Assignment 1 – MPI

Parallel CrackMe challenge

Abstract

In this paper is provided a solution to the assignment CrackMe, using recursion as a search function and dividing work up by calculation to be performed in parallel using MPI.

Introduction

The CrackMe challenge is a simple task of covering all possibilities of a C string of varying length, each length having a different and predetermined generated solution. The real task is to create a process that can be run in parallel to solve the CrackMe challenge. The string for the challenge must be auto-generated and compared to the solution given by a function p that provides a yes or no answer to the comparison. Additionally, the solution must be tested for efficiency using time metrics.

The C string is generated using a character pointer from the C language. A C char has a potential value from -128 to 127, giving it a range of 0 – 255 possibilities which shows a character based on the ASCII standard. It is represented as 1 byte. The C string is a pointer to a memory location carrying the first character and works as an array. Usually, a string includes a null-terminator, but for this task it is not necessary.

Parallel design solution

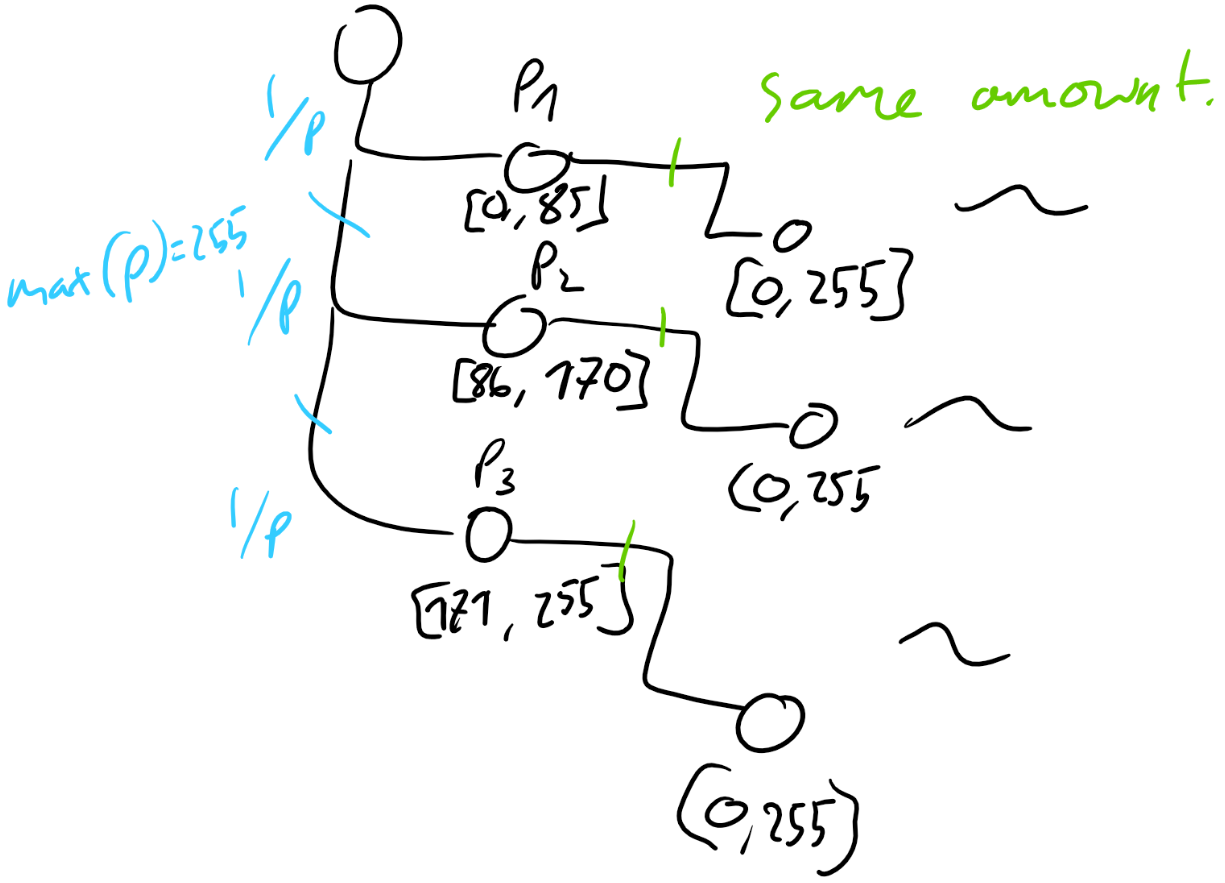
CrackMe solution

To solve the CrackMe challenge, the solution is designed as a brute-force attempt at guessing every possibility using recursion. First, the program generates a string to a predetermined length that the program is looking for. The string is then checked from end to start using every combination of the char byte. The range possible is 255, so for each added length of the string, the increase in complexity is an order of magnitude greater.

