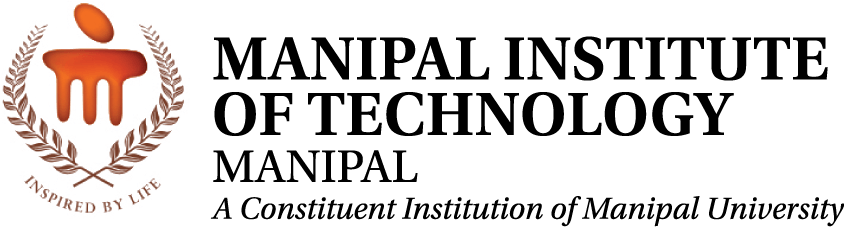
**Password Recovery**

**(Using MPI and CUDA)**

**Team Members:** **Abhilash ( Roll no. 1, D-1)**

**Shivam Varshney ( Roll no. 17, D-1)**

**Nihal Pereira ( Roll no. 19, D-1)**



**Research Domain**

Network Security

**Abstract:**

One of the raising security concerns in the modern day cyber security is electronic password potential. The human set passwords are bound to be lost or misplaced or forgotten for good, resulting in the loss of important data. Also Brute forcing of passwords is a computationally challenging field varying in bounds by the passing hour. With the modern day computer architecture reaching the levels of heavy neural compute engines, graphic processing units and other distributed architecture, it gives a totally different perspective at brute forcing passwords for security purposes. In this project, we plan to apply the hashed functions over a heavy list of known words and run them parallel in an attempt to have a hash match and thus the deciphering of the hashed password. The implementation will be with help of both MPI and CUDA. Due to the high level of fine-grained parallelism of this type of problem, GPU computing using Compute Unified Device Architecture (CUDA) can be used to further improve performance. In addition to recovering lost passwords, this work could be used to help improve the security of computer systems by identifying accounts with weak or common passwords. The framework described may also be useful for other research that needs to process large amounts of data with similar characteristics using MPI and CUDA.

**Motivation:** Nowadays password lengths are getting bigger. Using Dictionary Attack technique using serial codes can take hours and sometimes upto day to recover a password. In this project, we find a solution t this by using parallel codes and GPU. Such an approach will result in future performance improvements of password recovery softwares. Although the project input dictionary takes in a limited no of passwords, we have computed the permutations of these passwords to generate a larger set of passwords.

**Objectives :**

* Build a fast and efficient GPU Dictionary Attack Technique based on available input passwords.
* Efficiently compare all the strings in a parallel manner to avoid iterative comparisons.
* Achieve a better performance and scalability as compared to the sequential algorithm.

**Introduction :**

Electronic authentication is the process of confirming a user identity presented to an information system. A widely deployed method for confirming a user identity is based on passwords. A person submits her user id and password to a computer system. The computer system processes the password with a hash function and compares the hash function output with a hash value associated with that user id from a password database. If the two hash values match, she is granted access to the system. Millions of people follow this protocol to confirm their identity multiple times each day.

The hash functions used to authenticate users are called cryptographic hash functions. These functions read a string of characters of any length, perform a hash operation on the string, and then return an encoded, hashed value as a fixed-length string. A good cryptographic hash function is one-way, which means the original input string cannot be recovered by reversing the function. This one-way characteristic has led to passwords being stored as hashed values instead of in their plain-text format. Accordingly, even if the password database is compromised, the plain-text passwords are still somewhat secure.

One problem with password authentication is that some passwords are weak; they contain little randomness from one character to the next. When people create their own passwords, it is natural for them to choose words that are easy to remember. Passwords are sometimes based on familiar names, common words, or are simple sequences like “12345.” Those types of passwords are easily guessed by attackers.

A dictionary-based password recovery application performs a large number of hash and string comparison operations. A password dictionary is filled with familiar names, common words, and simple character sequences. With a large dictionary or a large number of entries in a password database, password recovery is computationally expensive. However, this type of analysis has a high level of parallelism; testing one word from the dictionary is independent of testing another. This makes dictionary-based password recovery well suited for a HPC environment, like MPI or CUDA. The parallelism can be taken even further and implemented in a hybrid MPI+CUDA environment, which allows for an initial course-grained division of the problem using MPI and a fine-grained division of the problem using CUDA.

**Methodology:**

The input file is read into a buffer. We are then calling the kernel which consists of n no\_of\_threads equal to the no of words.

The hash function is computed for each string in parallel and then comparing it with the hash value of the password.

As soon as we get a match we are displaying it .If we do not get a match,

We are then computing the permutations of each and every value in the dictionary and comparing its hash with the hash value of the password.

**Algorithms:** 1) Divided Dictionary Algorithm-

2) Hashing Algorithm

3) Permutation Algorithm

**MD5 :** MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 264.

SHA : *Note 1: All variables are unsigned 32-bit quantities and wrap modulo 232 when calculating, except for*

*ml, the message length, which is a 64-bit quantity, and*

*hh, the message digest, which is a 160-bit quantity.*

*Note 2: All constants in this pseudo code are in* [*big endian*](https://en.wikipedia.org/wiki/Endianness)*.*

*Within each word, the most significant byte is stored in the leftmost byte position*

*Initialize variables:*

h0 = 0x67452301

h1 = 0xEFCDAB89

h2 = 0x98BADCFE

h3 = 0x10325476

h4 = 0xC3D2E1F0

ml = message length in bits (always a multiple of the number of bits in a character).

