PC-2020/21 DES Finder Project

Lorenzo Arena
`lorenzo.arena@stud.unifi.it`

Abstract

This project was made as the first work for the "Parallel Programming" exam at the University of Florence. The objective was to create two version of a software which, given a dictionary of passwords, could find the one matching a given combination of hash and salt; the first version had to be implemented as a sequential program, while the second had to be parallel, thus taking the advantages of multithreading. The encryption algorithm used was the DES used by the `crypt` *and* `crypt_r` *Linux C functions (see* `man crypt`*); for the ease of development another utility software has been created to generate both random or progressive dictionaries of passwords with the relative hashes and salts. The projects has been developed in C on a Ubuntu machine and all tests were made on a Intel i9-9900 CPU. The project is hosted on GitHub at* `https://github.com/lorenzo-arena/des-finder-project`.

Future Distribution Permission

The author of this report gives permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. The sequential solution

The sequential solution has a straightforward approach: it takes the hash and salt as the inputs, with an optional name for the dictionary file; then it starts reading the dictionary line by line, computing the hash and comparing it with the given one. Once a correspondence is found the program stops and prints the found password on the command line; if a correspondence is never found an error log is reported.

2. The parallel solution

The parallel solution, implemented using C PThreads, uses one of the design patterns for multithreading solutions: *producer* and *consumers*. The program, as in the sequential solution, is started with a given hash and salt; optionally, the desired number of threads to use can be specified. The main thread then creates one *producer* thread and the stated number of *consumers* (or the maximum number for the CPU on which the software is running); then, using a barrier realized with a condition variable contained inside the search result structure, it waits until the password has been found or the entire dictionary has been processed. The *producer* thread starts reading the dictionary file and fills a structure with a slice of the dictionary; once such structure has been filled the *producer* appends the data to a queue and starts creating a new set of passwords taken from the dictionary. Each *consumer* thread, when started, tries to get a set of passwords to compute on the queue, possibly waiting for a condition variable until the *producer* has created a set for them; once a set is collected all the contained passwords are encpryted and checked against the original given hash, one by one:

- if they doesn't match the thread tries to collect a new set
- if they match the thread signals the event to all the other threads, including the main one, using the condition variable on the search result

3. Results and observations

An extensive suite of tests has been conducted on a machine with a i9-9900 CPU; the dictionary used contained 5M passwords. The tests involved changing both the number of threads used and the position of the password in the dictionary; the times have been measured multiple times and averaged. The Figure 1 represents the trend of the time necessary to find the matching password on a given position for different threads numbers: most notably, the parallel solution with just 1 *producer* and 1 *consumer* threads is already better than the sequential solution because of the reading and computing split in different threads; we can also notice how for increasing number of threads the improvements in time start to reduce, or the performance even decreases (in the 16 vs 15 threads case).

It's also interesting to see how the different solutions behave when the password is found at the start of the dictionary (i.e. between the position 1 and 1000). From the tests it came out that the sequential solution is more efficient if the password is found in the first ~350 positions; this is because the used size of each set of password created by the *producer* thread is 300. Instead, the parallel solution run with a big number of threads (from 10 on) takes about three times the sequential solution time to find the password for the overhead related to thread management; this is rapidly pulled down by the time necessary for I/O operation (i.e. reading from the dictionary file) when the password is found after position ~500. The chart for time necessary when password is found between position 1 and 10000 of the dictionary is showed in Figure 3 and we can see that each line for the parallel solutions presents a periodical increase in the time; that is due to the fact that in some cases the password is found by a *consumer* on the beginning of a set, while some other times it is found at the end.

Other important evaluations can be made by observing the speedup charts, showed in Figure 4 for passwords found at position 1M and in Figure 5 for passwords found at position 5k:

- when the password is found later in the dictionary using more threads gives better results but that stops at around 15 threads, thus we can say that that is the optimal number for running this solution on this machine; it is expectable that this number will be higher when CPUs with more cores are used.
- when the password is found sooner in the dictionary the parallel solution gives anyway better results than the sequential solution but those results stops getting better when using more than 6-8 threads and even start to degrade when using more than 14 threads.

