TDT4200 PS 1 - Password Cracker (MPI)

Håvard O. Nordstrand

Due Friday, 10pm, Sep 11, 2020

Introduction

In this assignment you will parallelize a provided simple password cracker using OpenMPI.

Background

Passwords are strings of characters typically used to authenticate users. The simplest mechanism for this is simply storing the user's password in plaintext, and then comparing that stored password to the one provided by a user attempting to authenticate itself as that user. This isn't considered secure as an adversary gaining access to the system would be able to steal all stored passwords, and then try to gain access to the users' accounts on other systems where they're likely to use the same password. To avoid storing the passwords in the vulnerable state of plaintext, a cryptographic hash function is used to securely transform the plaintext password one-way to a constant-length string of seemingly random characters. For strong cryptographic hash functions, the most feasible way of finding the original password is trying to hash a whole lot of different text strings and trying to find a resulting hash that matches the hash of the original password, known as a "brute-force attack" or "exhaustive key search".

Common cryptographic hash functions include MD5, SHA1, SHA256 and SHA512. As a desirable property of cryptographic hash functions for passwords (often called "password hash functions") is that they're slow to make brute-force attacks less feasible, these mentioned algorithms (which are fast) are run through many iterations (thousands, often). This is how e.g. the password hash functions "MD5Crypt" and "SHA512Crypt" which are used in Linux are designed, which we will be using in this assignment. Another important mechanism to mention briefly is known as "salting", where a random string of data is concatenated with the plaintext password before being hashed. The "salt" is stored in plaintext together with the password hash. The salt is randomly generated and different for all users on one system and different for all systems for the same user, such that users with the same password should produce different hashes due to the different salt. This is to avoid seeing which users are using the same password, as well as defeating an attack known as a "rainbow table attack" where an adversary can pre-compute a large set of passwords and then simply compare the hashes without performing the time-consuming hashing operation.

Modern Linux systems store password hashes as well as some other user data for local users in the "/etc/shadow" file (see the provided shadow files). Each line consists of a set of fields which describes a single local user, where the first field is the username and the second field is the password field. The password field typically contains an invalid character like "*" to indicate that the user should not be able to log in with a password, or a set of fields separated by "$", where typically the first field is the algorithm (password hashing function) used, the second field is the salt and the last field is the hash. Algorithm 1 means it's based on MD5, 5 means it's based on SHA256 and 6 means it's based on SHA512. While the provided password cracker already handles reading the shadow file and handling the password field, you should have a basic understanding of what is happening.

Requirements

- Linux (e.g. Ubuntu)
- C compiler and linker
- Make
- OpenMPI toolkit

On Ubuntu, you can install everything using this command:

```
sudo apt update
sudo apt install -y build-essential openmpi-bin openmpi-doc libopenmpi-dev
```

Provided Files

- This assignment paper.
- A serial password cracker implemented in C, using the "crypt" function from the GNU C Library (aka "glibc") to hash passwords.
- A set of shadow files containing passwords to be cracked. Their names tell you what kinds of passwords they are. Each line is an entry for a user, where the second field is the encrypted password field. The "passwords.txt" file contains the original passwords for each variation, so you can check that your results are correct.
- A set of dictionary files containing symbols (words or characters) to try while cracking. Each line is a different symbol, which may be combined into multi-symbol strings according to the length and separator parameters provided to the application.

Usage

The password cracker generates passwords to compare against by giving it a dictionary of symbols, a max length of how many symbols passwords may contain and a separator to separate the symbols. To test for random alphanumeric passwords, you would give it a dictionary containing only single numbers and letters ("dict/alnum.txt"), no separator (the default) and a max length equal to the number of characters. To test for space-separated words, you would give it a dictionary of words ("dict/12dicts/2of5core.txt"), space as the separator and the number of words as the max length. To test for common passwords, simply give it a dictionary of common passwords ("seclists/10-million-password-list-top-10000.txt") and length 1 (the default). This simple password cracker does not support features like adding numbers at the end (like "password2") or mutating letters (like "P422w0Rd"), which is common for real passwords.

- Build application (using "mpicc")

  ```
  make build
  ```

- Remove temporary files (including binary)

  ```
  make clean
  ```

- Run OpenMPI application

  ```
  mpirun -n <process_count> [--oversubscribe] <binary> [options]
  ```

If it complains about "not enough slots available", specify "–oversubscribe" for "mpirun". If you run as root for some reason (strongly discouraged), specify "–allow-run-as-root" for "mpirun".

- Run the password cracker:

```
mpirun [...] crack [-q] [-v] -i <shadow-file> -d <dict-file> [-l
↪  <max-length>] [-s <separator>] [-o <result-file>]
```

  - -q: Don't print results to console.
  - -v: Print all password probes tried to console.
  - -i <shadow-file> (required): The shadow file with password hashes to crack.
  - -d <dict-file> (required): The dictionary file to use, consisting of one symbol per line.
  - -l <max-length>: The maximum number of symbols to combine to password probes. Defaults to 1.
  - -s <separator>: The text string to use to separate symbols when max length is more than one. Defaults to the empty string. Note: Space must be quoted, like "-s " "".
  - -o <result-file>: The file to write results to.

