Parallel Computing - Mid-term assignement - Password Decryption

Ubaldo Puocci
Università degli Studi di Firenze
ubaldo.puocci@stud.unifi.it

Abstract

The art of password decrypting has surely been one of the most hot topics in programming for a long time. What we are trying to achieve in this work are results that can show whether or not this problem lends itself well to a parallel programming approach, and what of the aforementioned approaches works best. We will present three different algorithms implemented with different directives introduced by the OpenMP API.

1. Introduction

The first thing we have to talk about to define our problem is what our inputs are. We start by taking as an input a password crypted using a symmetric-key algorithm, the DES algorithm, for which we assume to know the salt used to crypt it. To simplify our problem, we only take into consideration passwords in the set [a-zA-Z0-9./]. This set of allowed password is stored in a .txt used as a dictionary by the program. The algorithm used to find the password given its salted hash is the following:

1. Read one of the password contained in the given dictionary
2. Crypt it with the known salt
3. Compare the returned hash to the one given in input to the program
4. If the hashes are equal, the password has been found
5. Otherwise repeat from step 1

2. Implementation of the algorithm

Our algorithm needs to know how many runs per password it has to do: this information is given as input by the user. While doing multiple, identical runs per password does not help us achieve anything for our problem, it is done as it will smooth out execution time and speedup numbers. In fact, we can then compute the mean for this values instead of taking only the information from one run.

All the decryption methods implemented for this project are managed by the **PCWorker** class. First, we will discuss about the two parallel implementations: `parallelAutomaticAttack` and `parallelAttack`. Both of them were implemented using the OpenMP framework and have the same argument:

- `int numberOfThreads`: it specifies the number of threads to be used within each run of the parallel version of the algorithm. This number is "fixed" for it can't be changed by the user. We will use 2, 4, 6, 8, 10, 12, 14, 16, 20, 100, 500 and 1000 threads for each run.

Our sequential implementation of the algorithm, `sequentialAttack`, takes no arguments because it doesn't need the number ff threads for which it has to split the work. To measure the execution time of the algorithm the `chrono` library has been used. In particular, `std::chrono::steady_clock` has been used just before and after our algorithm be-

gan iterating on our password dictionary, for both parallel and sequential version of it.

2.1. Parallel automatic attack

This method is automatic in the sense that it splits the work between each thread automatically. The number of threads is passed as an argument to the function that implements this particular algorithm.

First, the algorithm allocates a `crypt_data` structure for each thread that will later be used with the `crypt_r` function to compute password hashes. This choice was mandatory as `crypt_r` is the reentrant and thread-safe version of `crypt`, and the latter could have not been used in our algorithm. Then, the algorithm uses the `#pragma omp for` directive to automatically split the dictionary between each thread and each of them then starts computing hashed for a particular password.

If a password has been found, the value of a `volatile bool` variable is flipped. When this happens, each thread encounters a `continue` at the start of the loop. This lets us not do any more computation but introduces a significant delay that is proportional to the number of cycles yet to be completed and the number of threads. This was a mandatory choice as the `#pragma omp for` directive does not support `break` statements.

2.2. Parallel attack

This method is quite different from the previous one. First, the dictionary split is calculated manually before the execution of the algorithm by splitting the dictionary in equal number of words, rounding the number up for each thread. Then, a parallel region is initialized using the `#pragma omp parallel` directive. Inside this region, a `crypt_data` structure is allocated and each thread begins analyzing only the words assigned to it, that is from $\textit{numberOfWords} \times \textit{threadNumber}$ to $\textit{numberOfWords} \times (\textit{threadNumber} + 1) - 1$.

As the previous method, a `volatile bool` variable is still used to check wheter the password has been found or not. However, this time we can use the `break` statement inside our `for` loop. This means that as soon as one thread founds the password, any other thread will exit the loop, resulting in generally faster computations than the automatic method.

2.3. Sequential attack

This method is quite trivial as it implements the algorithm proposed in the Introduction in a sequential way: it simply passes through all the words in our dictionary crypting and comparing them one by one with the wanted password using the `crypt` function. Once found, a `break` statement halts the `for` loop to avoid doing unnecessary work.

