Password Decryption (DES) with OpenMP and POSIX Threads

Giulia Bertazzini
giulia.bertazzini@stud.unifi.it

Niccolò Guiducci
niccolo.guiducci@stud.unifi.it

Abstract

The aim of this project is to brute force decrypt a password of eight characters, encrypted with the POSIX standard function crypt(). In order to do this, it has been created a specific bag of words, containing some of the most used eight characters passwords to compare with the encrypted one. We propose a sequential and two parallel implementations to measure how the parallel versions can improve the performances.

1 Introduction

Given a password of eight characters in the set [a-zA-Z0-9./], the purpose of the project was to decrypt it. To achieve this, we implemented some kind of brute force attack: it means that we created a bag of words containing some of the most used eight characters passwords, then, one at a time, we encrypted it using the function *crypt()* and finally we compare it with the encrypted one that we want to decode (which is also encrypted with the function *crypt()*, by hypothesis).
In this project, we propose a sequential implementation of what was previously explained and also two parallel versions (the first one using OpenMP and the second one using PThreads). In this way, we can estimate how the parallel implementations can improve the performances, measured in terms of mean execution time and speedup.

2 DES Decryption

The **Data Encryption Standard (DES)** [1] is a symmetric-key algorithm for the encryption of digital data, based on Feistel cipher.
It uses 16 round of Feistel structure and the block size is 64 bit. Though key length is 64 bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm, but are used only for checking parity and then discarded.
Since DES is based on Feistel cipher, it consists on a round function, a key generation and, additionally, a initial and final permutation, as shown in the figure below [1].

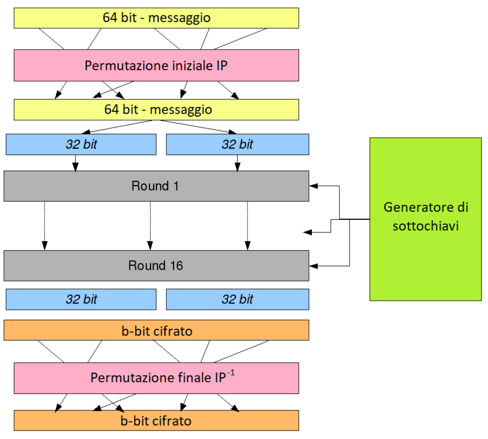

Figure 1: General structure of DES

3 Implementation

In this section, we describe the sequential and parallel implementations of password decryption, which all make use of a bag of words specially created for this purpose. All the implementations are written with the programming language C++.
Our project consists of four main modules:

- **word_generator.h**: contains the function to extract the bag of words specifically created for the project purposes;
- **sequential_decryption.h**: contains the functions which implement the sequential version of the project and the related tests;
- **omp_decryption.h**: contains the functions which implement the first parallel version of the project us-

ing the multi-platform API *OpenMP* and the related tests;

- **pthread_decryption.h**: contains the functions which implement the second parallel version of the project using the POSIX Threads library (PThreds) and the related tests.

3.1 Bag of Words

The bag of words we created is extracted from a bigger one [4], which contains the 10 millions most common used passwords from real users. Since in this project was requested to decrypt a password of eight characters in the set [a-zA-Z0-9./], the function **extractBOW(string bow_path)** of the module *word_generator.h* extracts, from the 10 millions bag of words, each password that satisfies the requirements and saves it in a txt file (bag_of_words.txt). At the end of this process, the extracted bag of words contains **303846** unique passwords, that we used to brute force decrypt the encrypted password.

3.2 Sequential Implementation

The simplest implementation is clearly the sequential one. As mentioned before, the module *sequential_decryption.h* contains two methods: sequentialDecryption and testSequential.
The **sequentialDecryption** method is the one responsible to decode the encrypted password. More specifically, the encrypted password, the salt (which is a two-character string chosen from the set [a-zA-Z0-9./], used to perturb the algorithm in one of 4096 different ways) and the vector containing all the passwords of the processed bag of words are passed as method's parameters; then, the method encrypts each password in the bag of words, one at a time, using the function *crypt_r()* and compares the encrypted string obtained with the one passed as parameter: if they are equal, the password has been found and the research stops, otherwise it continues to scan the vector until it reaches the end.
The function we use to encrypt (*crypt_r()*) is a reentrant version of the function *crypt()*, which make use of a particular structure to store result data and bookkeeping information.

```
string sequentialDecryption(const char
    *encrypted_password, const char *salt,
    const vector<string>& bow){

   auto *structure = new crypt_data;
   structure->initialized = 0;

   for(const auto& word : bow)
      if(strcmp(crypt_r(word.c_str(), salt,
          structure), encrypted_password) ==
          0)
         return word;

   return "Password Not Found!";
}
```