A problem of this design is that the start of the check always go from -128 to 127, meaning that if the solution statistically lies closer to the start of the range for all letters, the solution will complete faster than otherwise, and slower should the solution lie closer to end of the range. One way to potentially average this out could be to randomly or alternatively start from the other end.

Sequentially, the program checks the last letter first, and for every exit of a recursion, the previous letter gets one iteration through before checking all the next letters until every possibility is tested.

In parallel, the program divides up a workload of the first letter equally between all processes. This means that the program can scale up to 255, where each process has to iterate between each subsequent letter length for only 1 early character, dividing up the largest order of magnitude between each process. The downside is that there is a cap of 255, meaning that any process above 255 will create problems. Once a process has found the solution, it tells all the other processes to abort, and gives out the solution.



Figur 1 The logic behind the workload division.

As shown in Figure 1, given that a process has a range of the first letter to check, the subsequent amount of calculations are equal to all processes. Given 255 processes, each process calculates 255 to the power of N-1 calculations, unless a solution is found faster.

A further solution could be to split up recursion between ranges, allowing further processes above 255 to split workloads through the next order of magnitude, to increase the process limit an order of magnitude.

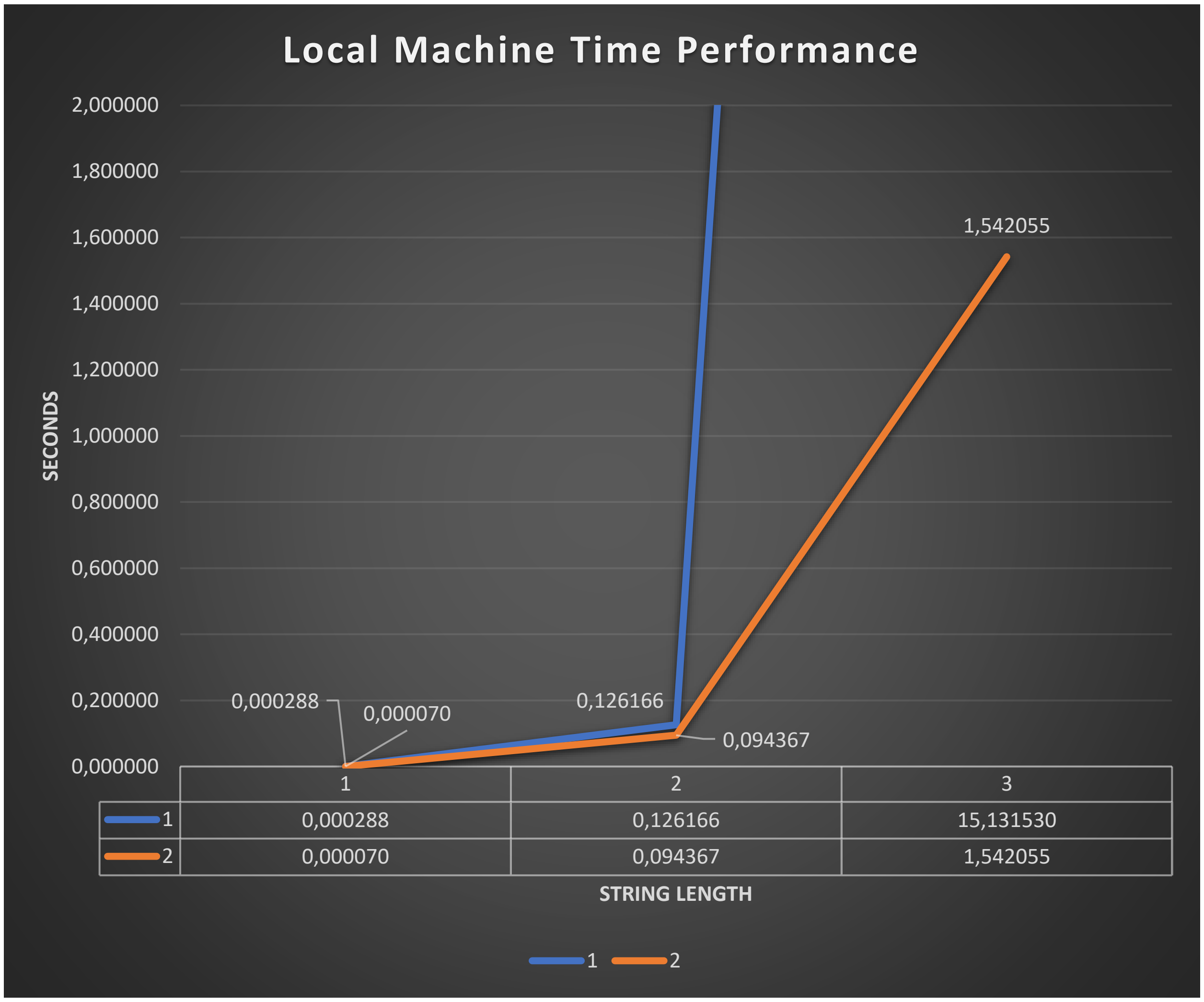
Time performance analysis

Time performance is measured by using clock_t from time library for sequential approach, and for multi-process, MPI has its own measurement function. Both are ran directly before and after running the search function, meaning they give the total completion time for only the search function for the successful finding node. The measurements are then collected 3 times for each length and stored in a spreadsheet, before being put into a graph.

Local machine

On the local machine used to create and test the function is a HP EliteBook 840 running on Linux Mint 20.3 with Linux Kernel version 5.4.0-126-generic. Using MPI, the machine is able to run on 2 processors total, meaning the testing was done between 1 and 2 processes and a string length up to 3, which means a total of 16 581 375 possibilities. A test was done on a string length of 4, but it took around 2050 seconds or about 34 minutes to complete.

The results are as follows:



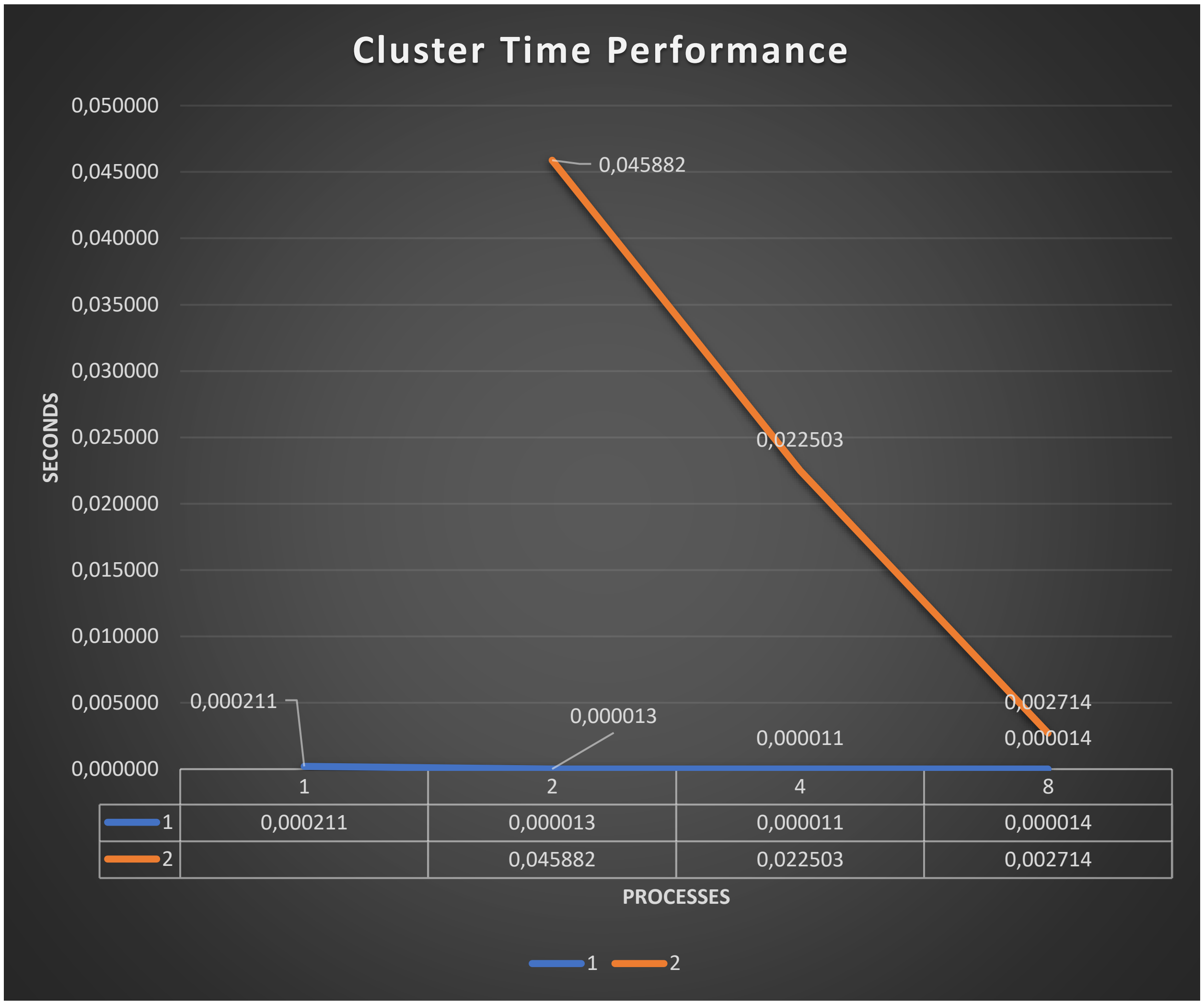
	1	2	3
1	0,000288	0,126166	15,131530
2	0,000070	0,094367	1,542055

