END OF SEMESTER PROJECT

END OF SEMESTER PROJECT

CS-326: Parallel and Distributed Computing

Hassan Shahzad 18i-0441

Abeera Fatima 18i-0411

Sana Ali 18i-0439

Azka Khurram 18i-461

TABLE OF CONTENTS

INTRODUCTION

The purpose of this project was to implement a brute force password cracker in a distributed system, using MPI. In Ubuntu, the password for a user is encrypted using a cryptographic hash function that returns a fixed-size alphanumeric string. Computationally, it is quite difficult to find the password given only the hashed text.

The OS (Ubuntu in our case) encrypts users' passwords using the SHA-512 algorithm, and stores the hash in the /etc/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, we have parallelized the code and execute it in a clustered computing environment.

This report will detail further on how the brute force algorithm was implemented and parallelized.

Note: The cluster was created using different physical machines, rather than having multiple virtual machines on one physical machine.

IMPLEMENTATION

All of our implementation was done from scratch. No help has been taken from either the internet or any fellow batchmate. Only help in debugging was taken from our respected instructor.

Implementation involves several steps: creating multiple processes, parsing the shadow file, dividing the work up, looking for the password and terminating processes once password is cracked. We created a temporary user named "Project" on whose password we have been testing. Its password can easily be changed and tested.

We tried our best to follow proper coding conventions, and to organize and comment our code properly.

PASSWORD CRACKER

Brute force is applied to crack the password. Given an initial character (and the salt + hash of the specified username), all combinations possible with that character and generated and checked on the fly. What this means is that rather than generating all combinations first and then checking all of them, we 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 -> a, b, c, d….aa, ab, ac…..aaaaabbb, aaaaabbc, aaaaabbd….azzzzzzz. Every generated string is passed to the crypt function (with the salt), and the resulting hash is compared to the salt + hash parameter. If they are equal, then the password is cracked, printed 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 send the found password.)

WORK DISTRIBUTION

In order to divide the work among processes we brainstormed and came up with the solution that in order to efficiently divide the work among the slaves, what would be better than to use the actual DIVIDE function 😊. Hence, this logic was implemented. For this we know that there are a total of 26 alphabets (characters in our case). As we are already getting the number of processes as an argument, we will first divide the total alphabets (i.e., 26) by the number of processes. 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 13 processes, we know that it is perfectly divisible and the quotient is 2. Hence, two alphabets are assigned to each process.*
2. **Not Perfectly Divisible (Remainder exists):** This is an interesting scenario. We thought why should master be sitting idle and making slaves do all the work. Afterall, we do believe in equality… 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).*

Note: The letters are 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 we thought it was a better approach.

PARALLELIZATION

MPI and openMP are both being 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.

SCREENSHOTS

OUTPUT

Our password cracker runs perfectly and is cracking the password in the shortest execution time possible. Following is the output of our code when trying to crack the same password (i.e., **abfa)** sequentially and in a distributed system:

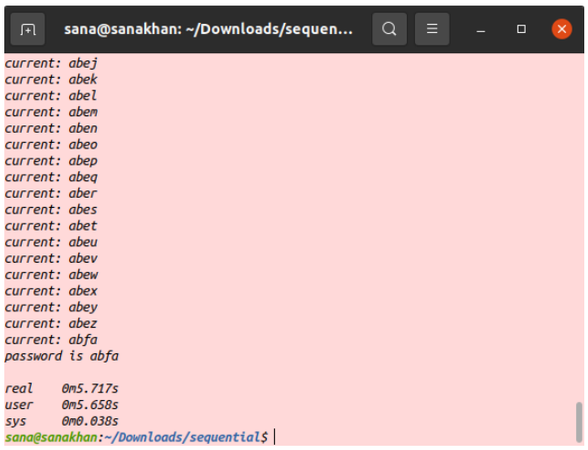

Fig 1.1: Code running sequentially

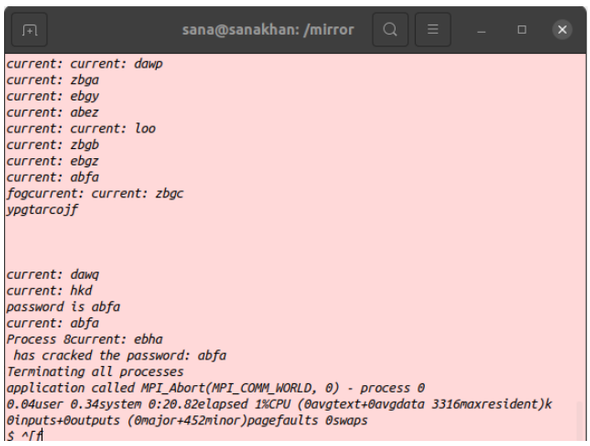

Fig 1.2: Code running in a distributed system

CODE

```
string get_salt_hash(const string file, const string username) {

    // this function reads the file (typically shadow.txt) and finds the value corresponding to the specified username
    // the value is then parsed to get the salt and hash (as one string) for this username
    // salt and hash are returned as one string

    string line, salt_hash;

    fstream infile;
    infile.open(file, ios::in);

    if (infile.is_open()) {
        while (getline(infile, line)) {
            if (strstr(line.c_str(), username.c_str())) {   // if line has specified username
                vector<string> tokens = split_string(line, ':');    // split by " : " character
                salt_hash = tokens[1];  // salt and hash are after first colon and before second
                break;
            }
        }

        return salt_hash;
    }

    else {
        cout << "Shadow file not opened :(\n";
        return "";
    }
}
```

