Final Project
CS5234 Advanced Parallel Computation.
Submission for Rishabh Rastogi

Introduction:

The project proposal was accepted by the professor as is. The purpose of this project is to implement a brute force password cracker software, using MPI and OpenMp in clustered environment.

In Ubuntu, the password for a user is encrypted using a cryptographic hash function that returns a fixed-size alphanumeric string. It is quite difficult to find the password given only the hashed text, computationally speaking. Historically, only a cryptographic hash function of the password was stored on a system, but over time, additional safeguards were developed to protect against duplicate or common passwords being identifiable (as their hashes are identical). Salting is one such protection. In cryptography, a salt is random data that is used as an extra input to a one-way hash function, a password or passphrase. Salts are used to safeguard passwords in storage.

The OS encrypts users' passwords using the SHA-512 algorithm, and stores the hash in the shadow file. It is possible to parse this file and extract the hash. Using brute force, every possible combination of characters – within the password limitations – can be encrypted and the computed hash can be compared to the original hash. Brute force can crack the password eventually, but will take a long time and a lot of computational power to do so.

To improve upon its performance, I will have to parallelized the code and execute it in a clustered computing environment. This report will try to outline further on how the brute force algorithm will be implemented and parallelized.

Implementation involves several steps: creating multiple processes, parsing the shadow.txt file, dividing the work up, matching the password and terminating processes once password is cracked. I created a temporary "user" with the name "Project" present in the shadow.txt file on whose password I have been testing.

I tried our best to follow proper coding conventions, and to organize and comment our code properly. Please read the Readme file on instructions on how to run the code.

Password Cracker (implemented in Password_Crack.cpp file):

Brute force is applied to crack the password using MPI. Given an initial character (and the salt + hash of the specified username), all combinations possible with that character will be generated and checked on the fly. What this means is that rather than generating all combinations first and then checking all of them, I generate one possibility, then check it and then generate the next. This means that time and memory is not wasted on computing all combinations at once – moreover, if the password is cracked, the function can return right there instead of wasting resources on finding more combinations.

Given an initial character, and a maximum length, all combinations with that character are explored in this way. Every generated string is passed to the crypt function which contains the salt, and the resulting hash is compared to the salt + hash parameter. If they are found

equal, then the password is cracked, print and the main loop terminates. If the hashes are not equal, then the rightmost non-z character in the current string is incremented. This process continues until the string has only z-characters (save for the first character.) At this point, the current length is incremented and the string is reset to initial character + a's (up to the current length.). Once current length exceeds the maximum length, the outer length terminates. Whenever the password is cracked, **MPI_Send** is called to signal to the master process that password has been found (and sends the found password).

Work Modelling:

In order to divide the work among processes and in order to efficiently divide the work among the slaves, we "DIVIDE" a total of 26 alphabets (characters in our case by the number of processes. The letters will be shuffled when sent to slaves to ensure equal probability of getting the password or else the first letter would have always been checked first followed by the second letter. It is not a guaranteed performance increase but I thought it was a better approach. Now there are two possible outcomes of this situation:

1. Perfectly Divisible (No remainder): In this case we know that the alphabets can be equally divided over the number of processes. Hence, we do that. The quotient of the division is stored in the variable and the value that the quotient has is the number of alphabets assigned to each process. For example: If we have 26 processes, we know that it is perfectly divisible and the quotient is 1. Hence, one alphabet is assigned to each process.
2. Not Perfectly Divisible (Remainder exists): This is an interesting scenario. I thought why should master be sitting idle and making slaves do all the work. Hence, our logic works in this sense that first we divide 26 by the number of processes. The remainder that we get is stored into a variable and is the number of letters that the master will have to process. Master is assigned the specific alphabets and the remainder is subtracted from the total number of alphabets (variable). The new value will be totally divisible by the number of processes; hence, the above point is followed and slaves are assigned their tasks. For example: If the user gives 12 number of processes. We know that 26/12 gives us Quotient = 2 and Remainder = 2. Hence, 2 alphabets (a and b) are given to the master. Now we subtract the remainder from the total alphabets and the new total will become 24. We know that 24/12 = 2 with no remainder. Hence, each process is given equal number of letters (i.e., 2).