*Pre-processing:*

append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.

append 0 ≤ k < 512 bits '0', such that the resulting message length in *bits*

is [congruent](https://en.wikipedia.org/wiki/Modular_arithmetic) to −64 ≡ 448 (mod 512)

append ml, the original message length, as a 64-bit [big-endian](https://en.wikipedia.org/wiki/Endianness) integer.

Thus, the total length is a multiple of 512 bits.

*Process the message in successive 512-bit chunks:*

break message into 512-bit chunks

**for** each chunk

break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

*Extend the sixteen 32-bit words into eighty 32-bit words:*

**for** i **from** 16 to 79

w[i] = (w[i-3] **xor** w[i-8] **xor** w[i-14] **xor** w[i-16]) [**leftrotate**](https://en.wikipedia.org/wiki/Circular_shift) 1

*Initialize hash value for this chunk:*

a = h0

b = h1

c = h2

d = h3

e = h4

*Main loop:*[[3]](https://en.wikipedia.org/wiki/SHA-1#cite_note-:0-3)[[54]](https://en.wikipedia.org/wiki/SHA-1#cite_note-54)

**for** i **from** 0 **to** 79

**if** 0 ≤ i ≤ 19 **then**

f = (b **and** c) **or** ((**not** b) **and** d)

k = 0x5A827999

**else if** 20 ≤ i ≤ 39

f = b **xor** c **xor** d

k = 0x6ED9EBA1

**else if** 40 ≤ i ≤ 59

f = (b **and** c) **or** (b **and** d) **or** (c **and** d)

k = 0x8F1BBCDC

**else if** 60 ≤ i ≤ 79

f = b **xor** c **xor** d

k = 0xCA62C1D6

temp = (a **leftrotate** 5) + f + e + k + w[i]

e = d

d = c

c = b **leftrotate** 30

b = a

a = temp

*Add this chunk's hash to result so far:*

h0 = h0 + a

h1 = h1 + b

h2 = h2 + c

h3 = h3 + d

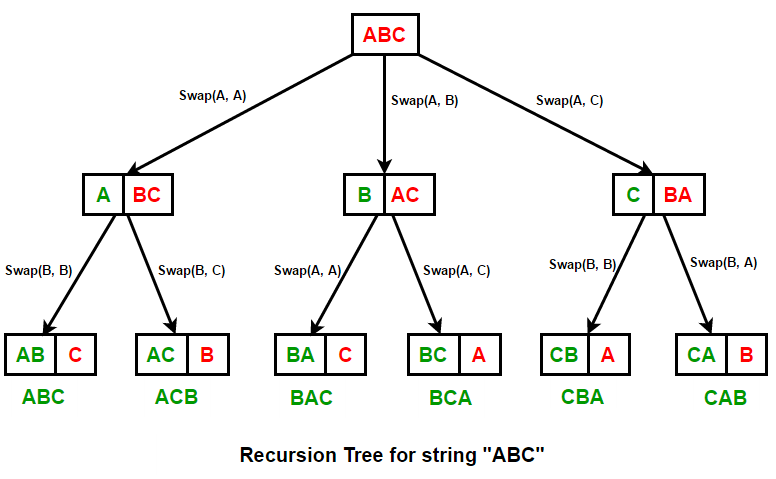
h4 = h4 + e

*Produce the final hash value (big-endian) as a 160-bit number:*

hh = (h0 **leftshift** 128) **or** (h1 **leftshift** 96) **or** (h2 **leftshift** 64) **or** (h3 **leftshift** 32) **or** h4

Sha1,sha256,sha512 are computed along with the requirement of the length of hash length and then brute forced to recover the password.

**Permutation Algorithm:** All the permutations of a string can be found by backtracking. The method used it to swap each of the remaining characters in the string with its first character and then find all the permutations of the remaining characters using a recursive call. The base case of the recursion is when the string is left with only one unprocessed element. Eg: Given below is the recursion tree for the string “ABC”.



#include <iostream>

using namespace std;

// Function to find all Permutations of a given string str[i..n-1]

// containing all distinct characters

void permutations(string str, int i, int n)

{

// base condition

if (i == n - 1)

{

cout << str << endl;

return;

}

// process each character of the remaining string

for (int j = i; j < n; j++)

{

// swap character at index i with current character

swap(str[i], str[j]); // STL swap() used

// recurse for string [i+1, n-1]

permutations(str, i + 1, n);

// backtrack (restore the string to its original state)

swap(str[i], str[j]);

}

}

// Find all Permutations of a string

int main()

{

string str = "ABC";