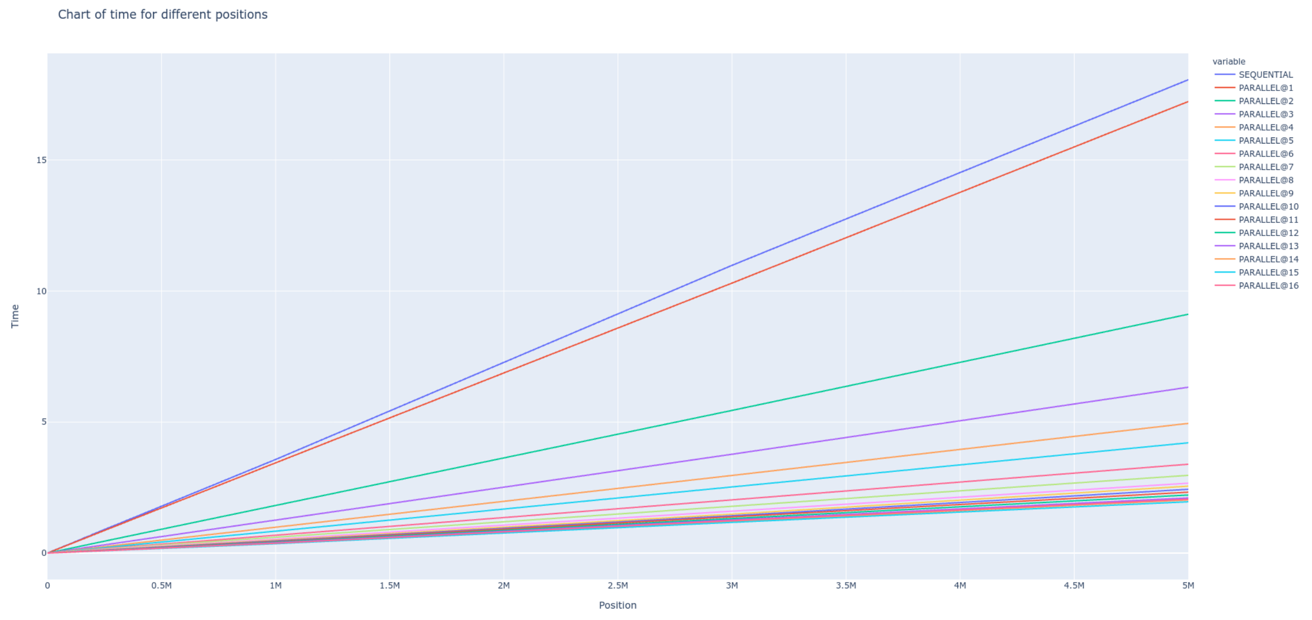

Figure 1. The times measured for different positions

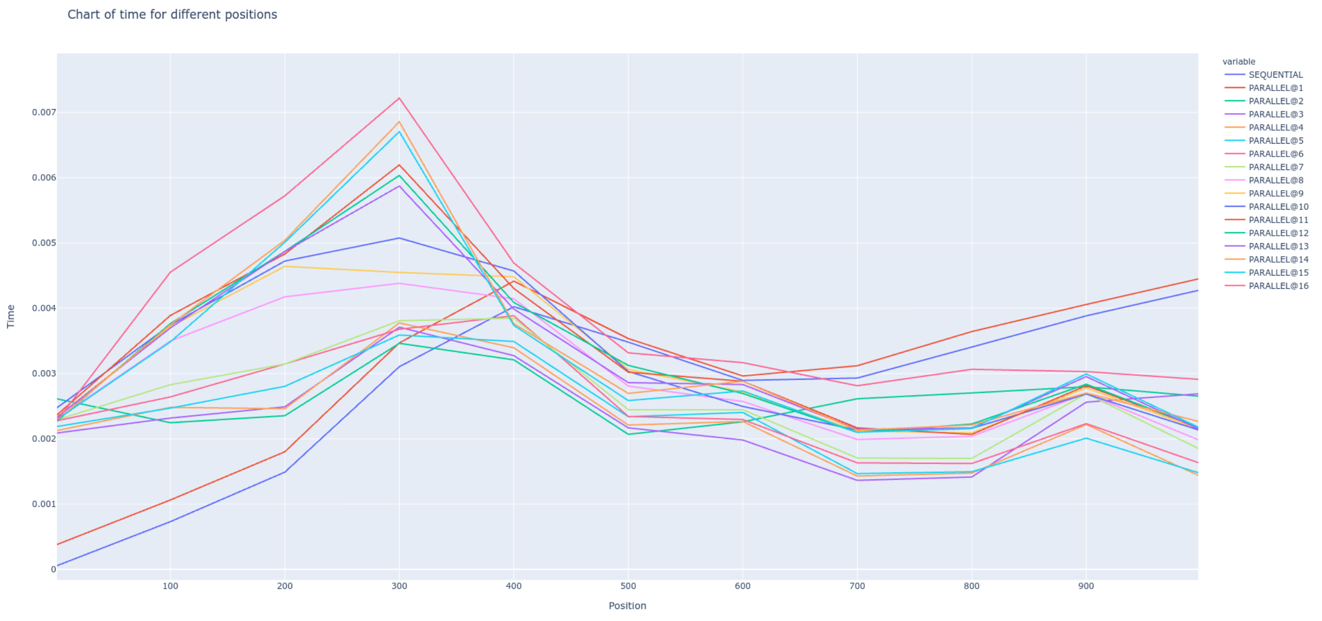

Figure 2. The times measured for the first 1000 positions

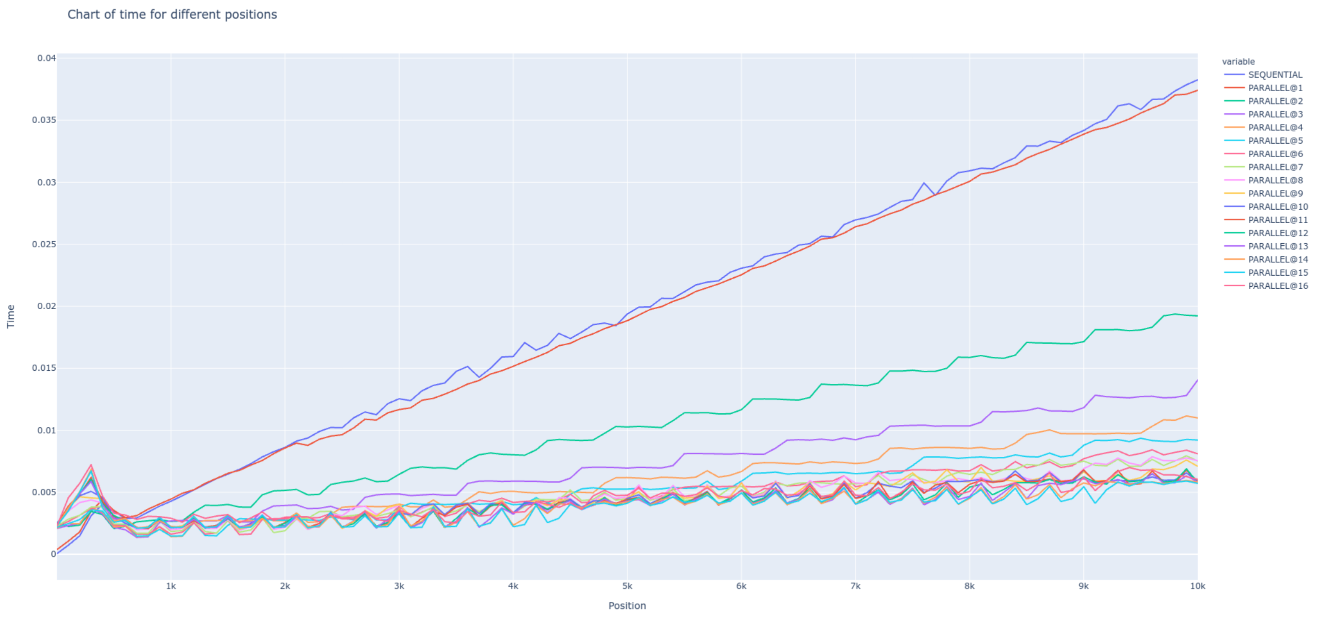

