1. Introduction

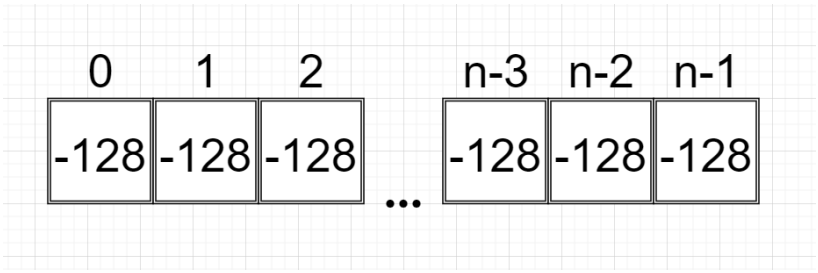
The purpose of the assignment is to make sequential and parallelized versions of a dynamic char array generator which would be used for picking up a combination of chars same as the one generated by provided in crackme.o file function p(). The solution implements brute force algorithm which is commonly used for similar tasks. The idea of brute force is to check every possible combination in the set of element to find the needed one. Most of the password hacking attacks are performed with using this technique [1].

After the implementation of the generator, the set of measurements should be done to evaluate the time performance of the solution. It is expected that on multiple processes the program would give much better performance than on a single thread.

The solution was implemented with C and MPI (MPICH implementation) without any additional non-system libraries.

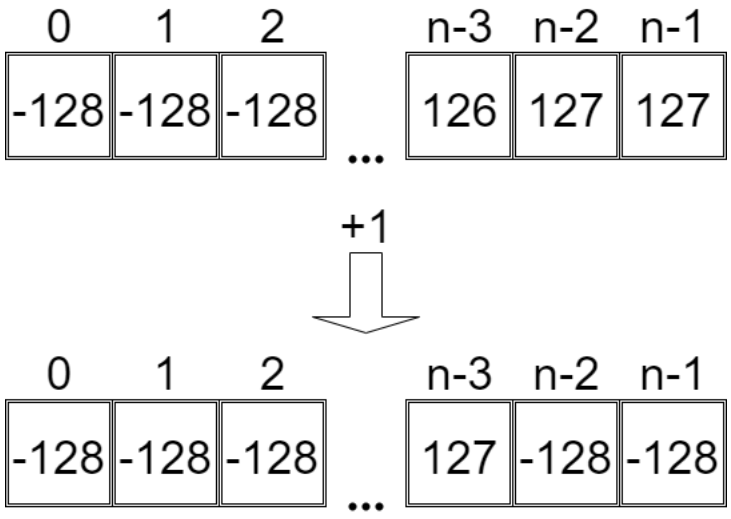
2. Sequential solution

As there is no specific information about the contents of the char combination we are looking for, lets assume that it would be chars with any possible numerical value from -128 to 127. So, due to the theory of combinatorics the amount of all possible permutations is 255^n where n is the size of the string [2]. I've made a quite simple brute force solution which iteratively goes through every possible combination until it finds the needed one.



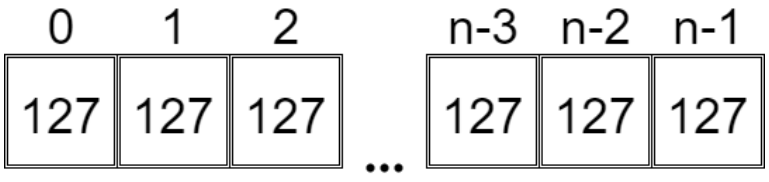
Picture 1: The initial contents of char*.

At first, the char* of the passed to the program length generates and getting filled with chars of minimum value of -128. After that starts the while loop where every combination of chars in the char* is being iteratively checked with p() function until every char in char* would become equal to 127, that means that every possible combination was checked. On every iteration of the loop numerical value of the last char in char* increases by 1, if it becomes more than 127 (maximum value), this char is being assigned the value of -128. After that the value of the char on the position before the last char increases by 1 and if it becomes equal to 128, it's being assigned the value of -128 and the value of the char on the position before increases by 1 and this loop goes until the char's value becomes less than 128 or until the first char of char* becomes equal 128. An example of this algorithm during a loop iteration is presented on Picture 2.



Picture 2: An example of computing a new combination.

