Parallel Computing Mid-Term - Password Decryption with Java

Federico Palai
E-mail address
federico.palai1@stud.unifi.it

Federico Tammaro
E-mail address
federico.tammaro@stud.unifi.it

Abstract

Decryption of passwords using the DES function, upon strings of fixed length set to 8 and characters in the set [a-zA-Z0-9./]. This work is a simulation of a dictionary attack and it is intended to show the differences between a sequential and a parallel approach. Both algorithms are developed in Java.

1. Introduction

Main purpose of this work was to show how a Java decrypter would work for password protected with the DES algorithm. The problem presented is a *dictionary attack*: it is supposed to have a clear passwords dictionary and, by giving an encrypted password, the program should find out if it is in the dictionary, encrypting one by one each word and comparing the result with the provided hashing of the password.

It is easy to assume that such a problem would be better solved with a parallel solution. Aim of this work was to show whether this intuition was right or wrong.

1.1. Problem description

As mentioned in the Introduction, the problem we wanted to solve is a research problem. The idea was to be as close as possible to a real case scenario. It is not unusual for a real hacker to have a dictionary containing all the most used passwords and to use it to carry out dictionary attacks (like the one we simulated). In fact a file of this type is easily available on the Internet, and this was also our starting point. Since our was a toy example, we had boundaries:

- The encrypting algorithm was the DES (*Data Encryption Standard*) a proved-insecure symmetric-key algorithm for the encryption of data but easy to use thanks to already implemented functions and high speed performances.

- Was assumed that every word was composed by 8 characters. This might be the biggest assumption, even because is demonstrated that, usually, if users are forced to use a 8-characters password, they tend to insert a date. Under this assumption a brute force attack with only numbers might be the best solution in a real hacking attack.

- To reduce hacking attacks, usually the DES algorithm presents a *salt*, a sequence of random bits. However, because the salt sequence must be saved in order to retrieved the plain password, we have assumed that along with the dictionary of passwords we also have the salt sequences.

- Finally, the code was ran over a laptop with 4GB of RAM, and an Intel® Core™ i7-4500U CPU @ 1.80GHz $\times$ 4 with Hyper-Threading Technology.

2. Java Approach

2.1. Preprocessing

As pre-processing operations we have decided to retrieved all the words in the dictionary file, saving them within an ArrayList. This choice is intended to save time by avoiding I/O operations.

The DES algorithm has not been implemented but a library version has been used instead. In particular the version inside `javax.crypto` library (imported in the project using Maven) is used.

2.2. Sequential Version

The sequential version of the algorithm is quite simple. Given the dictionary (retrieved from a `.txt` file and stored within an ArrayList of type String) and the hashed-key we want to retrieved as input parameters, basically it iterates over the dictionary and at every step, encrypts the current word from the dictionary by using the DES algorithm and compares this with the hashed-key. If these two are the same, the cycle is interrupted for the password has been found. Otherwise it will continue until the end of the dictionary.

2.3. Parallel Version

There were many possible solutions for the parallel version of the algorithm due to the advanced methods implemented for threads programming in Java.

Among those, we have decided to use a *Threads Pool*, managed by an *Executor*. The main idea is to create a pool of threads with fixed dimension (N), give the executor some tasks and let it decide how to mange the threads in order to execute those tasks. In this way we avoid to manually creating the threads giving the executors this responsibility and we can start all the threads just once.

The tasks submitted to the executor are similar to the sequential version. They represent the work every threads should do. So, as in the sequential version, the task iterate over an ArrayList of String, encrypting at every step the current word and comparing that with the hashed-key we want to found. However, we have some differences with the sequential algorithm. First of all, the tasks will not iterate over the whole dictionary but only on a part of it. So, the dictionary is ideally divided into a fixed number C of chunks (the way of dividing the dictionary will be discussed later) and every task received a chunk to work on. Another difference is that whether the password is found, an Atomic boolean, used as a flag, is set to true and the cycle interrupted. After every checking, the task also check the flag and whether it is found to be true, break the cycle. This mechanism is used to stopped every still running task once the password is found, ensuring that, even with a delay (although short enough in terms of CPU cycles), the tasks will not work pointlessly.