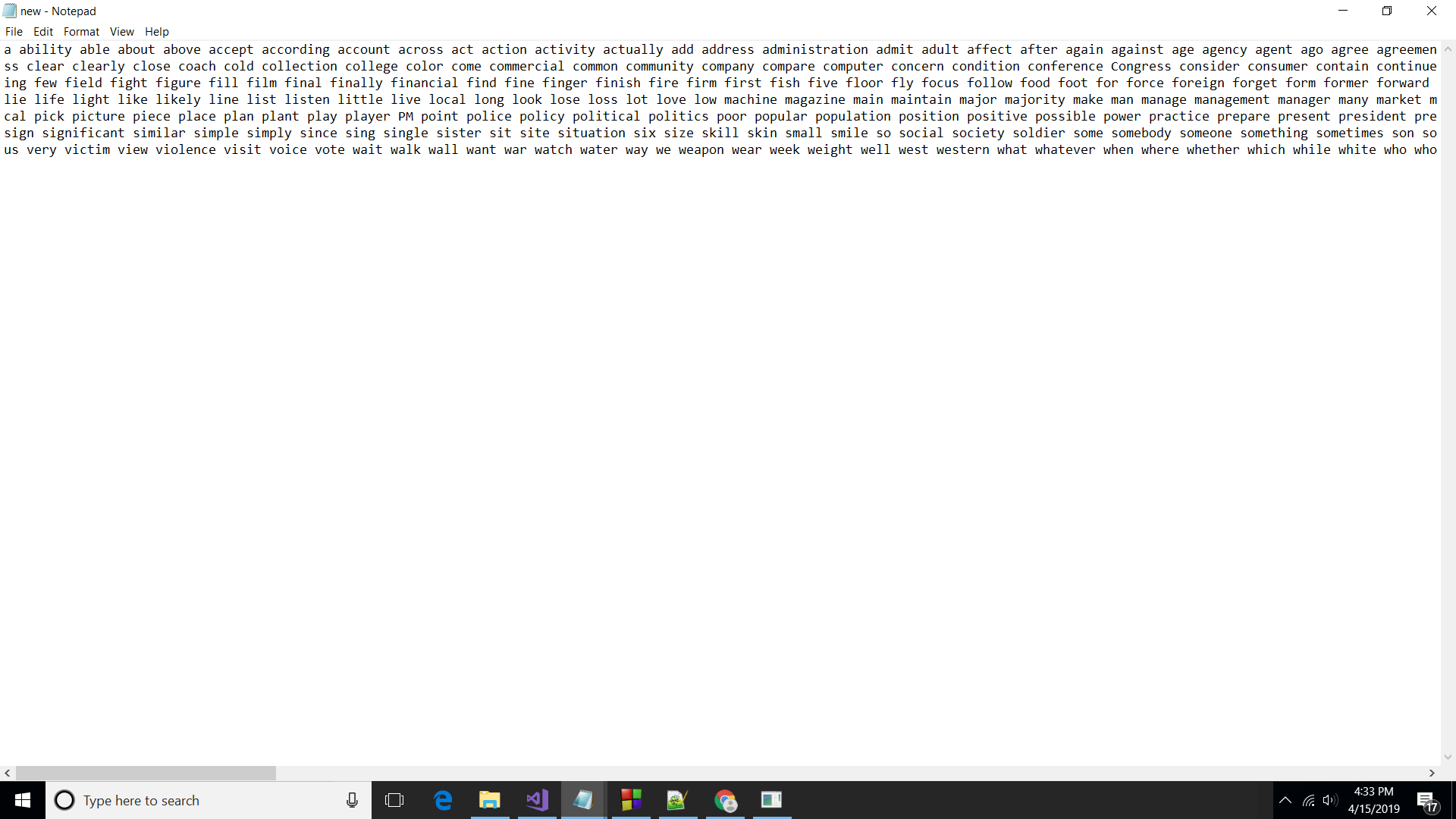
permutations(str, 0, str.length());

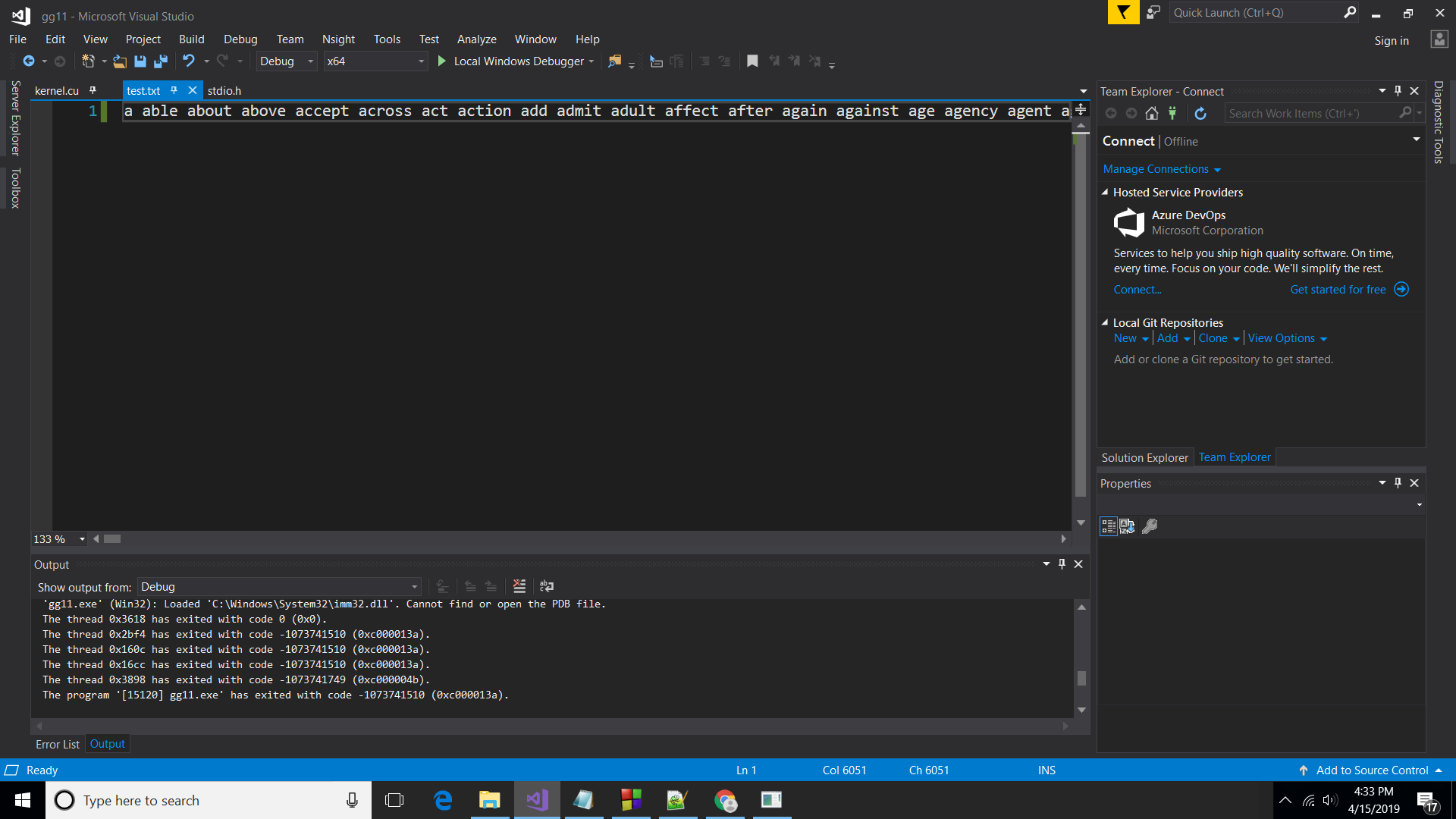
return 0;