Figur 2 Local Machine. X axis: String Length, Y axis: Seconds. Lines are Processes and time.

As we can observe, adding an additional process to handle the workload significantly reduces the amount of time on the local machine. With a length of three, the calculation is reduced by 10 whole seconds. Even by a string length of one, it takes one third the amount of time to calculate.

With the local testing we can observe that the solution for generating and comparing a string can scale both with number of processes and string length. The increase in complexity seems to scale logarithmically, as we would expect given the mathematics of 255 to the power of N.

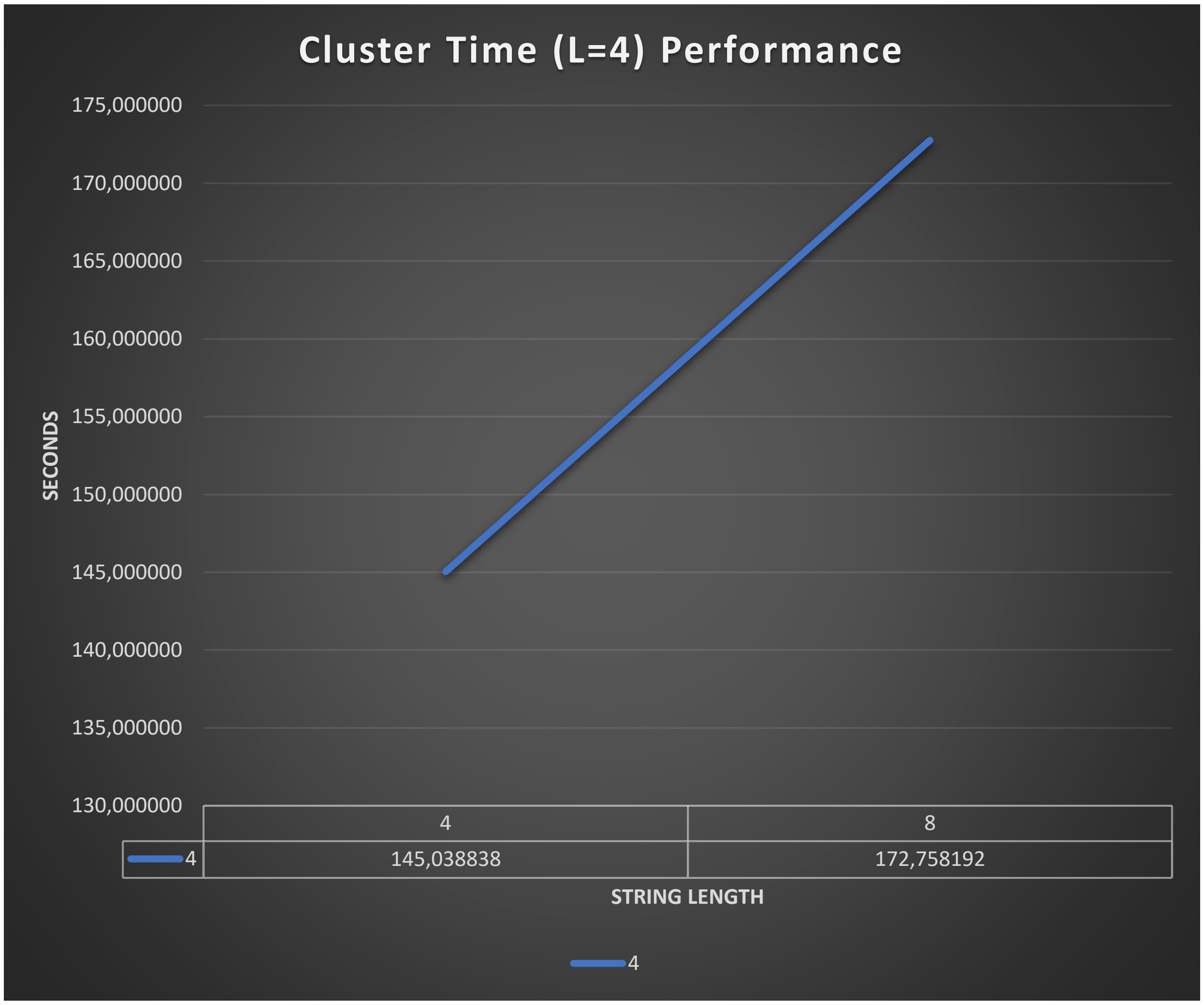
Cluster



	1	2	4	8
1	0,000211	0,000013	0,000011	0,000014
2		0,045882	0,022503	0,002714

Figur 3 String lengths of 1 and 2, processes from 1-8 on the X axis.

For the cluster, string length 1 and 2 using nodes up to 8 shows expected results, especially for string of length 2, cutting down search time drastically, nearly comparable to searching for string length of 1 with 1 node. Additionally, we can observe that the solution for string length 1 is so close that adding more nodes does not change computation time.



	4	8
4	145,038838	172,758192

Furthermore, using a string length of 4 shows that more processes might not even be a better deal. At 8 nodes, the time taken increases with the current solution. However, on the local machine it still took about 2050 seconds to find a string length of 4, so reducing it down to an average of 172 is still better.

Results beyond string length of 4 on the cluster proved difficult, as the cluster and VPN would time out and reconnect before the cluster could finish processing the length.

Discussion

As mentioned in previous sections, the positives of this solution is that every possibility is checked. It is a simple and naïve brute-force solution that does exactly as described only. And due to the time demand for a simple check of a string, the true downside of brute-force solutions, making them inelegant when attempting to crack something. It also highlights how, especially for security, complexity is not as good as length.