3. The tests

All tests have been run on a PC with an AMD Ryzen 3600 CPU with 6 core, 12 threads and a 3.6 GHz base clock speed. The password dictionary contains 904197 passwords and we have to take into consideration which password we are trying to decrypt since the user can pass them as an argument to the program. Choosing the right password is important because it can land on a different part of a chunk during our inevitable split during the computation. We have chosen:

- `pjpO1g7v`, as it's always going the be at the start of the first chunk;
- `2UBdti0U`, as it's always going the be at the end of the last chunk;
- `Kg575EAq` and `williazz`, as their position is going to vary from the start and the middle of their respective chunk, depending on the number of splits, hence threads, that we use.

All tests have been run 10 times to eliminate aleatory overhead.

4. Results

What we are trying to analyze from our tests is the speedup of each method compared to the sequential one.

First, we are going to analyze the speedup time for our first password, `pjpO1g7v`. As we would expect, the sequential method is always faster than both our parallel methods, and that's because they have to account for thread creation time, chunk split and assignment, and for the time it takes to declare and initialize the `crypt_data` structure.

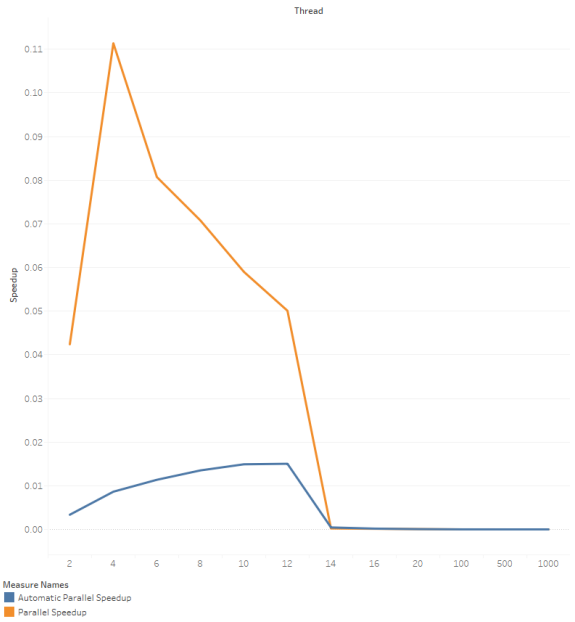

Figure 1. Speedup for the password `pjpO1g7v`

There is no speedup as it is < 1, and yet we can see a direct effect of our implementation of the algorithms: the parallel method is faster than the automatic parallel method because of the `break` statement that stops every single thread as soon as the password is found. The automatic parallel method instead has to wait for all threads to finish even if the password has been found because of the `continue` statement.

If we analyze the run time of our algorithms, we can clearly see that the sequential one has a constant run time, whereas the other two implementations do not. We can see their run time spike up from the 14 threads mark, exactly as the results in Figure 1 would suggest.

It is clear that the overhead caused by the initialization of each structure needed for the parallel methods to run, far exceeds the time it takes to the sequential method to decrypt and find the right password.

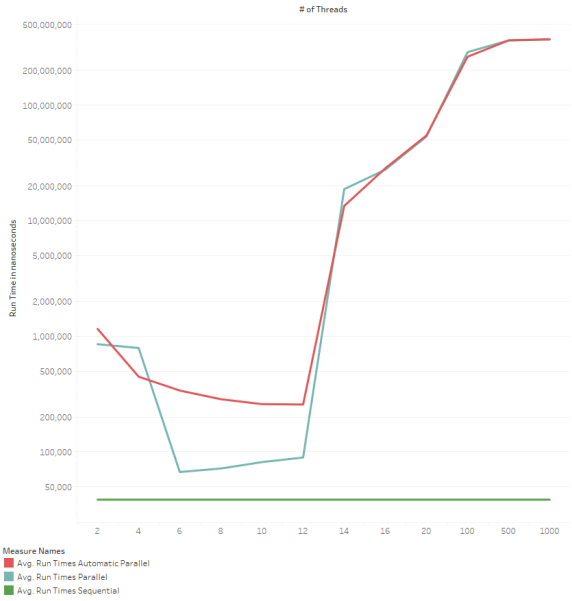

Figure 2. Run times for the password `pjpO1g7v`

We are now going to consider the results from the computation where we were trying to decrypt the password `2UBdti0U`.