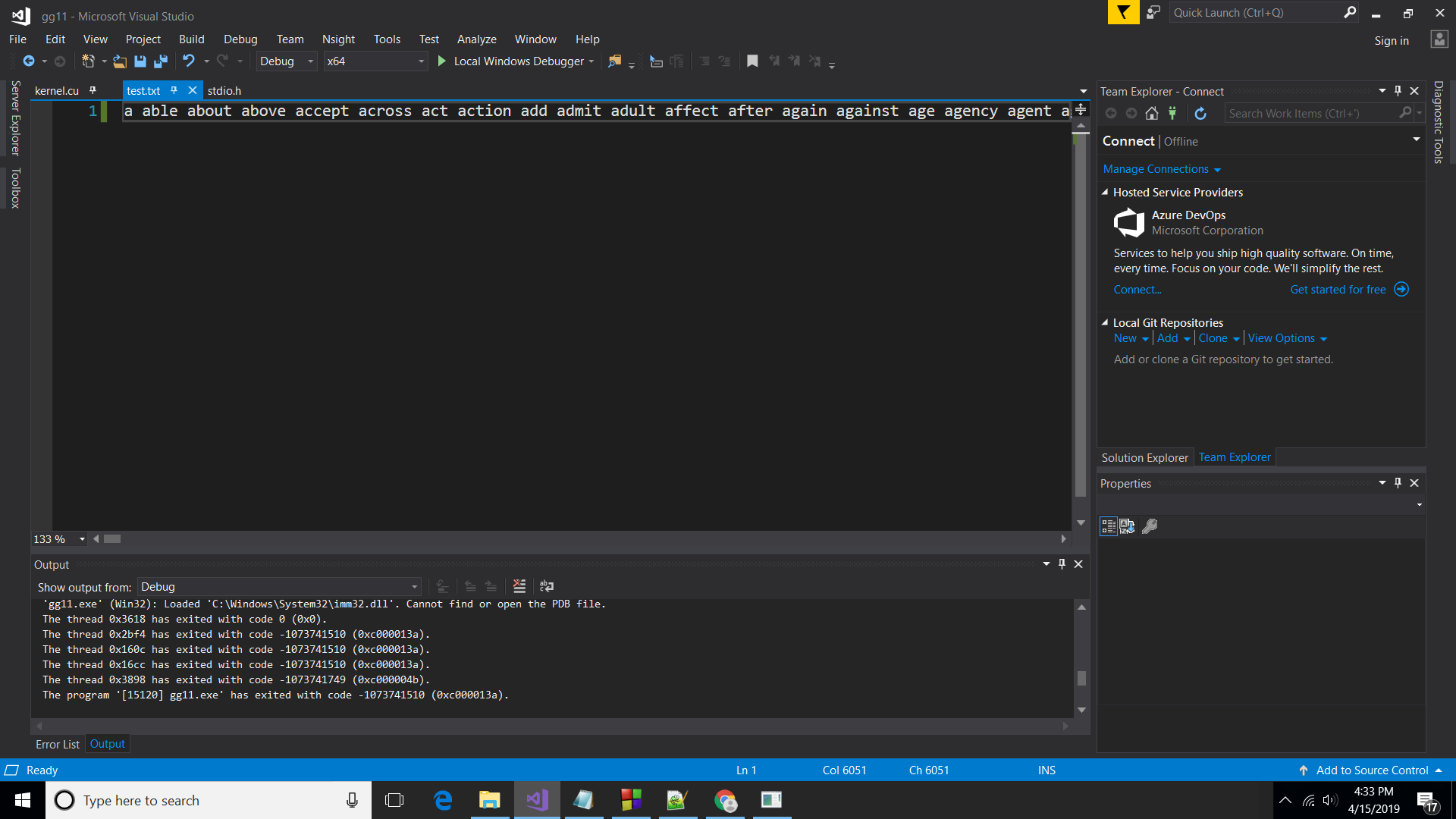
}

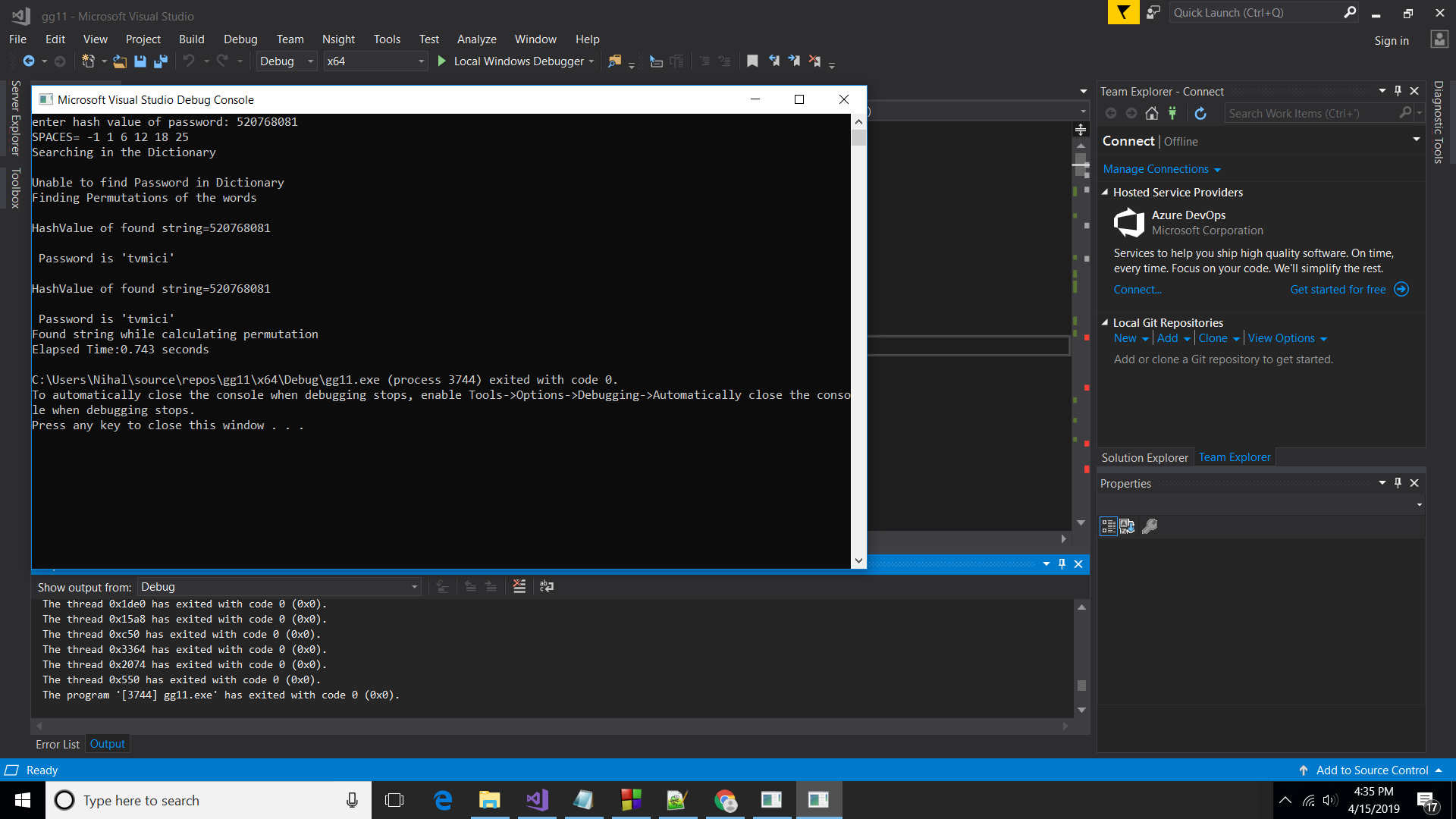