When the first char in char* is being assigned the value of 128, program goes out of the loop. This means that all of the combinations have been compared by the p() function and no correct combination was found. Although it is not possible not to find the needed combination with such a brute force, this check had to be done for protection from mistakes in other parts of the code which could lead to infinite loop, for example, bad passing char* to p() function. The last possible combination which breaks the loop is shown on picture 3.



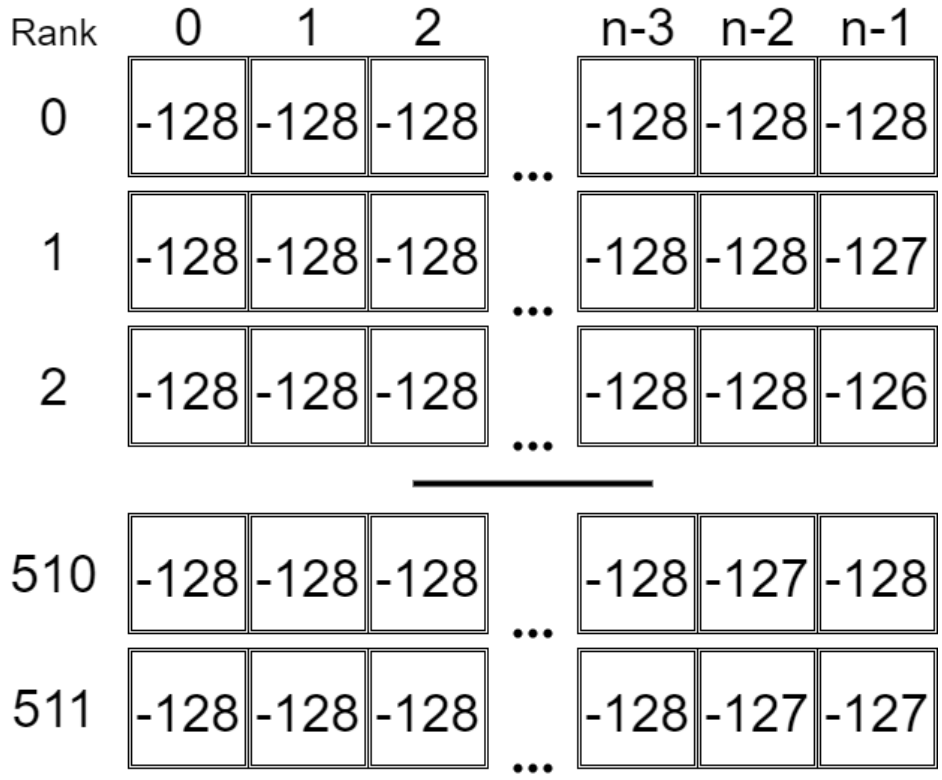
Picture 3: The last possible combination before terminating.

3. Parallel design solution

The sequential solution was parallelized by dividing the computations among all processes.

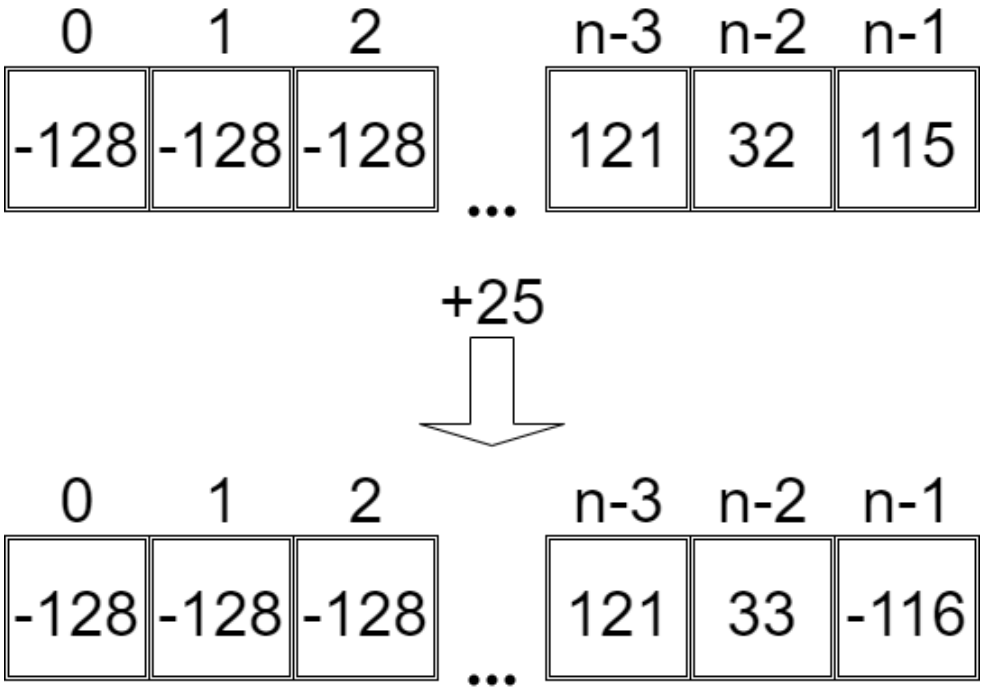
To be sure that processes will not compute same combinations, starting combination is being generated for every rank before the main loop. The last char in char* increases by the number of rank. If rank number is bigger than 255, the last numerical char value increases by the remainder of rank size divided by 255 and the result of division by / operator is saved for computation of value for chars on previous positions, after that same operations are being made for calculating the chars until the division result becomes 0 or if it equals more than 0 when loop hit the first char of char* (it means that number of processes is more than number of possible combinations).