On the other hand, the method **testSequential** is the one responsible to execute the test, in the form of repeated executions of the function **sequentialDecryption**, in order to evaluate the performances, expressed in terms of mean, minimum and maximum execution time. We discuss more in details about the tests and the results obtained in sections [4] and [5].

3.3 Parallel Implementation

To measure how parallelism can improve the performances, in this project we propose two parallel implementations: the first one make use of a multi-platform API, known as *OpenMP*, and the second one make use of *POSIX Thread* library.

3.3.1 OpenMP

OpenMP (Open Multi-Processing) [2] is an API that supports multi-platform shared-memory multiprocessing programming in many different languages: in our case we used C++.
It is an implementation of multi-threading, a method of parallelization where a primary thread forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently and, by default, each thread executes the parallelized section of code independently. The section of code that is meant to run in parallel is properly marked with a compiler directive that will cause the threads to fork before the section is executed.
The module *omp_decryption.h* consists of two functions, as the sequential one: openMPDecryption and testOMP.
As suggested by the name, **openMPDecryption** is the function responsible to decrypt the password in a parallel way. The main benefit of using OpenMP is that it does not require restructuring the serial program, but it is only needs to add compiler directives to reconstruct the serial program into a parallel one.
As we can see, the section of code we want to run in parallel is marked by the OpenMP directives **#pragma omp parallel** and **#pragma omp for**, which is where

we compare the encrypted password with each password contained in the bag of words that we properly crypt using the function *crypt_r()*. More in details, thanks to the first directive, we specify the number of threads that form a team, created to execute the parallel region in parallel, and a list of variables that need to be shared among the threads, like, in this case, the boolean variable *found*. This variable, unlike the others, needs to be shared because indicates when one thread finds out the password and in this way the other threads can terminate early and save computation time.
The others variables could be shared but it isn't a forced choice; they can be also private because, as we previously said, the bag of words is a list of unique passwords and so we can have truly independent executions of the brute force algorithm, dividing the bag of words in chunks: this is an implementation of data parallelism.

```
string openMPDecryption(const char
    *encrypted_password, const char *salt,
    vector<string> bow, int num_threads) {

   string pwd;
   struct crypt_data data;

   data.initialized = 0;
   bool found = false;
   unsigned int bow_length = bow.size();

   #pragma omp parallel
       num_threads(num_threads) default(none)
       shared(bow_length, bow, found, salt,
       encrypted_password, pwd)
   {
      auto *structure = new crypt_data;
      structure->initialized = 0;
      #pragma omp for schedule(static)
      for (unsigned int i = 0; i <
          bow_length; i++) {
         if(!found)
            if
                (strcmp(crypt_r(bow[i].c_str(),
                salt, structure),
                encrypted_password) == 0) {
               pwd = bow[i];
               found = true;
            }
         else
            i = bow_length + 1;
      }
   }
   if(found)
      return pwd;

   return "Password Not Found!";
}
```

The second function in the module, **testOMP** is concerned with executing the tests, as for the sequential implementation. Even in this case, we discuss about it in section [4] and [5].