Fig 2.1: Parsing shadow file to get the salt and hash of the password (for the specified username)

```
map<int, string> divide_alphabet(const int slave_procs) {

    // this function divides the alphabet characters and assigns certain number of characters to each process
    // also takes into account the case when alphabet is not perfectly divisible by number of slave processes
    // so may also assign master some characters
    // the distribution is stored in a map with (key, value) pair being (process rank, characters assigned)
    // rank 0 is master, rank 1 is slave 1, rank 2 is slave 2 and so on...

    map<int, string> distrib;
    string alphabet = "abcdefghijklmnopqrstuvwxyz";
    random_shuffle(alphabet.begin(), alphabet.end());

    // case 1: alphabet perfectly divisible by number of slaves
    if (alphabet.length() % slave_procs == 0) {
        int chunk = alphabet.length() / slave_procs;

        distrib[0] = "";    // no letters assigned to master (characters can be evenly distributed among the slaves), s

        int index = 0;
        for (int i = 1; i <= slave_procs; i++) {
            int start = index;

            string letters = alphabet.substr(start, chunk);

            distrib[i] = letters;   // assigning letters to that slave rank

            index += chunk;
        }

    }
```

Fig 2.2: Distributing the alphabet among the slave processes

```
    // case 2: alphabet not perfectly divisible, some characters will have to be allotted to master too
    else {
        int mod = alphabet.length() % slave_procs;

        distrib[0] = alphabet.substr(0,mod);

        int new_len = alphabet.length()-mod;
        int mod1 = new_len/slave_procs;

        int index = mod;

        for (int i = 1 ; i <= slave_procs; i++) {
            int start = index;

            string letters = alphabet.substr(start, mod1);

            distrib[i] = letters;

            index+=mod1;
        }
    }

    return distrib;
}
```

Fig 2.3: Distributing the alphabet among the master and slave processes

```
map<int, string> distrib = divide_alphabet(nprocs-1);   // getting the alphabet division per process (including master, if required)

// send salt_hash and the letters assigned to that process to every slave
for (int i = 1; i < nprocs; i++) {
    MPI_Send(salt_hash.c_str(), 200, MPI_CHAR, i, 100, MPI_COMM_WORLD); // send the salt + hash string to every slave
    MPI_Send(distrib[i].c_str(), 30, MPI_CHAR, i, 101, MPI_COMM_WORLD); // send every slave their allocated characters
}
```

Fig 2.4: Master using MPI to send each slave their assigned characters of the alphabet

```
// slave specific code
else {
    char salt_hash[200], letters[30];

    MPI_Recv(salt_hash, 200, MPI_CHAR, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  // recieve salt_hash (same for every slave)
    MPI_Recv(letters, 200, MPI_CHAR, 0, 101, MPI_COMM_WORLD, MPI_STATUS_IGNORE);    // recieve allotted characters (different for every slave)

    cout << "slave " << rank << ": " << letters << endl;

    sleep(2);   // this is just to ensure all slaves print their allotted characters then start execution (for neatness :) )

    // go through its assigned characters -> for every character, call the cracking function on it
    for (int i = 0; i < charToString(letters).length(); i++) {
        password_cracker(letters[i], charToString(salt_hash));
    }
}
```

Fig 2.5: Slaves using MPI to receive their allotted characters, then trying to crack password with them

```
// crypt the current string with constant salt
string crypted = crypt(current.c_str(), salt.c_str());

// compare the hash generated to the salt_hash
if (crypted == salt_hash) {
    found = true;
    cout << "password is " << current << endl;

    // signal master that password has been found (and send the password along)
    MPI_Send(current.c_str(), 10, MPI_CHAR, 0, 200, MPI_COMM_WORLD);
    break;
}
```

Fig 2.6: If password cracked, MPI used to signal the master process and to send the password to it

```
    // in this case, the characters have been divided equally among the slaves
    // master has nothing to do except wait for a slave to report password found
    else {
        char password[10];

        MPI_Status status;
        MPI_Recv(password, 10, MPI_CHAR, MPI_ANY_SOURCE, 200, MPI_COMM_WORLD, &status); // blocking recieve waiting for any process to report

        cout << "Process " << status.MPI_SOURCE << " has cracked the password: " << password << endl;
        cout << "Terminating all processes" << endl;

        MPI_Abort(MPI_COMM_WORLD, 0);   // terminates all MPI processes associated with the mentioned communicator (return value 0)
        // this is not a graceful way of exiting but will suffice for the functionality we require
    }
}
```

Fig 2.6: Master using MPI to await the cracked password from other processes

machinefile

```
1  192.168.6.11:2
2  192.168.6.48:2
3  192.168.6.192:2
4  192.168.6.13:2
5
```