Final Project
CS5234 Advanced Parallel Computation.
Submission for Rishabh Rastogi

Parallel Processing:

MPI and openMP will **both** be used to create processes/threads in this program. **MPI** is utilized to create processes, and different code written for different processes on the basis of process rank. The master is responsible for getting the salt + hash of the specified username, distributing the work among slaves and sending the salt + hash and the allotted work to each slave. From here, two cases follow:

CASE 1: MASTER ASSIGNED SOME WORK

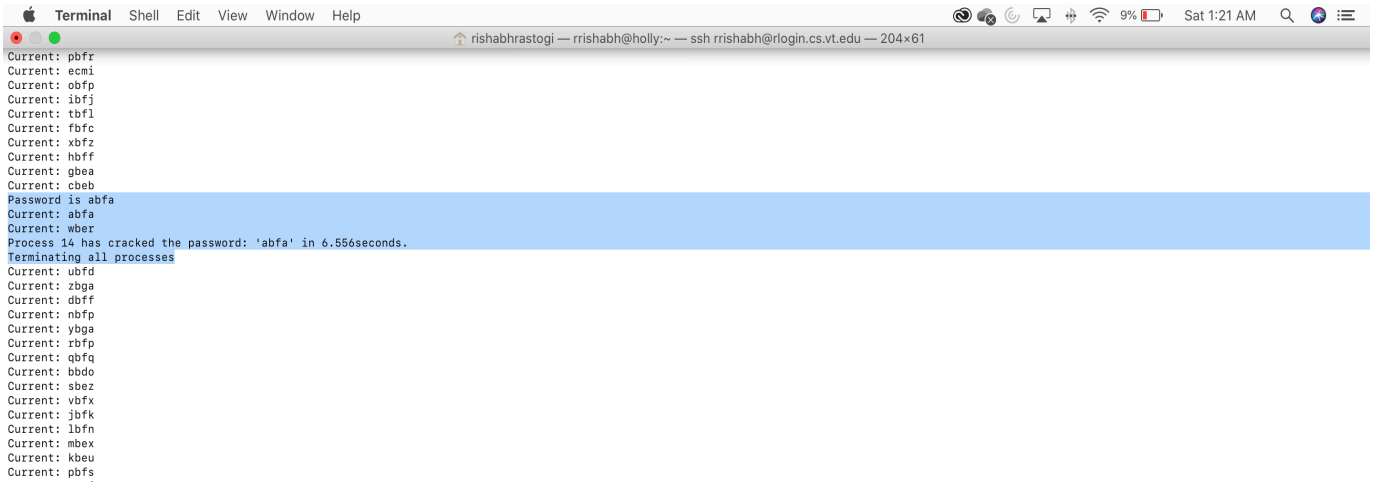
In this situation, the master has two tasks: to do its assigned work, and to wait for a message from any slave that the password has been found. **openMP** is used here to create two threads – one is searching for the password and the other is waiting to receive a message that the password has been found. Once password is cracked (whether by master or slave), the master receives it, prints it then calls MPI_Abort to terminate all processes.

CASE 2: MASTER NOT ASSIGNED ANY WORK

In this situation, the master has nothing to do except wait on any slave to send the message that the password has been cracked. A blocking MPI_Recv is called, and once the cracked password is received, MPI_Abort is called to terminate all the processes (as there is no need to search any further for the password.)

In any case, slave processes all have the same work. They are assigned their respective characters; they loop through them and try to crack the password. If cracked, they signal the master, send the password then terminate

Output:



Final Project
CS5234 Advanced Parallel Computation.
Submission for Rishabh Rastogi

Time taken: 6.556 seconds for parallel running of this code through MPI and OpenMP on a single machine.

This code can also be run on clustered computing environments and the running time reduces as the number of machine this is set up on increases. I checked the setup on two computers and running time decreased by half to 3.23 seconds.