CS 5500 Project Report: MPI Password Cracker

Andrew Knoblach
AMKnoblach@gmail.com

Carter McGee
cartermcgee7@gmail.com

April 8, 2021

Project Description

The goal of this project was to create a simple brute force password cracker. The reason we chose this is because password cracking shows up on the list of Embarrassingly Parallel problems. The idea was simple, take a hashed password and try every single combination until you find a string that when hashed matches your original hash.

For this project we choose to use an MD5 hash as it is a common hash that isn't too computationally expensive. Although MD5 is no longer considered a secure hash (and hasn't been for a while) we figured it was good for our proof of concept.

Overview of Code

Our password cracker program is broken up into several bits. The first bit is concerned with gathering information and sending it out to all the processes. In this part we gather:

- The length of the password
- If the password space contains lowercase letters
- If the password space contains uppercase letters
- If the password space contains numbers
- The hash to be broken

All of these settings are gathered by rank0 then using `MPI_Bcast` the settings are sent to all other processes and `MPI_Send` is used to send the hash to all other processes.

After this section the password spaces are created according to what was dictated in the settings. The password spaces use the ASCII values for each character and then are appended to a password

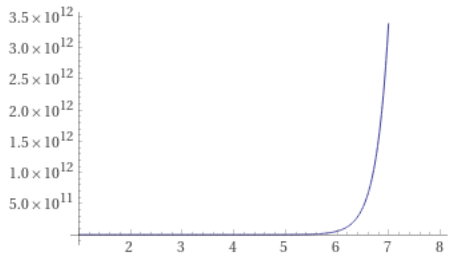
space vector. For this project we only included `a-zA-Z0-9` the math behind this decision is covered in the next section.

Next the actual password cracking begins. Using a recursive function each possible string of length n is generated within the given password space. Once a string is generated it is then hashed. The hash is then compared to the original hash provided by the user. If the hash matches the program reports the string that gave a match as well as how long the crack function was running to find this string.

Lastly the program will inform all other processes that the correct hash has been found and that it is time to terminate.

Testing

First, we would like to explain why we set our password space as we did. With the characters `a-zA-Z0-9`the password space maxes out at 62 possible characters for each space in the password. This makes the number of possible passwords 62^n where n is the length of the password. To give an idea of how many passwords that contains here is a graph of 62^n.



Observe that there is a large spike around $n = 6$ due to this spike we did not test many passwords over length of 6 as passwords of length 6 already bordered on taking an hour to crack.

We split testing up into 4 different categories:

- Only A-Z, Only 0-9, Only a-z
- Lower Case with numbers
- Upper and Lower case
- Uppercase, Lower, and numbers

We then tested each of categories with passwords of upto $n = 6$ and with 2,4, & 8 cores. The results were as follows.

Note: Some times listed in milliseconds as processes finished faster than a second Or in some cases measuring in milliseconds gave a better indication of time difference

Only A-Z, Only a-z, Only 0-9				
Password	Length	Time(2 Processes)	Time(4 Processes)	Time(8 Processes)
a	1	0s	0s	0s
yz	2	0s	0s	0s
qsc	3	5ms	6ms	5ms
test	4	250ms	43ms	91ms
qwert	5	3625ms	5056ms	3762ms
asdfgh	6	17s	19s	36s
5	1	0s	0s	0s
43	2	0s	0s	0s
733	3	0s	0s	0s
1252	4	2ms	2ms	1ms
99054	5	101ms	80ms	110ms
645367	6	300ms	102ms	199ms
U	1	0s	0s	0s
YS	2	0s	0s	0s
LIX	3	17ms	7ms	7ms
LSOX	4	424ms	204ms	210ms
KIXES	5	9657ms	4383ms	2722ms
SEDVDE	6	126s	3873ms	6720ms

Lower Case & Numbers				
Password	Length	Time(2 Processes)	Time(4 Processes)	Time(8 Processes)
8	1	0s	0s	0s
3c	2	0s	0s	0s
10b	3	25ms	1ms	21ms
8ggw	4	1470ms	679ms	917ms
98ffe	5	61s	33s	38s
bnd3d0	6	170s	226s	300s

Upper and Lower Case				
Password	Length	Time(2 Processes)	Time(4 Processes)	Time(8 Processes)
Z	1	0s	0s	0s
wC	2	2ms	0s	0s
QoD	3	87ms	18ms	3ms
aFEM	4	173ms	290ms	303ms
QQZZo	5	264s	77s	27s
dadEES	6	2360s	3595s	

Upper Case, Lower Case, & Numbers				
Password	Length	Time(2 Processes)	Time(4 Processes)	Time(8 Processes)
Z	1	0s	0s	0s
3c	2	0ms	0s	0s
1Gb	3	175ms	66ms	84ms
F3sZ	4	422ms	1357ms	4115ms
d5pZ1	5	177s	159s	258s

Conclusion

In conclusion, our parallelized password cracker does actually work. The password cracker can crack 5 character long passwords in about a few minutes. With more time to test we imagine it might be able to handle 7-8 character long passwords if on larger machines or given a greater period of time to spend cracking. In most cases during our testing adding more processors did reduce the time needed to crack the password, although not always. The likely reason that it did not always reduce the time needed to crack the password is that we were running this program on our work stations and the presence of other processes may have taken some of the compute time away from the password cracker, though this could be tested further. All in all, we think the project is quite a success though if we were to continue with it there some further improvements and testing that we would like to do.