**Input/Output:**

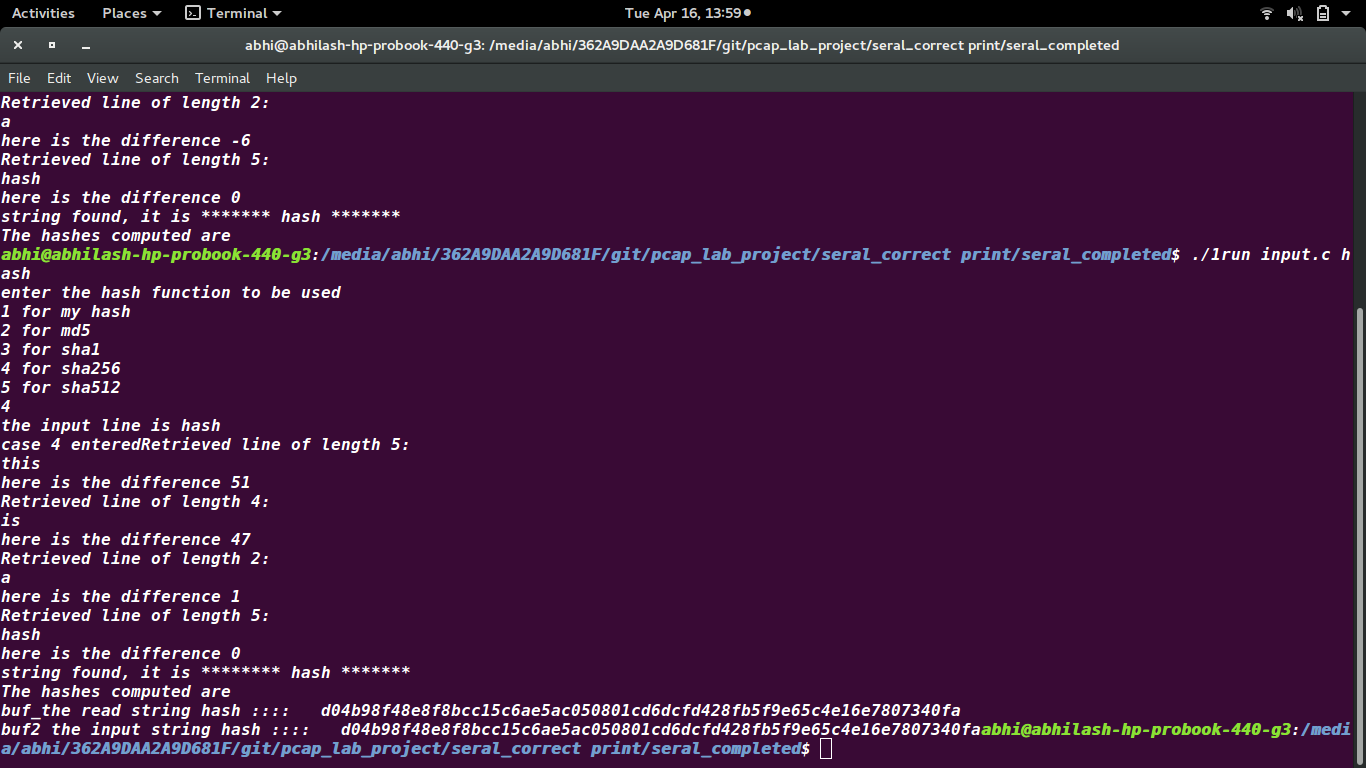
CUDA Code :