So, we can summarize how the parallel algorithm works with the following steps:

1. After creating an ArrayList of Task instances, it is filled up with a number of tasks equals to N (the threads number).
2. After that, an executor with a fixed size threads pool is created:

```
ExecutorService executor =
    Executors.
newFixedThreadPool(numOfThreads);
```

3. Finally, all tasks within the ArrayList are simultaneously started, giving the executor full management responsibility:

```
executor.invokeAll(tasksToExecute);
```

   The `invokeAll()` function will wait until all the tasks are completed. After all tasks have finished their work, the executor is shut down.

```
executor.shutdownNow();
```

Concerning the `Task` class used, it will implement *Callable*. This solution has been chosen to use the `invokeAll()` function over a collection of Task instances (as discussed above). Apart

from that, the class has been designed to ensure that every internal variable is as light as possible. For instance, in an earlier version of the algorithm, an ArrayList of String representing the chunk was passed to the task. To avoid wasting time creating the ArrayList and passing it to the task, this solution has been abandoned, preferring to pass the indexes of the chunk. The dictionary itself has been preferred not to be passed, but instead was passed the calling class, already containing the dictionary accessible with a get method.

The only method present in the Task class is `call()`. This is mandatory to be implemented when using the Callable interface. Within this method, the chunk is visited and for each word within it, this one is encrypted and compared with the searched hashed key. To better understand the obtained results, it is important to highlight that, when a password is found, an AtomicBoolean is set to true and the loop stops. This Boolean is shared among all the threads which will check its status at the end of the test of each word. Once the boolean is found to be true, the loop is broken and the task is terminated.

```
public Boolean call() {
   for(int index=startIndex;
       index<endIndex; index++) {
      //encrypting the current word
      if
          (cryptedPassword.equals(hashedKey))
          {
         outcome.set(true);
         System.out.println("Password
             found: " + currentWord);
         return true;
      }
      if(outcome.get()) {
         break;
      }
   }
   return false;
}
```

3. Problem Analysis

3.1. Metric Used

The main metric used to evaluate the results is the *Speed-Up*. Speed-up can be defined as the ratio of the sequential execution time to the parallel execution time, i.e. :

$$S_P = \frac{t_s}{t_p} \tag{1}$$

where P is the numbers of processors used, t_s is the execution time for the sequential algorithm and finally t_p is the execution time for the parallel algorithm.

Another metric used which could give useful information, is the *Efficency*. Efficency can be defined as the ration of speed-up to the number of processors used, i.e. ;

$$E_P = \frac{S_P}{P} \tag{2}$$

Also important and relevant for understanding the results obtained, is *Amdahl's Law*. It states that the speed-up, given P processors is

$$\begin{aligned} SP &= \frac{t_s}{f \cdot t_s + (1-f)t_s/P} \\ &= \frac{P}{1 + (P-1) \cdot f} \end{aligned}$$

So, the maximum speed-up is limited by $SP \rightarrow (f-1)$ as $P \rightarrow \infty$ [1].

Note that even though 95% of a program is parallelizable, it is highly unlikely to see a speed-up of more than 20 times.

3.2. Hypothesis and Thesis

The assumption that was made during the developing of the algorithm was that each chunk has a dynamic size, depending on the number of threads we are using at the time (so, depending on N, the number of threads within the threads pool). Another assumption is that we are looking for words that are definitly inside our dictionary.

[1] This contents are an extract from the slides by Professor M. Bertini, for the corse of Parallel Computing at UNIFI, academic year 2018/19

As already mentioned, this is a strong assumption, justified by the academic and didactic purpose of the work.

According to this assumptions, in the parallel dictionary attack, the actual position of the word inside the dictionary becomes highly relevant. Suppose, for example, that we are looking for a word that is in one of the first position of a chunk (and also that it is in a chunk subsequent to the first, the reason will be discussed later). In this scenario, of course, we would observe a really high speed-up value, much more than the 20 times stated by Amdahl. This result is justified by the fact that, in the code, after a word is found, each task will terminate its work (perhaps there may be a delay to try the current password before checking the boolean, but it is not really relevant). Amdahl's limitation, on the other hand, concerned workloads that must be executed completely, without interruption, for each thread.