Appendix-A Code

```
#include <mpi.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <cstring>
#include <vector>
#include "md5.h"
#include <chrono>

#define MCW MPI_COMM_WORLD
#define TERMINATE 8

MPI_Status myStatus;

std::string hash;
int startIndex;
int endIndex;
int passwordLength = 6;
int size, rank, data;
```

```
int myFlag;
auto start = std::chrono::high_resolution_clock::now();

void generate(char* arr, int i, std::string s, int len) {
    // base case
    if (i == 0){ // when len has been reached
        // check if the password was found
        if(hash.compare(md5(s)) == 0){
            auto stop = std::chrono::high_resolution_clock::now();
            auto durationSeconds = std::chrono::duration_cast<std::chrono::seconds>(stop -start
            auto durationMs = std::chrono::duration_cast<std::chrono::milliseconds>(stop-start)
            std::cout << "The password is: " << s << std::endl << std::endl;
            std::cout << "It took " << durationSeconds.count() << " seconds to find" << std::en
            std::cout << "It took " << durationMs.count() << " miliseconds to find" << std::end
            // tell all the other processes to terminate
            for(int i = 0; i < size; i++){
                if(i != rank){
                    MPI_Send(&data, 1, MPI_INT, i, TERMINATE, MCW);
                }
            }
            MPI_Finalize();
            exit(0);

        } else {
            // check if a process found the password
            MPI_Iprobe(MPI_ANY_SOURCE, TERMINATE, MCW, &myFlag, &myStatus);
            if(myFlag){
                MPI_Finalize();
                exit(0);
            }
        }
        return;
    }

    // iterate through the array
    for (int j = 0; j < len; j++) {

        // Create new string with next character
        // Call generate again until string has
        // reached its len
        std::string appended = s + arr[j];
        generate(arr, i - 1, appended, len);
    }

    return;
}

// function to generate all possible passwords
```

```
void crack(char* arr, int len) {
    for(int i = startIndex; i <= endIndex; i++){
        std::string startString = "";
        startString += static_cast<char>(arr[i]);
        generate(arr, passwordLength - 1, startString, len);
    }
}


int main(int argc, char **argv) {

        bool includeLower = false;
        bool includeUpper = false;
        bool includeNumbers = false;

        //MPI INIT Stuff
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MCW, &size);
        MPI_Comm_rank(MCW, &rank);
    int hashLength = -1;
    std::string yes = "Y";

        //Get settings in rank 0 then distribute settings to other ranks
        if(rank == 0){
        //std::cout << "md5 hash: " << md5("test") << std::endl;
                //Settings
                // -Known password length
                // -Provide password hash
                // For now, assume the password has upper case, lower case, and numbers
                bool havePassLen = false;
                while(!havePassLen){
            std::cout<< "\nHow long is the password? " ;
            std::cin >> passwordLength;

            if(passwordLength < 1){
                std::cout << "Please input a password length of at least length 1." << std::end
            } else{
                havePassLen = true;
            }
                }

                bool validPWS = false;
                while(!validPWS){
            std::string lower = "";
            std::cout << "Does the password contain lower case characters? (Y/n) ";
            std::cin >> lower;
            if(yes.compare(lower) == 0){
                includeLower = true;
```

```
        }

        std::string upper = "";
        std::cout << "Does the password contain upper case characters? (Y/n) ";
        std::cin >> upper;
        if(yes.compare(upper) == 0){
            includeUpper = true;
        }

        std::string numbers = "";
        std::cout << "Does the password contain numbers? (Y/n) ";
        std::cin >> numbers;
        if(yes.compare(numbers) == 0){
            includeNumbers = true;
        }

        if(!includeLower && !includeUpper && !includeNumbers){
            std::cout << "Please enter a valid password space." << std::endl;
        } else {
            validPWS = true;
        }
            }

    //flush buffer
    std::string s;
    std::getline(std::cin, s);

            std::cout << "What is the password hash? ";
            std::getline(std::cin, hash);
            hashLength = hash.size();
    }

    // make sure every process knows the password space
MPI_Bcast(&includeLower, 1, MPI_C_BOOL, 0, MCW);
MPI_Bcast(&includeUpper, 1, MPI_C_BOOL, 0, MCW);
MPI_Bcast(&includeNumbers, 1, MPI_C_BOOL, 0, MCW);

// make sure every process has the length of the hash string
MPI_Bcast(&hashLength, 1, MPI_INT, 0, MCW);

// send/recv the hash string as a char array
if(rank == 0){
    for(int i = 0; i < size; i++){
        MPI_Send(hash.c_str(), hash.size(), MPI_CHAR, i, 0, MCW);
    }
}else {
    char hashCharArr[hashLength];
    MPI_Recv(hashCharArr, hashLength, MPI_CHAR, 0, 0, MCW, MPI_STATUS_IGNORE);
```

```
        hash = hashCharArr;
    }

    // broadcast the password length
    MPI_Bcast(&passwordLength, 1, MPI_INT, 0, MCW);

    //Calculate password space
    std::vector<int> passwordSpace;
    if(includeLower){
        for(int i = 97; i < 123; i++){ passwordSpace.push_back(i);} // ascii values a-z
    }
    if(includeUpper){
        for(int i = 65; i < 91; i++){ passwordSpace.push_back(i);} // ascii values A-Z
    }
    if(includeNumbers){
        for(int i = 48; i < 58; i++){ passwordSpace.push_back(i);} // ascii values 0-9
    }

    startIndex = (passwordSpace.size() / size) * rank;
    if(rank == (size - 1)){
        endIndex = passwordSpace.size() - 1;
    } else {
        endIndex = (startIndex + (passwordSpace.size() / size));
    }

    char pws[passwordSpace.size()];
    for(int i = 0; i < passwordSpace.size(); i++){
        pws[i] = static_cast<char>(passwordSpace[i]);
    }

    // find the password
    ::start = std::chrono::high_resolution_clock::now();
    crack(pws, passwordSpace.size());

    MPI_Finalize();
        return 0;
}
```