Parallel Computing: Homework 6

Introduction

During this last lab, you will work on parallellising and optimising a password guessing parallel program. We will provide you with a basic implementation of such a program that you are expected to parallelise using MPI. In addition, you are expected to develop heuristics to create an effective password guessing strategy.

Background

On Unix systems, user account information is stored in two separate files: `/etc/passwd` and `/etc/shadow`. The passwd file contains information such as the user name, home directory, and preferred shell, but counterintuitively, it does not contain any information about the password. The password hash is stored in `/etc/shadow`, together with information about which hashing algorithm and which salt[1] were used.

The format of `/etc/passwd` is as follows:

```
jde100:x:1001:1002:John Doe,,,:/home/jde100:/bin/bash
```

`jde100`	is the username.
`x`	indicates that the password is stored in `/etc/shadow` (it *used* to be stored here).
`1001`	is the user ID (UID).
`1002`	is the group ID (GID).
`John Doe,,,`	is the comment field (`GECOS`), typically a comma-separated list with as first item the user's full name, and after that other information like phone numbers.
`/home/jde100`	is the user's home directory.
`/bin/bash`	is the user's preferred shell.

The format of `/etc/shadow` is as follows:

```
jde100:$1$M9$80ZJhwG6Ngh1lLoKg5MdC1:14486:1:2:3:4
```

`jde100`	is again the username.
`$1$M9$...`	is the password hash. There are three sections to it, separated by `$`. The first section (`1`) is the algorithm used (MD5); the second section is the salt used (`M9`), and the last section is the hashed password in base64 format. In this case, it is the hashed version of `iloveyou`.
`14486`	is day the password was last changed; in days since Jan 1st, 1970 (the Unix epoch).
`1`	is the minimum number of days required between password changes.
`2`	is the maximum number of days a password is valid.
`3`	is the number of days before the expiry date of the password that a warning message should be shown about changing the password.
`4`	is the expiration date of the account.

[1] `https://www.hypr.com/salt/`

Objective

The objective of this assignment is to guess passwords as efficiently and effectively as possible. This has two aspects: on the one hand your algorithm needs to be efficient and parallellised, but on the other hand your password guessing strategy will greatly influence your results: simply starting with `a`, `b`, ... probably won't get you very far.

Your final submitted code will get **exactly 10 minutes** of runtime on Peregrine, on **24 cores** and **6GB of memory**, to produce as many correct passwords as possible. The performance of your implementation in this test will influence your grade. Additionally, there is a small competitive factor: the relative ranking of your implementation as compared to that of your classmates might award you a *bonus* point (so the maximum grade for this assignment is 11).

Execution

On Nestor, we have already provided a fully-functional password guesser that handles everything from reading the shadow and passwd files, to computing hashes and printing the results. However, this implementation is (of course) single-threaded and is not very intelligent about its guessing strategy; you can definitely do better.

Please make sure to carefully read the comments in the provided header files and in **`guessword.c`**, outlining all data structures and functionality provided. In particular, the function **`parseInput()`** will read and parse the shadow and passwd files, and put all data in appropriate `structs`. The function **`tryPasswords()`** will hash all the given passwords in a list and try to match them with the 'target' list of hashes - if a result is found, the username and password combination will be printed.

The code comes with an accompanying **`Makefile`**, which makes it easy to start going with the code: after installing OpenMPI or MPICH, simply run `make` to (re-)compile your code, after which you should have an executable **`./guessword`**. This executable expects two arguments: the path to a password file and the path to a shadow file, in that order. Contrary to previous assignments, the number of threads/cores is not passed to the program directly, as this is handled by MPI.

Parallelisation

You are expected to parallellise your code using the most suitable MPI functions. Before getting to work, think thoroughly about the parallellisation approach you will be using. How will you split the work? How will you get that work to the different processes? How can you ensure all processes will continue working as much as possible? How will you ensure each process is doing the most effective work at any given moment? How will you deal with the potentially large amounts of data? There are just some of the questions you should ask yourself before starting to work on a particular implementation, to prevent yourself from getting stuck after many hours of hard work.

The provided **`Makefile`** already compiles your code using **`mpicc`**, so running your code in an MPI environment is as simple as running `mpiexec -n [threads] ./guessword [passwd] [shadow]`. As always, the number of processes running is available to you through **`MPI_Comm_size()`**. The function **`parseInput()`** has a parameter that allows you to mute the debugging output. Note that the framework code does not require changes for a parallel implementation - in particular the printing to **`stdout`** by **`tryPasswords()`** should continue to function without issues using MPI. [2]

[2] One might expect issues where different 'threads' are printing at the same time, causing the output to be garbled, but from our testing this seems to be prevented by MPI.