Potential improvements:

- Implement threading to allow for message sending and receiving instead of using MPI_Abort upon finding solution (graceful exit).
- Change implementation of search to go in stages: attempt the most popular combinations of characters either using statistical analysis or some other form of algorithm.
- Implement threading of search to potentially utilize hardware, if available, better.

From the time performance analysis on the local machine, the observations are that attempting to "guess" by brute force a random string is an extremely intensive process. By looking at how the computer handles a length of 1, which resulted in the special character of '\006' which is the 6th number, we can estimate that each lookup took around 0,000048 seconds, or 48µs. Considering a CPU may have cycles down to nanoseconds, this is slow.

Summary

This implementation of CrackMe has used recursive search to generate a string that is compared against a predetermined solution using a predefined function. The search can be done sequentially, or in parallel using MPI. As a character in C is a value between -128 and 127, giving 255 variables to attempt, generating the string requires at worst 255 ^ N computations, which increases drastically by each order of magnitude. To divide up this work, the program calculates its own workload given the amount of processes that are available. The assumption is that by dividing up work only on the maximum order of magnitude, each subsequent lower order will be equally divided, up to a maximum of 255 processes.

Given that a string of length 2 will have 65025 comparisons, 255 processes will each need to process 255 variables, or one char each. From the time analysis it is concluded that each computation takes about 48µs. By increasing complexity, the shift of lookup time is dependent on search start location, but to a maximum of 12,24ms. By instead increasing the string by 1 letter, the computation time would increase by this logic to 3,12 seconds.