So, following this idea, the test we intended to execute were:

- Trying to confirm the hypothesis by using words whose will always be in the first position of a chunk and evaluating their speed-up. For being sure that the word will be always in the first positions of a chunk, we are going to use multiples number of threads. The figure below will better shows our intent:

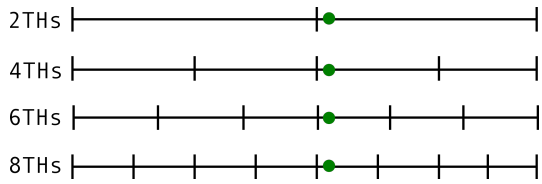

Figure 1. Every line represents the whole dictionary, divided into a numbers of chunks according with the number of threads used. The green dot represents the position of the word to find. It is possible to see that it is always at the beginning of a chunk.

  The expected result is: *really elevated speed-up value*.

- Trying to stuck as much as possible to the conditions of truthfulness of Amdahl's law, we can test the algorithms with words positioned at the end of a chunk (always with multiples number of threads to assure that). In this way, assuming a real parallel execution (so every chunk dumped on a physical core), every thread will be forced to examined almost all the chunk passed:

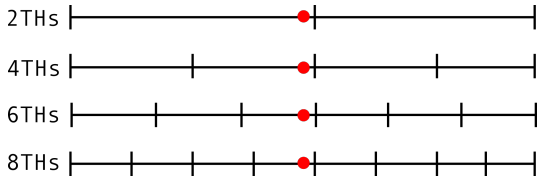

Figure 2. The red dot represents the position of the word to find. It is possible to see that it is always at the end of a chunk.

  The expected result is: *a more normalized speed-up value, not beyond the 20 times Amdahl's law suggests*.

- By using one of the previous case but with the not-tested numbers of threads (1,3,5,7,...) we will have the scenario of words positioned in the middle of a chunk:

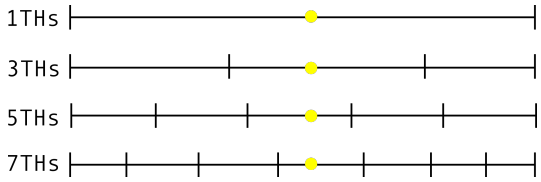

Figure 3. The yellow dot represents the position of the word to find. It is possible to see that it is almost always in the middle of a chunk.

  The expected result is: *a middle value between the previous results*.

4. Test and Results

All the tests have been conducted on the base of the following image, summarizing what said before. As previously shown, every line represents the whole dictionary and the partitions in which has been divided. It is important to remind that the number of chunk (and consequently the chunk dimension) is due to the number of thread used. On each line are also represented the four words we are looking for:

- In green the words whose will always be in the initial part of the chunk, multiples of 2. In

particular those words are: `password` and `Maple800`.

- In red the words whose will always be in the final part of the chunk, multiples of 2. In particular those words are: `mara1992` and `vjht1051`.

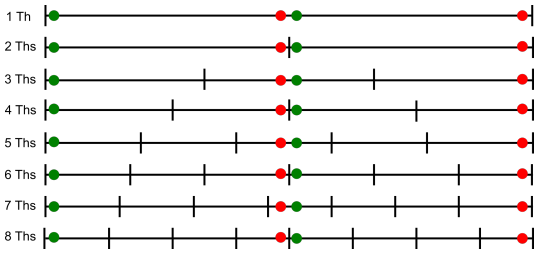

Figure 4. Words position scheme for testing. In green we have, in order, `password` and `Maple800`, while in red, `mara1992` and `vjht1051`

4.1. Tests with chunk initial passwords

The intuition we had about the words in the initial part of a chunk was that speed-up values would be much higher than the limit suggested by the application of Amdahl's law. Below is the graph obtained with the green words, using 10 threads:

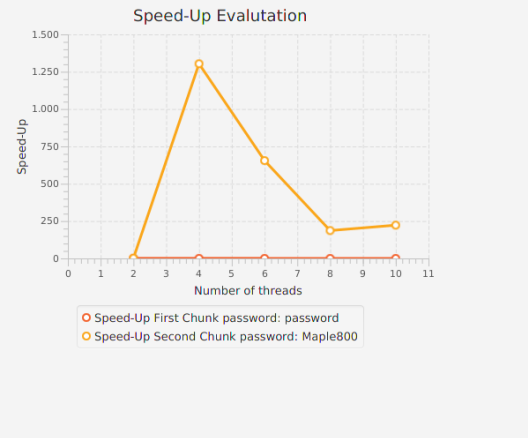

Figure 5. Green words research with up to 10 threads

The first hypothesis seems to be partially verified. In fact, we have extremely high speed-up values. In addition, the highest peak is reached with 4 threads since this is the number of physical cores on the machine used. Moreover, the use of a larger number of threads leads to worse performances. The only unusual thing at first, could be the always 0 value for the first green word. But this is perfectly reasonable, because the first word is found by the two algorithm in the exact amount of time but, for the parallel one, there is more time to consider because of the creation and start up of the threads.

Following there is the same test, but on a maximum number of threads set to 100:

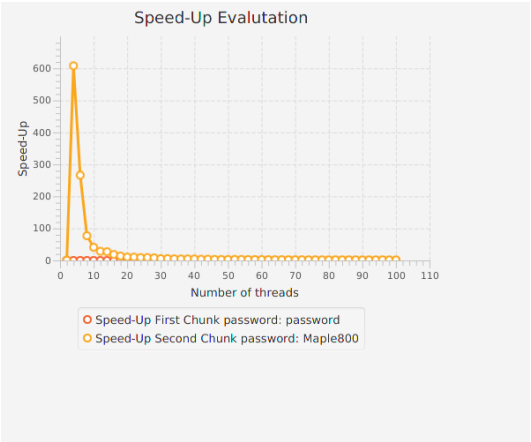

Figure 6. Green words research with up to 100 threads

here it is even more evident how increasing the number of threads heavily affects the performance.

4.2. Tests with chunk final passwords

As for the red words, the intuition was that the speed-up values would be much more "normal" and respectful for Amdahl's law. Below is the graph obtained with the red words, using 10 threads:

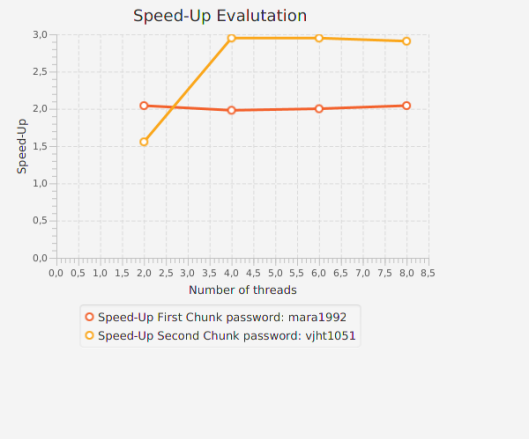

Figure 7. Red words research with up to 10 threads

We can notice that the first red word has almost constant values, while the second one has an initial increase and then remains almost constant. Our hypothesis was that the values would have been under the 20 times faster, and they actually are. Moreover, it is reasonable that the second word (in the last part of the dictionary) has better values than the first one: the parallel time are quite similar while the sequential are really different.

If we use 100 threads, the result is a bit different from what we could expect:

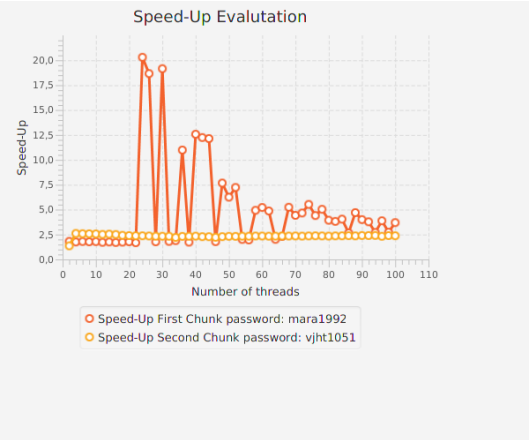

Figure 8. Red words research with up to 100 threads

we can notice that the initial part has the same behaviour of the previous graph. From around the 22nd thread, however, if the last word still remains constant in values, the word in the middle begins to have oscillatory peaks converging to the constant values of the second word. This behaviour could be explained by thinking that, starting from a certain number of threads, the chunk are so small that we basically are in a situation where the word is quite at the beginning of the chunk rather than at the end. It is like being again in the same situation as before. But, unlike the previous test, we must now consider the higher number of threads the JVM have to handle, scheduling the works among all of these.