Fig 2.7: The Ips of the physical machines that are used for clusters

EXECUTION VIDEO

Following is the link to the execution video of our code:

https://drive.google.com/file/d/1kIaIWt-kcW0ZcR0Dulmww8_5xd7wxXC-/view?usp=sharing

DEMONSTRATION & ANALYSIS OF THE RESULT

Following graphs show the difference in the time of execution and speedup of the implementation:

1: PERFORMANCE GRAPH

This graph shows the comparison in the execution times of both sequential and distributed codes while cracking the same password:

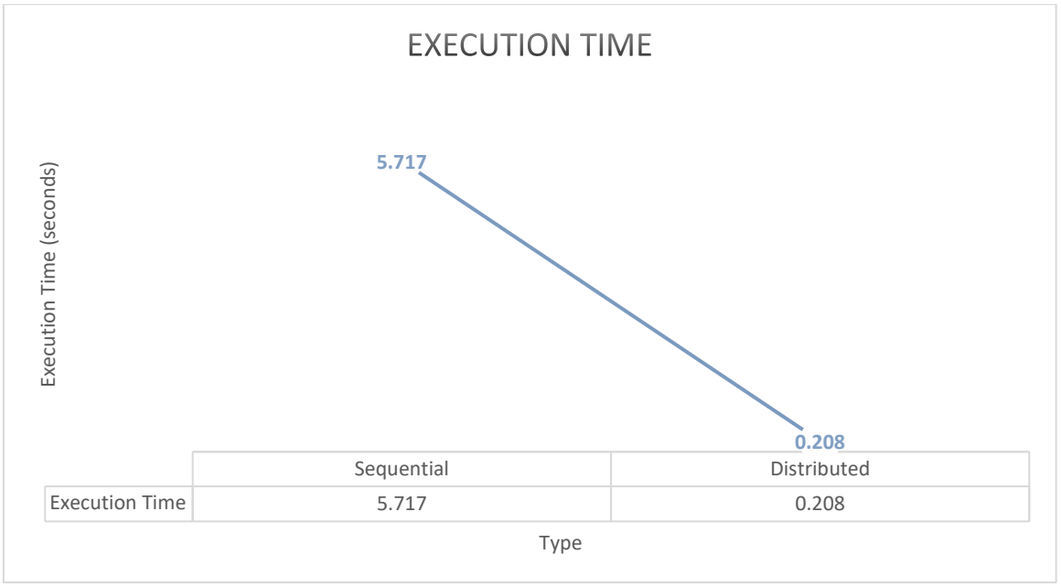

Fig 3.1: Performance graph

As can be seen from the above graph, there is a huge difference in the execution time of the distributed approach. Our distributed approach is almost 5.5 seconds faster meaning almost **96.4% faster**.

2: SPEEDUP GRAPH

This graph shows the comparison in the performance of both sequential and distributed codes while cracking the same password:

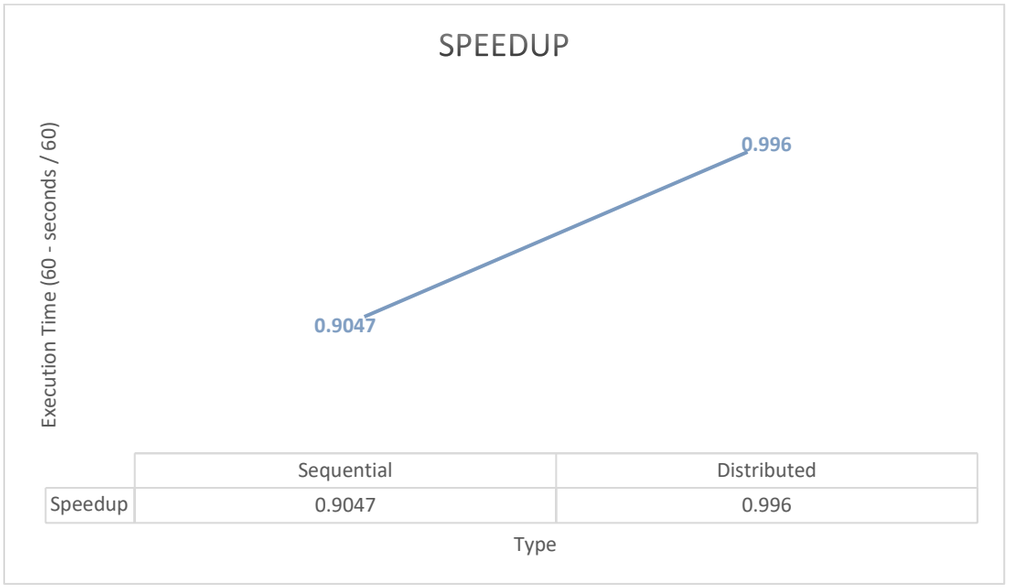

Fig 3.1: Speedup graph

As can be seen from the above graph, there is a huge difference in the speedup of the distributed approach. Our distributed has amazingly better performance than our sequential approach.