. The operating system is Ubuntu 18.04.6 LTS.

The CPU used for these experiments is an Intel(R) Core(TM) i5-7300HQ CPU@2.50GHz with 4 physical cores and no hyper threading. Comparison will be done considering, for every password, the speed up of parallel implementations. However we also want to see the effect of the variation in the number of threads.

3.5.1 First password: password

As we expetcted, a sequential method is much faster to decrypt the first word of the dictionary. While the sequential method iterates only one time and stops, the parallel method have to setup threads, creating and initializing them with the indexes of the chunk to iterate. Then it has to create the crypt_data structures required by the crypt_r function and compute results. All these actions and the cost of managing threads cause an overhead that is not irrelevant. As soon as the password to decrypt moves forward in the dictionary, the overhead becomes irrelevant with respect to search time and we will see the benefit of parallel implementations. As we can see in Figure 1, there is no speed up at all in both parallel methods. However we can see that the second method will always performs better as it immediately stops every thread as soon as the password has been found.

3.5.2 Second password: sky_surf

The second password "sky_surf" will vary between the end and the middle of a chunk, Figure 2. As we can see it performs obviously better with an odd number of threads, being approximately in the middle of the chunk. The speed

Figure 1. Speed up using a parallel method to decrpyt the hash of "password"

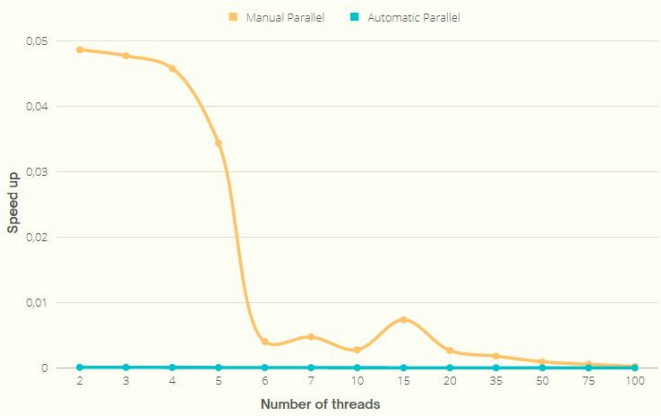
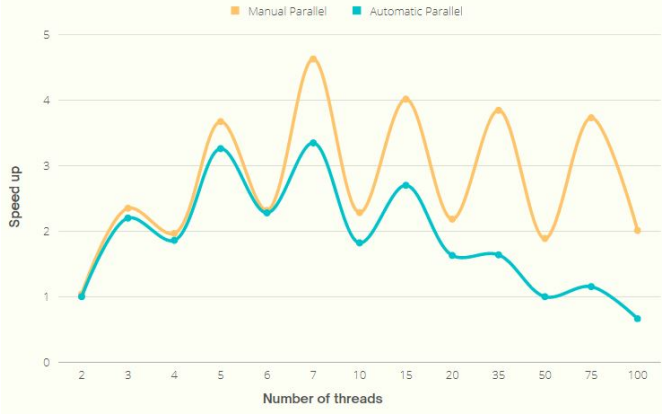


Figure 2. Speed up using a parallel method to decrpyt the hash of "sky_surf"



ups of both parallel methods, with odd number of threads, peaks at seven threads with a superlinear behaviour, thanks to the early termination of loops. The speed up of both parallel methods, with even number of threads, however is always sublinear. With more than 7 threads the speedup starts decreasing but the manual parallel method maintains a good speedup with respect to the increasing number of threads. As before, the second method outperforms the first stopping each thread after finding the word in the dictionary.