Figure 3. Speedup for the password `2UBdti0U`

As we can see from the graph in Figure 5, the speedup almost reaches 4.5 at 12 threads, that is the native CPU thread limit. Then, the algorithm

performance actually decreases because the time cost of initializing and organizing all the threads and their structures exceeds the time it takes to the sequential algorithm to find the password. After that, we reach a plateau were our performance gains are marginal given by the fact that the research space gets so small that this fact balances the overhead given by the high number of threads. We can see that we have some spikes in the graph, the major one being at 100 threads. It is most surely because, given the high number of threads, we divide the password dictionary in really small samples and the thread that found the password had been executed before the others by the internal scheduler.

As before, the parallel method outperforms the automatic parallel method because of the `break` statement: this is also easily seen at the 100 threads spike.

Now we can continue to analyze our results with the two passwords that are varying their position in the computation: `Kg575EAq` and `williazz`.

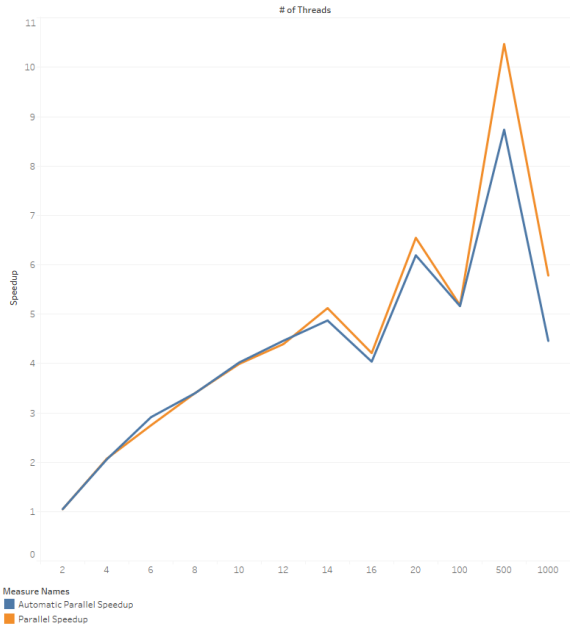

Figure 4. Speedup for the password `williazz`

With a behaviour that's similar to the other computations, we can see a linearly rising speedup from 2 threads to 12 threads. After that, we see what seems and erratic graph, but it has and explanation: as the words move from the start to the middle of their respective chunks, the run times of the algorithm vary so much that it has a big effect on our speedup. What we can also clearly see, as always, is that our parallel method performs better than the automatic parallel one.

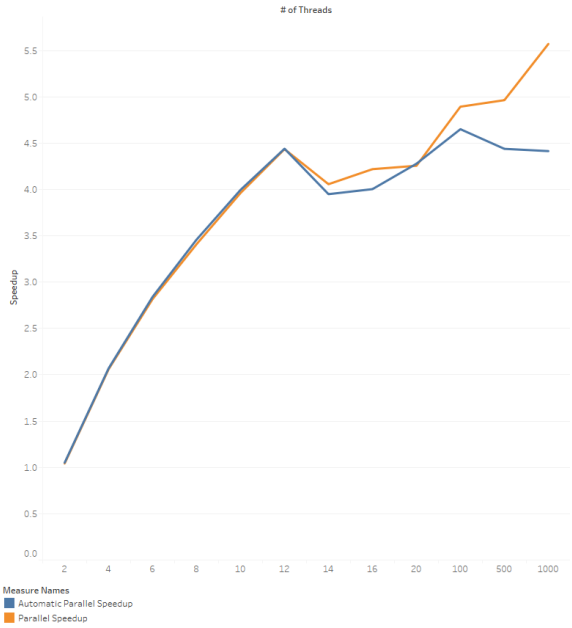

Figure 5. Speedup for the password `Kg575EAq`

5. Conclusions

As we may have thought since the beginning, the problem of decrypting passwords based on a given dictionary lends itself well to show the capabilities of parallel programming. The best approach is surely a parallel one, the problem simply shifts to find the optimal number of threads to run our algorithm on.

As expected, the speedup we obtained through our tests never exceeded the theoretical upper limit set by Amdahl's law, which is 20.

Regarding the use of the `#pragma omp for` directive, it surely is a really helpful and easy way to implement automatic parallelization based solely on a given number of threads. Not being in charge of the work split facilitates the job of the programmer, although one can achieve far better performance with the `#pragma omp parallel` directive and a manual split. That

is because, as we have already mentioned in this work, the ability to stop all threads early when our goal is reached hugely reduces the overhead caused by waiting for them to complete all their, now useless, tasks.