3.3.2 PThreads

POSIX Threads [3], commonly known as pthreads, is an execution model that exists independently from a language. It allows a program to control multiple different flows of work that overlap in time, where, in particular, each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.
As the previous ones, also the last module of this project, *pthread_decryption.h*, contains two functions: partialDecrypt and testPThread.
Even in this case the idea is to use data parallelizzation, that is split the bag of words in a number of parts depending on threads' number (meaning that with 2 thread, for example, the bag of words is split about in half). In this way, each thread managed only a piece of the bag of words and, as soon as a thread find out the password, it set the shared variable ***found*** to true, so that each other thread stops the research of the password (by hypothesis the bag of words is made up by unique words).
The shared variable could, in this case, lead to data racing problems: for example, when a thread finds out the password, it proceeds to modify the variable ***found*** and it could happen that another thread, during a check, tries to read the variable in an inconsistent state; a **mutex** would solve this problem but it would add complexity to the code so it is not really necessary.
The operations of writing and reading on a single memory location, in modern CPUs, are atomic operations, it is therefore very difficult to obtain inconsistent values; moreover, in the worst case, a thread (and only one, since the bag of words is a list of unique words) will modify the ***found*** variable and the rest of the threads, even reading an inconsistent value, will only be able to terminate or, at most, perform an additional iteration.
The presence of the struct *arguments* in this module is due to the fact that in calling of the function *pthread_create* (as shown later) can be passed only a single argument to the start routine.
On the other hand, the function **testPThread** is the one that manage the tests. In order to do this, clearly we first need to create the threads' team. Once the threads' numbers has been established, the bag of words needs to be properly split among them, as explained before. As

shown by the code below, the variable *chunk_length* represents the number of parts to divide the bag of words into. After that, we can create each single thread, assigning to each a specific chunk of bag of words and, clearly, the encrypted password to find out. Each thread is created through the method *pthread_create*, which indeed is a function that starts a new thread in the calling process, then the newly created thread starts the execution by invoking the *start_routine()*, which in our case is *partialDecrypt*, a function with the same behavior and logic as the counterpart of **OpenMP** one. Then, to create a **barrier** and wait the termination of all threads, we call the function *pthread_join()* on each thread. As usual, we discuss more in details about the tests implemented and the results obtained in sections [4] and [5].

```
int chunk_length =
    floor(bow.size()/thread_num);
for(int i = 0; i < thread_num - 1; i++) {
   //threads creation (assigning a chunk of
       bow to each thread)
   arguments[i].bow = &bow;

               [...]

   arguments[i].found = &found;
   if(pthread_create(&threads[i], NULL,
       partialDecrypt, (void*)&arguments[i])
       != 0)
      cout<<"Error in creation of thread
          "<<i<<endl;
}
arguments[thread_num-1].bow = &bow;

          [...]

arguments[thread_num-1].found = &found;
if(pthread_create(&threads[thread_num-1],
    NULL, partialDecrypt,
    (void*)&arguments[thread_num-1]) != 0){
   cout<< "Error in creation of thread
       "<<thread_num<<" (last thread)"<<endl;
}

// Barrier
for(auto thread:threads)
   pthread_join(thread, NULL);
```

4 Tests

All of the following tests are executed on a computer with the following specifications:

- OS: Ubuntu 22.04 LTS
- CPU: Processore Intel(R) Core(TM) i7-7700HQ @2.80GHz(TBoost 3.80GHz) Quad-Core
- GPU: GeForce MX 150
- RAM: 12GB
- MOBO: Asus X580VN

Tests consider mean, maximum and minimum execution times that are calculated by repeating the password search 100 times for each implementation proposed.
For these tests we decide to choose a particular password "**BertaGUI**" (and also a particular salt), which is positioned differently in each test execution: more in details, at each execution it is placed further and further away from the beginning of the bag of words and in one case is not present at all (worst case). In this way, we can consider the different performances based on the password position in the bag of words.
Regarding the parallel implementations, we make different tests even based on threads' number.

5 Results

In this section we discuss about the results achieved by the tests executions.

5.1 Sequential Implementation Results

In the case of sequential tests, we obtained the following results, in terms of execution time:

- Mean time: **0.736 s**
- Minimum time: **1.421e-05 s**
- Maximum time: **1.505 s**

The minimum time corresponds to the best case for sequential implementation, that is when the password is the first word in the bag of words; on the contrary, the maximum time is obtained in the worst scenario for sequential implementation, when the password we want to decrypt is not in the bag of words.
The computational complexity of the algorithm is $O(n)$, where n is the number of words to encrypt, so the mean execution time scales linearly with the bag of words size.