An example of forming initial combination for ranks with a number of processes = 512 is shown on Picture 4.



Picture 4: A schema of forming initial combinations for ranks.

Every rank performs computations in a similar to sequential solution loop, but the difference is that the last char of the char* increases by the number of processes in every iteration. The char increasement is very similar to the loop described in a previous paragraph but instead of using rank number, the number of processes is being used. An example of computing a combination for next iteration is shown on a Picture 5.



Picture 5: Computing a new combination for an iteration of rank 25.

After one of the processes finds the correct combination, it sends a message with tag = 1 to all other processes and exits the loop. In the end of the loop iteration every process checks if it has any incoming messages with tag = 1 by using MPI_Iprobe and if it has any messages to receive, it exits the loop and finishes its work.

4. Time performance analysis

After the implementation of parallel solution, a set of measurements was made on UIT cluster. At the beginning of the brute force algorithm the size_t variable is stored and the difference between this variable and time in the end is computed with difftime () function [3].

For analyzing time performance there were several executions of the parallel solution with char* size input from 1 to 6 and number of processes from 2 to 2^9 (the number of processes was only a power of two 2^n) and the sequential solution with the same size inputs. Program was executed on multiple nodes of the uvcluster. The table with the results is below.

Processes count (s) / char* size	1	2	4	8	16	32	64	128	256	512
1	<1	1	1	1	1	1	1	1	1	2
2	1	1	1	1	1	1	1	1	1	2
3	14	8	6	3	2	1	1	1	1	2
4	2443	1357	769	441	247	131	74	43	24	14
5	>86400	>86400	>86400	>86400	70159	35489	17853	9048	4682	2555
6	>86400	>86400	>86400	>86400	>86400	>86400	>86400	>86400	>86400	>86400

Table 1: Set of solution time performance measurements.

The measurements have shown that the maximum size of char* combination that can be picked by the solution using 512 processes within 24 hours is 5.

The parallelized solution starts showing significant advantages over sequential version since the char* size gets bigger than 3. In this case program works almost 175 times faster on 512 processes than on a single one.

The amount of program working time gets smaller by 40% - 50% with every time the number of processes doubles. The measurements result is presented on the Figure 1.