Figure 3. The times measured for the first 10000 positions

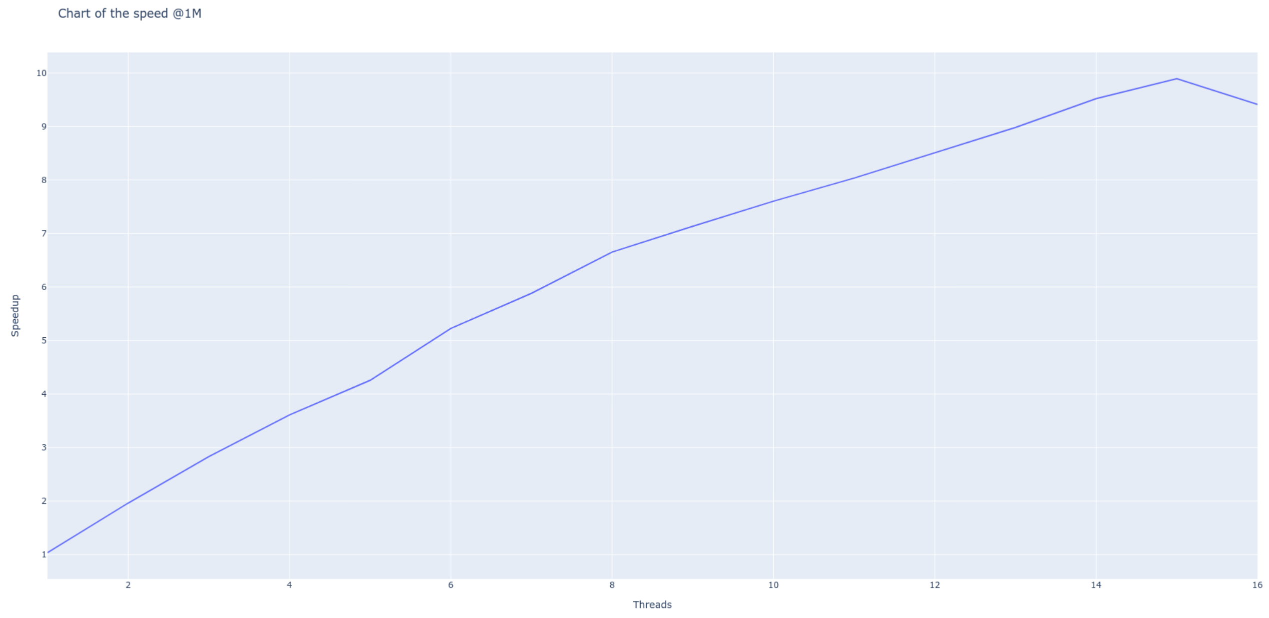

Figure 4. Speedup searching for the password at position 1M

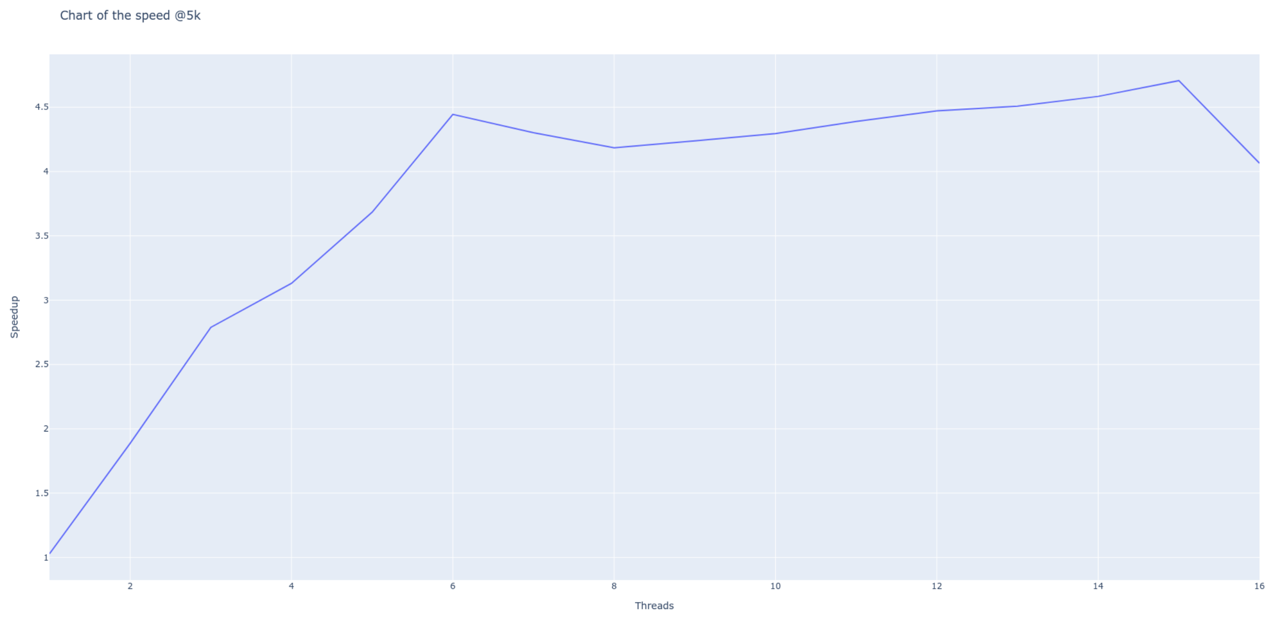

Figure 5. Speedup searching for the password at position 5k

References

[1] B. Barney. POSIX Thread Programming. `https://computing.llnl.gov/tutorials/pthreads/`.

[2] Linux. Crypt man page. `https://www.man7.org/linux/man-pages/man3/crypt.3.html`.