5.2 Parallel Implementation Results

About the parallel implementation, as expected, the results are significantly better. As we can see in the figure above [2], even with only two threads, the average execution time is lower than the one of the sequential implementation: in both parallel implementation, indeed, is around **0.4 seconds**, against the 0.74 seconds of the sequential

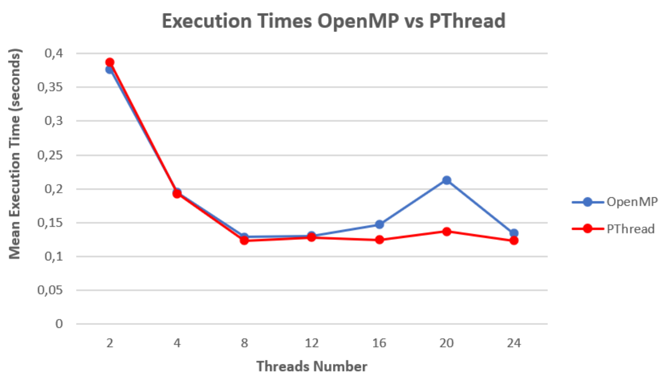

Figure 2: Comparison of execution times between the parallelization with OpenMP and PThread.

one. What we can also gather from the graph, is that increasing the number of threads, the mean execution times become lower and lower, until they achieve a sort of stabilization (which in both cases happens with 8 threads), plateau certainly due to the hardware used, the processor has in fact 8 logical threads. At least, we can see that in both cases the minimum average execution time obtained is the one corresponding to the parallelization with 8 threads (**0.128598 seconds** for OpenMP and **0.123179 seconds** with PThread): so the best result achieved with these tests is the parallel implementation using PThread with 8 threads.

The figure [3] represents the comparison between **speedup** in case of parallelization with OpenMP and PThread. The speedup is obtained as $S_P = t_S/t_P$, where P is the number of processors, t_S is the completion time of the sequential algorithm and t_P is the completion time of the parallel algorithm. We can notice that up to 4 threads the speedup is linear (Quad-Core CPU).

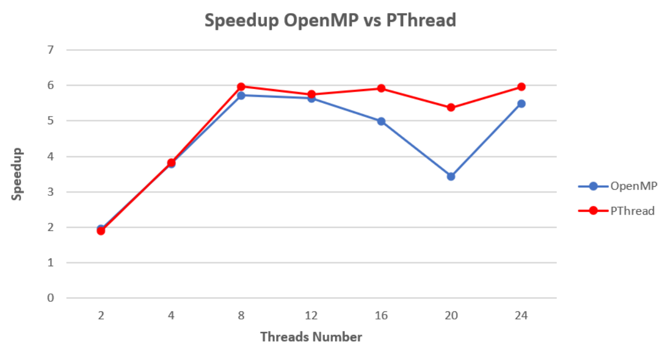

Figure 3: Comparison of speedup between the parallelization with OpenMP and PThread.

6 Conclusion

Without any surprise, both parallel approaches improve the performances in terms of execution time, especially when the password to decrypt in positioned at the bottom of the bag of words, in fact the computational complexity of the algorithm decreases to $O(n/k)$ where k is the threads' number. Even if both the parallel implementations achieved great results, the approach with PThread is slightly better, due to the fact that it provides more flexibility to the programmer, who can easily decide how to split the bag of words (even if the most reasonable approach is to give to each thread a piece of bag of words of approximately the same length) and in the thread creation and management; instead, with OpenMP even if the main benefit is that it does not require restructuring the serial program, but only to add the compiler directives, in this way it doesn't give to the programmer much flexibility, that in some cases could be useful.

References

[1] DES. `https://www.tutorialspoint.com/cryptography/data_encryption_standard.htm`.

[2] OpenMP. `https://www.openmp.org/specifications`.

[3] POSIX Threads. `https://www.bogotobogo.com/cplusplus/multithreading_pthread.php`.

[4] Daniel Miessler. SecLists. `https://github.com/danielmiessler/SecLists/tree/master/Passwords`.