3.5.3 Third password: skutterr

The third passoword "skutterr" will vary between the start and the middle of a chunk, Figure 3. As we can see it performs always better with an

Figure 3. Speed up using a parallel method to decrpyt the hash of ”skutterr”

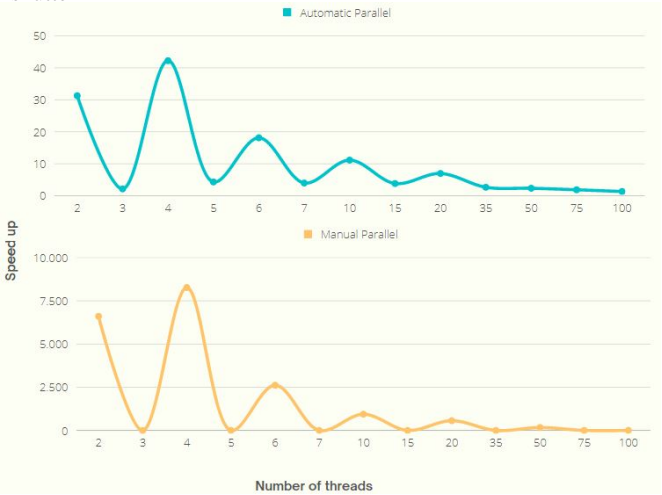
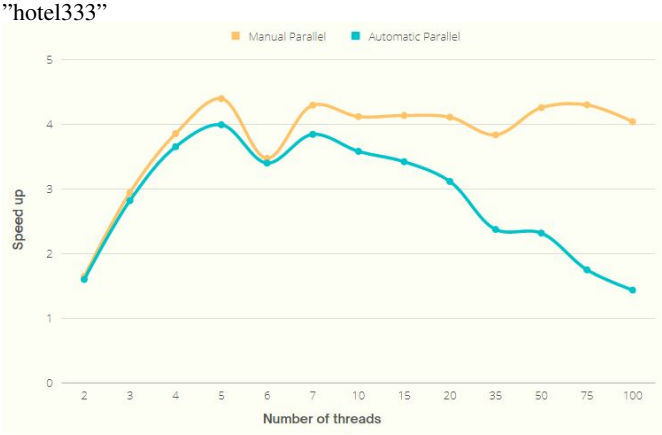


Figure 4. Speed up using a parallel method to decrpyt the hash of ”hotel333”



even number of threads, being approximately in the start of the chunk. In this case the manual method drammatically improves the speed up of the decryption and greatly outperforms the automatic method. This is because we’re exploiting at it’s best the fact that the cycle is terminated immediately without the need to iterate through the remaining indexes. On the contrary, with an odd number of threads and the password in the middle of the chunk, the two methods perform similarly to other passwords. After four threads we can see a significantly drop in performance due to the fact that chunks are getting smaller and we lose the benefit of the early termination of the cycle.

3.5.4 Last password: hotel333

The last password is at the end of the dictionary and its speed up is almost linear until five threads, Figure 4. Then, with the automatic parallel method, the speed up starts to fall off after five threads, due to fact that the overhead cost of creating and managing threads is costantly increasing. On the contrary the manual parallel method has an almost constant behavior of the speedup. Interesting is the fact that we have similar or even better results with more threads due to the fact that chunk are smaller and easier to search and compensate the larger number of threads.

3.6. Final Observations

Amdahl’s law is often used in parallel computing to predict the theoretical speedup when using multiple processors. Since the theoretical speedup is limited to at most 20 times, the speedup values of the password skutterr may seem wrong. However Amdahl’s law considers speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. The manual parallel method stops all threads as soon as a password is found, making the search process not operating on fixed length chunks. As we can see the best approach for a dictionary-based password decryption is a parallel approach. However results will vary depending on the number of threads and method used to perform the search. So in addition of chosing the optimal number of threads, the best way to improve the speed up of parallel execution is to manually handle indexes split. This way we can use a normal for loop inside the parallel region and terminate it as soon as the password is found saving time with the execution of the remaining indexes of the cycle.

References

[1] M. Bertini. Shared memory: Openmp optimizations. page 20, 2021.

[2] OpenMP.org. Openmp compilers & tools. 2019.