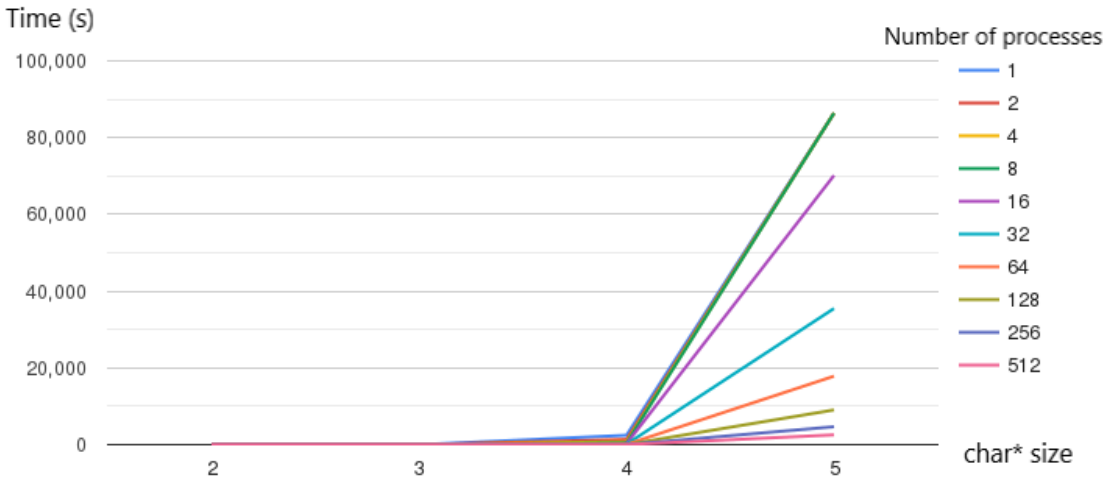


Figure 1: Relation of time to char* size for all numbers of processes measured.

Measurements of performance for char* size < 3 have shown that a big number of processes for a small char* size might cause a longer execution. This might have due to a higher complicity of computations for getting a new combination than the sequential algorithm or because of a higher load on a cluster by other processes during measurements. The results of measurements for char* size = 3 are shown on Figure 2.

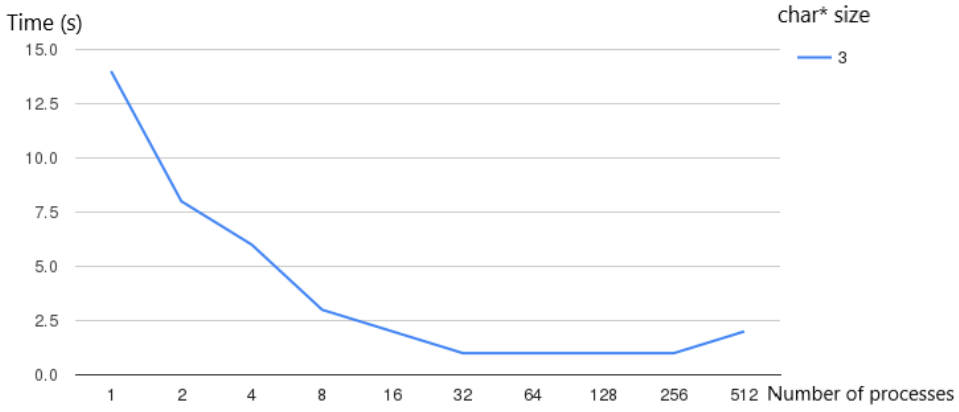


Figure 2: Relation of time to number of processes on char* size = 3.

5. Discussion

The positive side of this solution is the simplicity of implementation and parallelization of the algorithm.

The negative side is the low speed even with the big number of processes. The algorithm might be modified to ease computation. Also, the search of needed combination can be improved for higher performance. One of the methods to achieve that is to make half of the processes search the combination by starting at the beginning of the char* and moving to its end.

Also it is worth trying to use CUDA (Compute Unified Device Architecture) tool for the implementation of computation with Graphic Processor Unit. Brute force with CUDA shows 5 times better performance than brute force with MPI [4].

6. Summary

All the assignment requirements were successfully met. The sequential and parallel versions of char* combination searcher were implemented and described in this report.

The measurements of the solution time performance were made and analyzed, analysis of measurements was presented in this report.

7. Reference List

1. Leon Bosnjak. “Brute-force and dictionary attack on hashed real-world passwords”. https://www.researchgate.net/publication/326700354_Brute-force_and_dictionary_attack_on_hashed_real-world_passwords .
2. S. Chand Experts. “Mathematics: 15 Years Solved Papers And Practice Questions For Jee Main & Advanced (2004-2018)”, page 8-1.
3. difftime - cppreference.com. https://en.cppreference.com/w/c/chrono/difftime .
4. Francisco G. Montoya. Cryptanalysis of Hash Functions Using Advanced Multiprocessing. https://www.researchgate.net/publication/225910468_Cryptanalysis_of_Hash_Functions_Using_Advanced_Multiprocessing .