Research logbook for Parallel Computing CW1

By Cliff Kirkman

**Introduction**

In this logbook I will write down all stages of research and development that I work on during CW1 for parallel computing, during my work in industry year I have written many problems which work in parallel, this means I already have a firm understanding of the concept. For my research stage will mainly be focused on how AES encryption/decryption works and brute force techniques.

Below I will document each stage of creating a program which will crack a 128 bit key, using a brute force technique in three different ways serial, parallel threaded and parallel clusters. This is an ethical hacking operation, which starts with all data already known therefore the search algorithm can be completed by comparing original text to text which has attempted to be cracked.

**31/10/2018 – AES Research**

AED (Advanced Encryption Standards), is a worldwide encryption and decryption method used by a large amount of different software technology from Facebook messenger to Bitlocker. It replaced DES (Data Encryption Standard) after it was cracked in under 24 hours in 1999.

AES is a symmetric key algorithm which means the same key can be used to encrypt and decrypt the text data, there are three block ciphers AES-128, AES-192 and AES-256. Each cipher requires a key to encrypt or decrypt the data, key sizes are 128, 192 and 256 bits. Data is decrypted in 128 bit blocks.

Since the main focus of this assignment is to see the benefits of parallel computing, I will not focus a great deal on how AES works, instead just providing a simple diagram.

**Diagram 1:**

Key

Ciphertext

Plaintext

**7/11/2018 – Brute force techniques**

The design of the program will be based around a brute force tactic, this is when you attempt all possible character combinations until you find the once which works. This is an exhaustive method which can take a lot of time to complete, documentation named “Simplifications for encryption Key” has been provided for simplifying the encryption key to reduce the amount of possible character combinations.

Below are three different brute force algorithms which are being considered.

**Generate and test**

The generate and test method, would first create a potential key then attempt to decrypt the text with this key, then compare text which has been decrypted against original text, if the text matches, the program can stop, otherwise it repeats this process until text matches, or until all possible combinations have been tried.

**Diagram 2:**

Generate Key

Generate solution

Decrypt Ciphertext

Text didn’t match

Possible solution

Text matches

Compare Ciphertext with Original text

Program Completed

**Breadth-First**

Breadth first searches through combinations based on node level, starting at a simple layer and gradually moving down through all possible nodes on different layers. The layer represents the complexity level, Breadth first starts at the simplest level and works its way down to the most complex, this allows for easier combinations to be found first. To use this in our design each node would have to represent a potential character combination.

**Depth-First**

Depth first is essentially just the inverse of Breadth-first, it starts at the most complex node and backtracks through all less complex ones until the correct node is found.

Out of these three designs I have opted for the Generate and test design, since it seems the simplest to implement and the main focus of this project, is on the parallelization aspects.

**14/11/2018 – Program design plan**

In this section I will detail my initial program design, the details provided are based of knowledge obtained from the above research, lectures and weekly practical tasks provided on blackboard.

1. Create a program which can encrypt and then decrypt a simple message, using openSSL libraries.
2. Develop a serial brute force section which will generate keys based on the readable ASCII character set minus the space and delete character. Which then attempts to decrypt the message based on generated key.
3. Time and benchmark the results of the serial brute force method for later comparison.
4. Develop parallel version of the program using OpenMP to create threads and crack the message at a faster rate.
5. Time and benchmark the results of the OpenMP brute force method for later comparison.
6. Develop parallel version of the program using MPI to use multiple nodes on a cluster and crack the message at a faster rate.
7. Time and benchmark the results of the MPI brute force method for later comparison.

**16/11/2018 – Development of basic encryption/decryption program**

To start development of this section I have used tutorial provided in assignment specification (<https://eclipsesource.com/blogs/2017/01/17/tutorial-aes-encryption-and-decryption-with-openssl/>), this tutorial has shown me what is required to decrypt cipher using openSSL libraries. I have written my own simplified implementation of this decrypt function and tested against cipher, generated using command line instructions taken from “Simplifications for encryption key”.