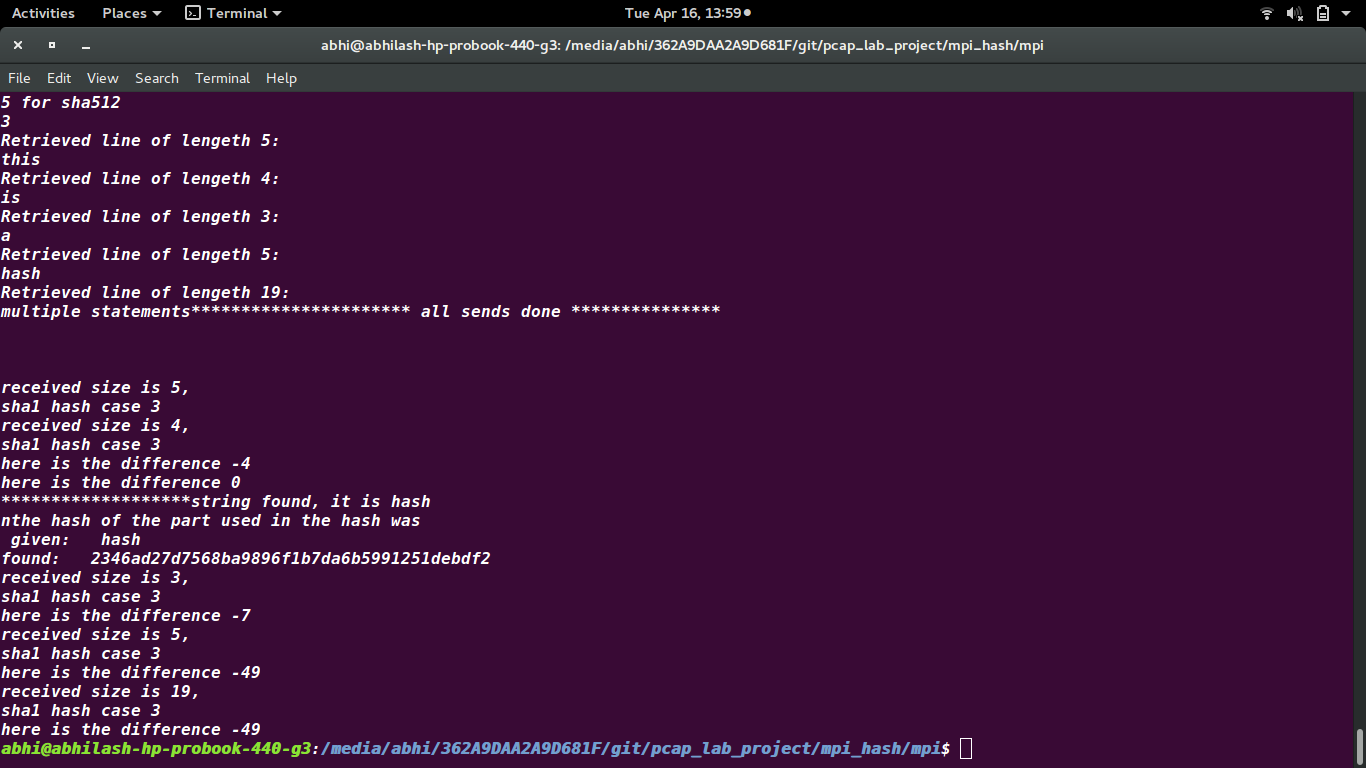




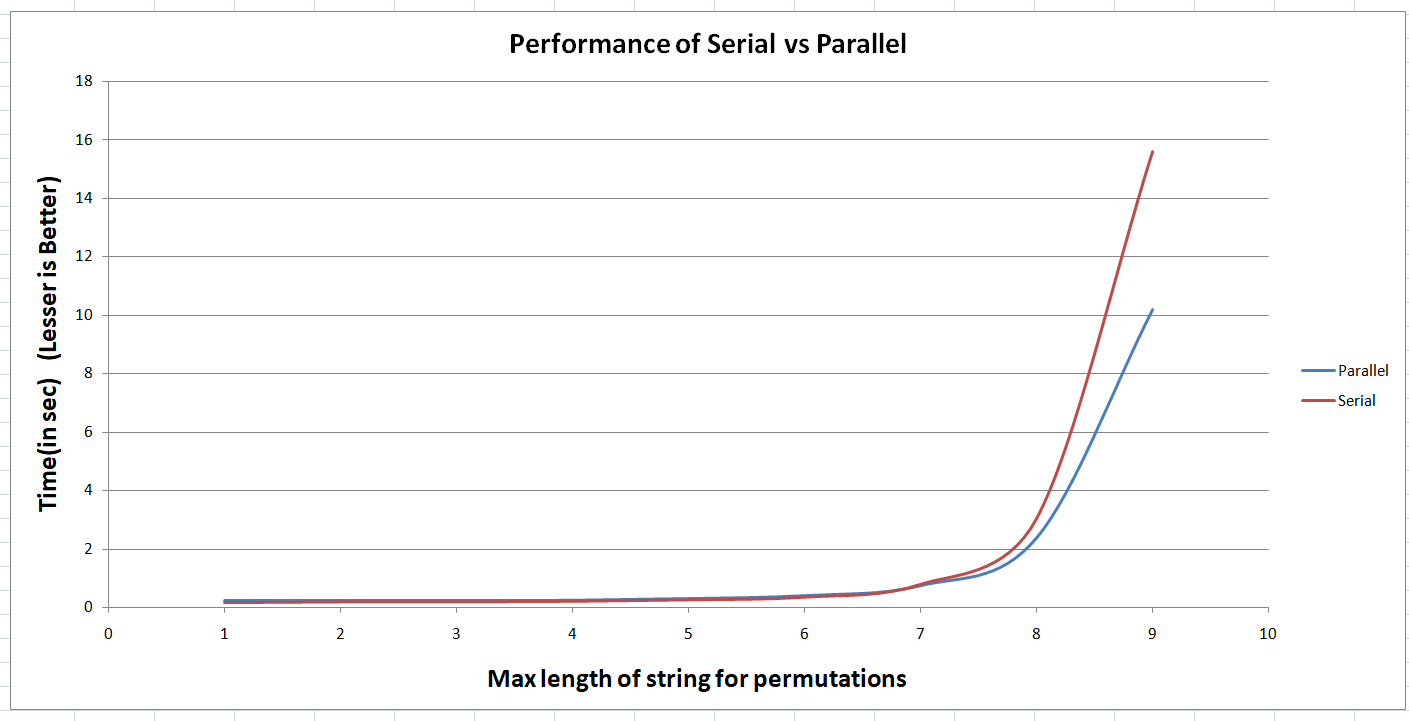


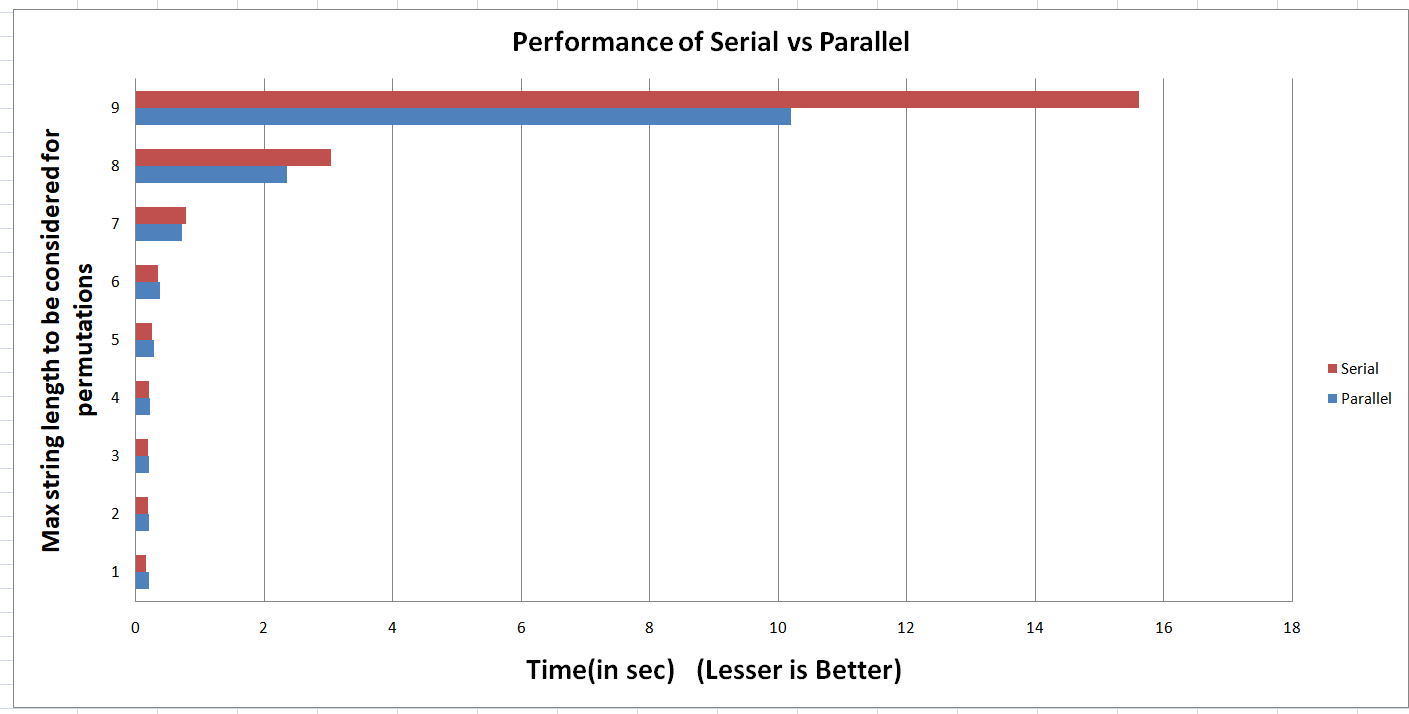


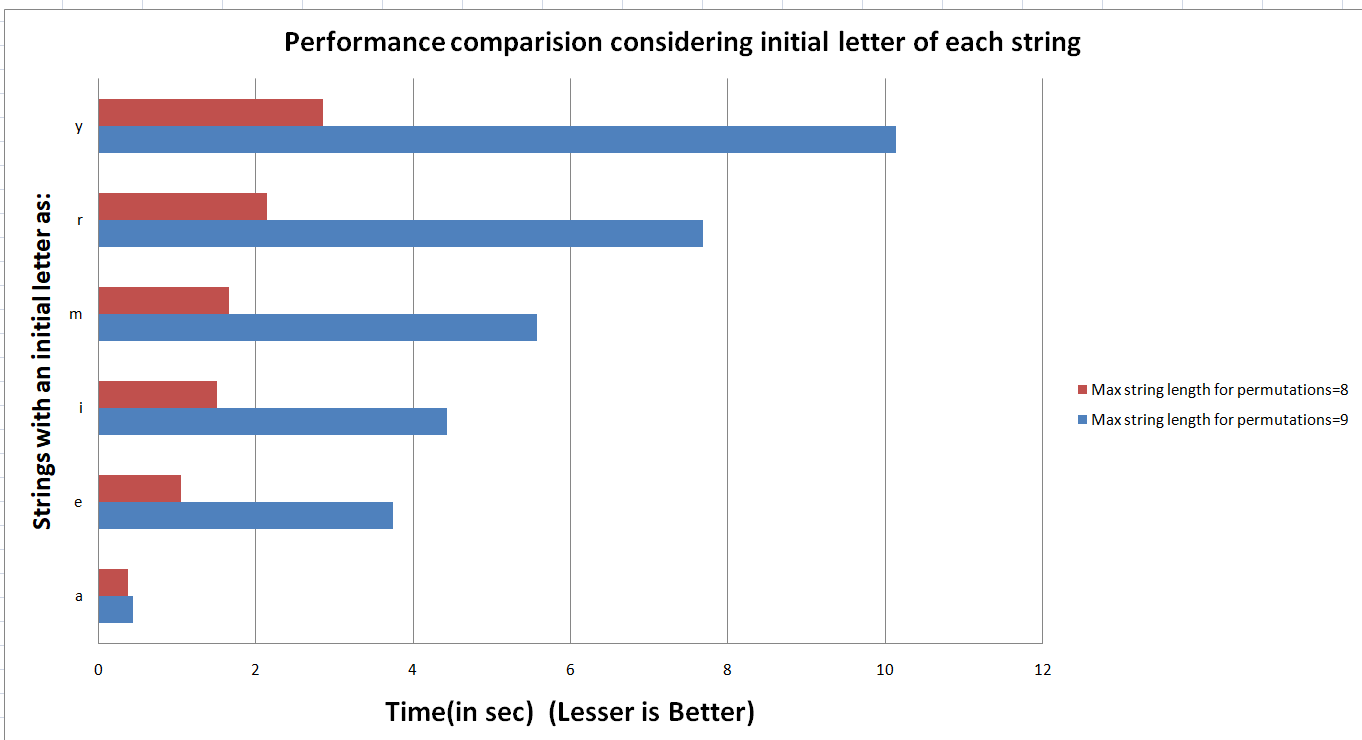
MPI Code: 



**Results :**

****

****

****

**Possible Improvements :**

* To increase the amount of inputs present in the Dictionary**.**
* To increase the amount of parallelism for the permutations.

**Limitations :**

* For small inputs the execution time between serial algorithm and parallel algorithm is small.
* Number of inputs to the input dictionary is limited.

**Reference:** https://ieeexplore.ieee.org/abstract/document/6507505

WWW.Wikipedia.com