Examples

```
# Example (SHA512, 4 digits) (fast)
mpirun [...] crack -i data/shadow/sha512-4num -d data/dict/num.txt -l 4
# Example (SHA512, 2 alphanumeric characters) (fast)
mpirun [...] crack -i data/shadow/sha512-2alnum -d data/dict/alnum.txt -l 2
# Example (MD5, 1 simple word) (very fast)
mpirun [...] crack -i data/shadow/sha512-1word -d data/dict/12dicts/2of5core.txt
# Example (SHA512, 1 simple word) (fast)
mpirun [...] crack -i data/shadow/sha512-1word -d data/dict/12dicts/2of5core.txt
# Example (SHA512, 2 simple words, space-separated) (slow)
mpirun [...] crack -i data/shadow/sha512-2word -d data/dict/12dicts/2of5core.txt
↪  -l2 -s" "
# Example (SHA512, 1 common password) (fast)
mpirun [...] crack -i data/shadow/sha512-common -d
↪  data/dict/seclists/10-million-password-list-top-10000.txt
```

Example Output

```
mpirun -n 8 --oversubscribe crack -i data/shadow/sha512-1word -d
↪  data/dict/12dicts/2of5core.txt

Workers: 8
Max symbols: 1
Symbol separator: ""
Shadow file: data/shadow/sha512-1word
Dictionary file: data/dict/12dicts/2of5core.txt
```

```
Read 4690 words from dictionary file.

Entries:
user="daemon" alg="UNKNOWN" status="SKIP" duration="0.000000s" attempts="0"
attempts_per_second="-nan" password=""
[...]
user="user1" alg="SHA512" status="SUCCESS" duration="0.322231s" attempts="635"
attempts_per_second="1970.635483" password="chair"
user="user2" alg="SHA512" status="SUCCESS" duration="1.094056s" attempts="2313"
attempts_per_second="2114.151119" password="lamp"
user="user3" alg="SHA512" status="SUCCESS" duration="0.843840s" attempts="1812"
attempts_per_second="2147.327172" password="glass"
user="user4" alg="SHA512" status="SUCCESS" duration="0.749800s" attempts="1660"
attempts_per_second="2213.923940" password="floor"

Overview:
Total duration: 3.010s
Total attempts: 6420
Total attempts per second: 2132.942
Skipped: 18
Successful: 4
Failed: 0
```

Tasks

Task 1: Get Familiar with the Serial Application

a) Build and run the application without using MPI for the shadow files "shadow/sha512-1word" (1 word), "shadow/sha512-2alnum" (2 alphanumeric characters) and "shadow/sha512-common" (1 common password) with the correct command line arguments (see usage section). All three should succeed within a short time. Take note of the total duration for "shadow/sha512-1word", so it can be compared with the parallelized version later.

b) Based on the total runtime for "shadow/sha512-2alnum" (2 alphanumeric characters), consider how long it would take to brute force a password of 8 alphanumeric characters (the number of symbols in the dictionary file is printed to the output). Also, consider how long it would take for a four word password based on the runtime for "shadow/sha512-1word" (1 word).

c) Look through the source code to get an overview of how it's structured. Look through the makefile as well to see how the MPI application is built (it's already set up to support the parallel MPI version).

Task 2: Add MPI to the Serial Application

For this task, you'll begin turning the simple serial application into an MPI application, but still letting only one process do all the work. We'll treat the process with rank 0 as the master/root process and all other processes as replicas. Only the master should handle the command line parameters, including reading the input files. The replicas don't need to do any password cracking yet, they can just print their own rank and exit so you know they're running using the correct rank. If the output is duplicated, indicating all processes are doing the same thing, then you're doing it wrong. Remember to both initialize and finalize MPI.

In addition, you must implement the time measurement for how long it takes to finish trying to crack a single entry, using "MPI_Wtime". This is mostly implemented already, you only need to modify it slightly.

Run it using MPI and more than one process (see usage). Make sure it still returns the correct results (as can be seen in "shadow/passwords.txt"). The total runtime should be roughly the same as for the non-MPI serial version.

Task 3: Parallelize the Serial Application

Now it's time to continue what was started in task 2 and distribute the work among both the master and the replicas. The master's role will be to prepare work for all ranks, distribute them, processing its own work, collect the results from all ranks, check the results and repeating until either a matching probe has been found or until the search space has been exhausted. The replicas should enter an infinite loop where they receive work, process it, and send the result back. The replicas will need to handle two special cases when structured like this. The first case is when the search space is nearly exhausted and there's not enough work for all processes. The second case is when there's no more shadow file entries to crack. You can use empty jobs with "ACTION_SKIP" and "ACTION_STOP", respectively, to handle these two cases. When the replicas handle these jobs, their results should have status "STATUS_SKIP". For the sake of correctness wrt. a deterministic attempt counter, when checking the results and increasing the attempt counter, don't count results with "STATUS_SKIP" or any results after a rank with "STATUS_SUCCESS" (simulate that jobs are processed sequentially).

Now run it using MPI. As this workload is embarrassingly parallel, the speedup should be nearly linear with the number of processes.

Hints: Only the master process (rank 0) should handle the command line options and read the input files. You're expected to use MPI's scatter/gather for distributing the jobs and collecting the results. You only need to distribute one job for each rank at the time. While not needed in this application, you can use MPI's broadcast to distribute e.g. the options loaded by the master process to replica processes. Due to the asymmetry between the master and replicas in this application, it may be simpler to structure the code for the master and for the replicas in separate functions instead of trying to stash everything into common functions. But make sure that all MPI communication functions still happen in exactly the same order for all ranks. If your program stalls for no apparent reason while running, it may be because of mismatched synchronous MPI calls.

Deliverables

The assignment must be delivered as a ZIP file and submitted in time on blackboard. The code must be documented and in a runnable state. Do not use any libraries apart from standard libraries or those provided. Do not change the calling conventions (parameters) of the application.

NB!: Deliver your code in the folder structure that is provided to you. Include at least every file that you have changed. Your application must stay compatible with the provided Makefiles or CMake files. **Do not include any object, input or output files**.

This assignment is mandatory and will be graded pass/fail.