One mistake I have made here is trying to decrypt a 256 bit cipher instead of a 128, since I have generated a 128 bit cipher, I had to use 128 bit decrypt function called “EVP\_aes\_128\_cbc()” in my decrypt initialization.

Each time I want to generate a different cipher text I did not want to have to use command line method, so I decided to make a encrypt function. This function ended up being very similar to my decrypt and since all the parameters would be the same. With this in mind I decided to turn these two functions into one by using a flag, which would indicate whether you were trying to decrypt or encrypt.

**20/11/2018 – Development of Serial brute force program**

Program steps:

1. Encrypt original text
2. Generate new key
3. Attempt to decrypt with generated key
4. Compare decrypted text with original, if it matches stop, if it doesn’t repeat from step 2

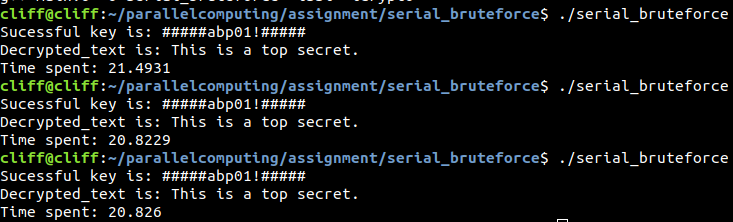
The key I have chosen to use, pads the first and last five characters with hashes and then for testing purposes the middle six characters will be only lower case characters. This is just for testing and debugging purposes, during performance analysis I want to generate harder keys and see what difference this makes to crack time.

To use diagram 2 as a starting point, I needed to find a way to generate every possible key combination from readable ASCII character set. To solve this problem I have used nested for loops which each refer to a location of a changeable character in my attempt\_key. In the lowest level for loop, I attempt to decrypt the encrypted text and compare the output against text which was previously not encrypted.

Since this a very extensive method it can take a very long to figure out the correct key, while testing I would ensure the first three characters looked for were ‘a’ and the first 3 letters in the key were ‘a’. This reduced down my search space by a lot and made it easy to test the program, each time I increased the complexity of the key it would increase my search time by around a factor of ten.

The main thing learnt from this step is knowledge of your search space, by knowing your key is simple and setting the search space up to start by looking at simple combinations you can save a lot of search time.

Output of three runs of serial program, cracking simple key:



**21/11/2018 – Development of OpenMP brute force program**

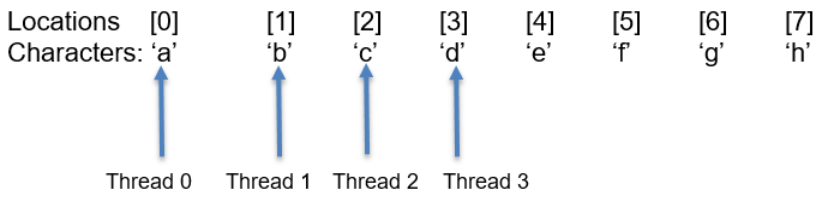
Program steps:

1. Encrypt original text
2. Start multiple threads
3. Generate different keys in different threads
4. Attempt to decrypt with generated key in each thread
5. Compare decrypted text with original in each thread, if it matches signal all threads to stop, if it doesn’t match repeat from step 3

The main problem to solve at this stage of development is, how to generate keys in different threads, these keys are different from one another and can’t don’t overlap one another. The reason you don’t want to overlap is trying a key which has already been tried will waste time and decrease performance of the search.

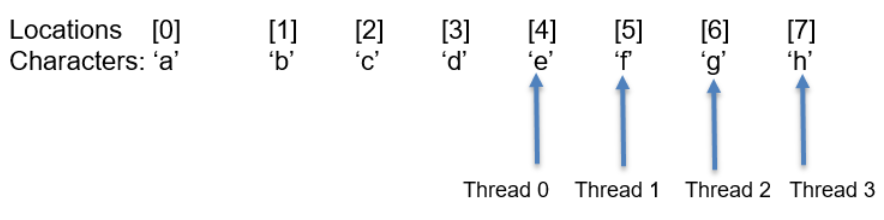
To solve this problem, I changed the starting point for each thread based on their thread ID. The first thread would start at location zero in my search space, then once it has searched through all combinations starting with letter at location zero, it would jump by the amount of threads to the next location. For example if you had four threads, the first thread would start at zero then four then eight and so on.

Visual representation of solved problem above, First iteration:



After each thread has gone through each possible combination it steps through the search space by four places and searches again. No overlap occurs.

Next iteration:

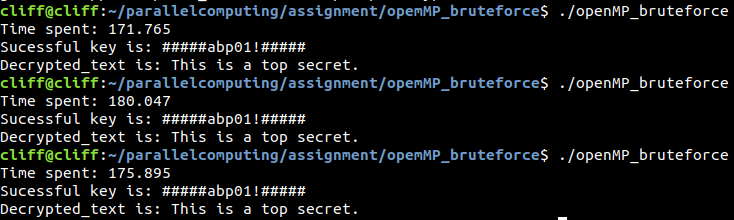


Another problem with this approach is if the search space is not divisible by number of threads then the program would end up stepping past the end of the ASCII array. This is because for loop only checks if number is less than size of array, but since we are stepping multiple places after the check it could end up stepping off the end and cause a segmentation fault.

A quick an easy way to solve this is, before addressing the location in the ASCII array to check if the value you are indexing is not larger than size of array.

One common aspect of threaded programming which I have learnt more about during this part of the project is, having to signal other threads to stop. Once the correct key has been generated by one of my threads, the other threads need to be made aware and stop looking. This is accomplished by having a flag which all threads have scope of, each thread checks this flag often and when it is set, they go to end program label.

Output of three OpenMP attempts at cracking simple key:



**22/11/2018 – Development of MPI brute force program**

Program steps:

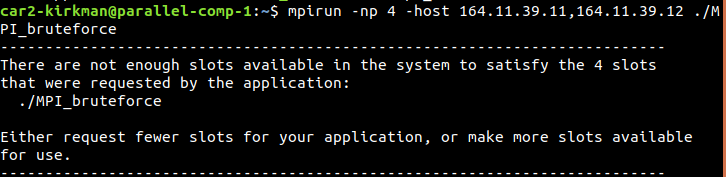
1. Encrypt original text
2. Start multiple nodes on different machines
3. Generate different keys in different nodes
4. Attempt to decrypt with generated key in each node
5. Compare decrypted text with original in each node, if it matches signal all nodes to stop, if it doesn’t match repeat from step 3

My first impression of implementing MPI program, was that it would be very similar to OpenMP since we are implementing a similar program, using different syntax which allows you to have separate nodes which in-turn have their own threads.

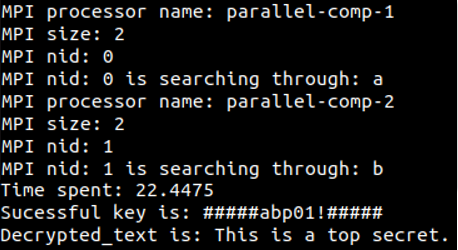
The hardest part of this implementation was not the coding, it was getting the program set up and running on a cluster. I have tried to create a host file containing the IP addresses of master and slave nodes, for an unknown reason using this host file with the argument –hostname would not work. I believe this could be due to a version mismatch, once I have identified this as the problem, I have used the command line argument –host followed by the IP addresses being used.

The same problem occurs with threads overlapping search area as OpenMP, to solve this problem I used the MPI\_Comm\_rank function to get identifier of the thread and use this as my starting point for searching my ASCII character set.

During this implementation I have also had a problem with threads becoming out of sync, to solve this I added MPI\_Barrier which will stop all threads from continuing until they have reached this step. Important thing to mention is when adding MPI\_Barrier you must insert it into an area which all threads would reach or you could leave some stranded in this position waiting for threads which will never arrive.

Unfortunately I was only able to make MPI work with one master and one slave, when trying to add nodes I also ran this problem.

Output of MPI cracking simple key:



**25/11/2018 – General program improvements**

During the project I created Makefile for each variant of the program, as a last improvement I decided to create a higher level Makefile which calls all of the lower level ones to create all binaries at once. This could save someone time when first compiling the project. It also provides the ability to clean all projects at once.

One realization I had during this project was that, print outs slow down the programs solution generation drastically. To help solve this problem well still being able to enable print outs when needed I added a verbose option to each program. This only enables additional log output when using the –v command line flag.

**02/12/2018 – Benchmarking and performance analysis of all three programs**

When considering the parallelization of programs, the developer is considering one of two things, to speed up the programs run time or accomplish multiple different tasks simultaneously. In this section I will break down why my program has failed to speed up solution generation of keys starting from ‘a’ and greatly increased the speed of solution generation from keys starting from any other character. One very important factor to remember during this, is that my programs start searching from the start of an array full of all possible ASCII characters which starts from ‘a’.

**Diagram 3:**

As shown in the table above my sequential program out performed my parallel variants, I believe the main reason for this architecture I was running the programs on. On my host machine I only have two cores, so running a program with four threads was expected to be the optimal amount. Although you still have to consider the negative effect of communication parameters.

The serial brute force program is only evaluated by its asymptotic runtime as a function of its input size. Although it was only marginally better than MPI brute force.

Another thing to consider was I knew my search space so when creating a key to be cracked, in the first two cases above the first character would always be ‘a’ to stop the number of possible solutions increasing by another factor of 92.

I was unable to obtain a result of the serial program cracking a key which began with a different letter other than ‘a’ because it increased the time to much, my host machine take from days to months depending on where in the search space I would start the key from. I have tried to solve the key ‘baf01!’ with serial solution but after twenty four hours of running it was still unable to solve it, the serial solution would take much longer than both parallel variants to work out keys starting from letters other ‘a’, since the parallel solutions running four threads generate keys starting from the first four characters of the search space at the same time.

**Diagram 4:**

From the above results you can see MPI was faster by 2.54 seconds, since the code is almost identical and these attempts were both run with two threads/nodes my deduction is that the CPU speed on the cluster is faster than my host machine. Put simply the underlying platform on the cluster has a better implementation.

Another possible reason for the above results could be differences in communication time between types, the OpenMP may take longer to notify the threads that it has finished than MPI.

In these programs there is not much overhead generated since they are all relatively simple programs which don’t have much if any thread idle time. One source for overhead would be communication time when considering the parallel variants.

To summarize this section because of the way my programs utilize threads to generate solutions, my serial program is much better at solving simpler solutions and my parallel programs are much better at solving more complex solutions. In an environment where the program is not able to narrow down the search space, the parallel algorithms will perform better, 99.9% of the time.

**05/12/2018 – Future improvements and different algorithm theories**

During the last day of this assignment I have had an idea for another solution, relating to the brute force algorithm. Instead of relying on thread/node ID and amount to step through the search space, I would divide the search space up by number of threads/nodes and allow each thread to search through there section. If the search space wasn’t divisible by number of threads/nodes then it would be likely some threads would finish searching before one another. Threads which have finished their section could then take the remaining half of another threads section to help speed up the crack.

**References**

OpenMP website: <http://www.openmp.org>

MPI website: <https://www.mpi-forum.org>

Chapman, B (2008) Using Openmp : Portable Shared Memory Parallel Programming. Cambridge, Massachusetts London, England: The Mit Press.

Hoefler, T. (2013) Mpi + Mpi: A New Hybrid Approach to Parallel Programming with Mpi Plus Shared Memory. Computing [online]. 95 (12), pp. 1121-1136. [Accessed 22 November 2018].