Guessing Strategy

The second big part of this assignment is to develop and implement an effective guessing strategy. The search space of all possible passwords is gigantic, but user's passwords are usually not fully random; they will have patterns that you can exploit to try strings with a higher probability of being the correct password. Developing a guessing strategy then means: determining what strings to test and in what order.

For the purposes of this assignment, all input files will be generated using the same patterns. The actual passwords will be different, but the underlying patterns are not. In addition to the guesser implementation, we provide you with a set of sample files (passwd, shadow, and accompanying plaintext passwords) for you to analyse and test your code on.

Given the set of sample files, you should be able to observe patterns in the plaintext passwords that you can exploit in your implementation. For example: some users might use their username, appended with `00` as their password. If you spot that as a pattern, you should write code to test passwords of that form.

To simplify your coding work, the provided implementation comes with a set of functions that implement a range of manipulation operations that might be useful to you. One of them is a function to substitute arbitrary characters with strings - you can use it to implement this 'appending `00`' by substituting `'\0'` (string end marker) by `"00"`. This particular strategy is already implemented in the provided code as an example.

In addition to sample data files, we also provide the top 250 most popular passwords, and a set of ebooks in `.txt` form. You can use these as 'dictionaries' for passwords to try. You are allowed to preprocess these or any other files into some other form that saves computation time. The provided implementation comes with a function to read a file with simple strings on each line - which is how this list of most popular passwords is formatted.

Reminder: Your code can (and should!) run as long as possible - you are allowed to fill 10 minutes of runtime on Peregrine after all. We will just stop your program after this time; you do not have to worry about stopping in time.
Hint: Precomputing a bunch of passwords and storing the hashes in files won't work, since the grading input sets will use a different salt - that will influence the final hash.
Hint: We recommend you just use the provided dictionary files.

Deliverables

We ask you to submit your code to Themis, just like previous weeks, which will only perform some basic sanity checks. It will test some trivial inputs, like just the password `password`, or `123456` (both are in the top 250), and allow your program to run for 6 seconds on 2 or 4 cores (with 256MB memory per core available, just like we allow on Peregrine). So if you manage to guess these trivial inputs within 6 seconds, you will pass Themis. The test on Themis mainly serves to check whether your code compiles properly and doesn't crash.

Your code does not need to do anything specific to deal with these timing limitations - Themis and Peregrine will just kill your program after the specified timeout. To prevent any losses due to buffering of the output, the provided implementation already calls `fflush(stdout)` after every found password to ensure the password is registered immediately.

Furthermore, generally you do not need to worry about printing the same password twice - Themis will filter out all unique results and grade only those. However, if you do print a password multiple times, you have probably tried and hashed it twice, which might not be the most efficient strategy. Printing excessive amounts of output will cause Themis to stop your code.

In addition to your program code, you can submit any other input (text) files that will be used by your code. By default, Themis will put all uploaded files in the same folder; if your code expects these auxiliary files to be in a different folder (like the provided code does), please put all the files you want to submit in a `tar` or `zip` archive with the proper directories - Themis will then transfer the folder structure to the execution environment as-is. The limit on file uploads is 10MB. Note once again that the salt used for our grading inputs will be different, so precomputing the hashes will not work!

Themis will compile your code through your `Makefile` by running `make`, and will run your code through `mpiexec -n N ./guessword ...`. You are allowed to change any parts of the provided implementation (or make a completely new one), as long as it compiles and runs this way.

Lastly, we ask you to submit (on Themis) a *short* and **concise** report (PDF, no more than **1** page) outlining your approach to parallellisation and your guessing strategy. You do not have to perform a performance analysis this time - the efficiency is fully determined by the performance in our test on Peregrine.

Grading

Similar to the previous labs, we will grade on the criteria *correctness*, *efficiency*, and *code style*:

Correctness corresponds to passing test cases on Themis. If you did not pass all test cases, you might still get partial poitns. A manual code check might override the results from Themis, e.g. if you hardcoded answers or circumvented explicit assignment instructions.

Efficiency corresponds to the number of passwords you manage to guess in the 10 minutes run on Peregrine. This score will be purely individual. In addition, **up to 1 bonus point** can be earned through your ranking among your classmates.

Code style includes, but is not limited to: sufficient documentation, clear naming of variables, proper code structuring, proper code formatting, etc. *This is not an exhaustive list!* Particularly sophisticated implementations might be awarded some bonus points.

Our grading will always be based on the **most recent** submission on Themis. Even though you might have submitted a 'better'(-performing) version before, we will only evaluate the most recent one.

You will have **two weeks** for this last programming assignment, and there will not be any exercises on Nestor.