4.3. Tests with chunk initial passwords but different number of threads

By using again the green words, but with number of threads no longer multiple of two but three, we now have at every step, these words in the middle of chunks (i.e. we are now considering the yellow words). The following image shows the test results for 10 threads:

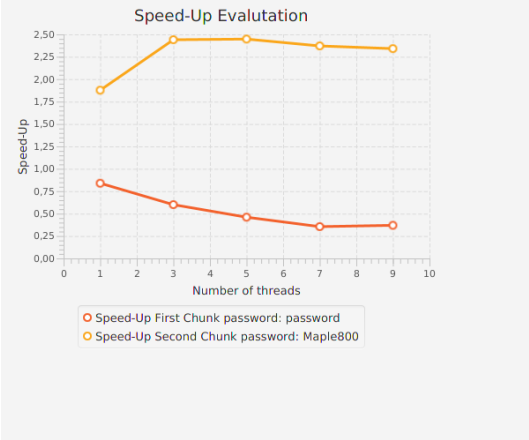

Figure 9. Green words research with odd number of threads, up to 10

We can notice that the first word is always found faster by the sequential algorithm and it is represented by speed-up values under 1. About the second word, can be noticed values pretty similar to the ones obtained with the red words, in accordance with our hypothesis.

This behaviour is quite maintained even if the number of threads grew till 100 (even with some more peaks), as the figure below shows:

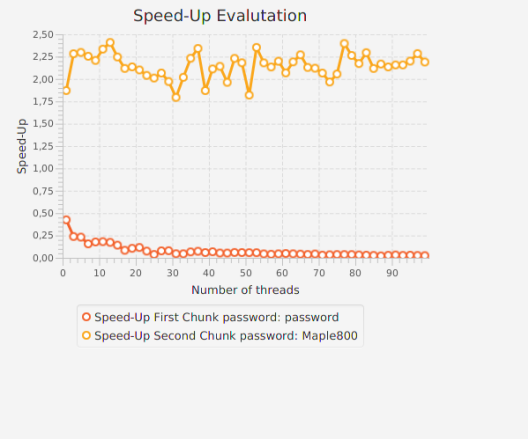

Figure 10. Green words research with odd number of threads, up to 100

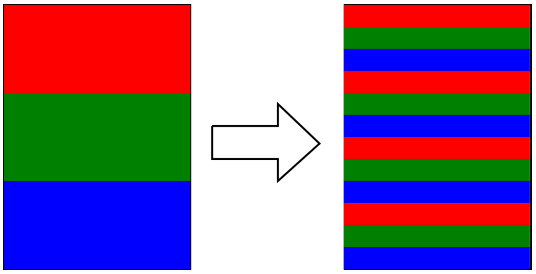

Figure 11. Red stands for H, Green for M and Blue for L

5. Conclusions

Thinking of a real case scenario, given these results, it might be smart not to continue to increase the number of threads. Both because we have tested that this have a marginal impact on the parallel algorithm performance, but also because it is important to keep in mind that, when we ask for more threads than the number of physical cores, the JVM starts the virtual ones, affecting the performance itself. So, one solution could be to continue using a fixed numbers of threads, equal to the number of actual physical cores of the machine, and work instead on the data structure. The dictionary could be divided into blocks, containing the words in order of probability. Thus, the first chunk will have the most common word in first position as well as the n-th chunk will have the n-th most common word, in the same position. Running the parallel algorithm on a data structure like this will increase the probability of having the high speed-up values we have experienced. For example, if we have a quad-core machine, we can run the parallel algorithm with a fixed threads pool set to four on a structured dictionary as explained, with four chunks of decreasing probability. If we assume some probability bands (H for high, M for medium and L for low), the dictionary could become as follow: