2018

# Log Book and Performance Analysis

PARALLEL COMPUTING CW1

LAURENCE BURTON (15003639)

## Contents

## Introduction

For this assignment the class was assigned to use the OpenSSL library to implement an encryption and decryption function. Then implement a brute force to search for the key that had been used. Once this was implemented the brute force algorithm had to be reimplemented in parallel using OpenMP and once again in MPI. During the project they had to keep a log of how they progressed through the assignment. After they had finished the tasks they had also do a performance analysis comparing serial to parallel.

## Logbook

During the assignment I kept a log off how I contributed towards the assignment each day.

### 12/11/2018

Today I went through the assignment specification to see what I needed to do. I then investigated AES encryption to see what I needed to implement.

Website used: https://thebestvpn.com/advanced-encryption-standard-aes/

### 13/11/2018

I spoke to Alex on my course about how he implemented encryption and decryption. He told me about the C library libcrypto. I used an example from there website, but I couldn't get any of them working. I decided to have a look at the OpenSSL library but didn't have time to implement anything.

Website used: https://wiki.openssl.org/index.php/Libcrypto_API

Website used: https://www.openssl.org/

### 17/11/2018

After further looking into OpenSSL I discovered that libcrypto was part of the OpenSSL library. I managed to use a guide from the OpenSSL website to implement an encryption and decryption function in C.

Website used: https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption

### 18/11/2018

I spent most of today trying to implement a function which would be able to generate every 6-letter combination using the alphabet. My original method was to create a function which would be recursive and increment through all the words. Unfortunately, after doing some research and trying to implement it. I wasn't quite able to generate every word. I found that some letters where getting's skipped. I then spoke to my cryptography lecturer and looked at her PowerPoints on the topic and decided to implement a simple version which I knew would work. This method would use 6 embedded for loops. Each loop would increment through an array containing the characters from 0 to 9. In each, For-Loop I would use that value that was being incremented as the position in the array and allocate it to a different position in the six-letter character array.

Once I managed to get to generate every word I had to pad the word with 10 # on each side because we were working a doing a 128-bit encryption which meant we had to use a 16bit key. To do the padding I created an array of 16 characters. I made the first and last 5 characters # and the middle six I made 0. In the for loop, I would only work with the middle 6 characters. If I wanted to up the number of characters in the loop I would have had to add an extra loop. This was the reason I didn't want to use embedded loops.

In the middle loop, I made an if statement which would compare the key that I generate to the key that was used to encrypt the data. I originally used the function strcmp() to compare the two keys. But I found that this wasn't reliable as it would use the null terminator at the end of the key to find the length of the strings. The function I used to replace this was strncmp(). This has a third parameter for the length of the string. This fixed that bug.

## 30/11/2018

I set up a new function which would print out how long the process would take.

I tested the serial function and found that it took, 0 Seconds: 22068 Milliseconds.

I then convert my serial into OpenMP. I inserted a pragma statement around the For-Loop. Bellow, you can see the average results that I got for different amount of threads.

Tests results:

1 thread. Time: 0 Seconds: 22607 Milliseconds

2 threads: Time: 0 Seconds: 23336 Milliseconds

3 threads: Time: 0 Seconds: 838 Milliseconds

4 threads: Time: 0 Seconds: 843 Milliseconds

## 2/12/2018

After running some tests on my code, I found that my parallel wasn't as quick as I expected so I moved the brute force code into a function call. This allowed me to exit the function once a correct key had been generated. I then found that you're not able to stop other threads. To overcome this, I created a flag before the #pragma. This meant that every thread would be able to see it. I then passed the address of the flag to each function call. If any of the threads found the correct key I would set the flag to one. Every time another thread generated another word it would check if this flag equals 1. If it did it would return to the main.

I'm still not happy with how it was implemented as some of the data was shared between the threads. To overcome this, I used an OMP Lock whenever a thread would write to a shared variable. To make the parallel work better I made my For-Loops uses the thread ID to initialize the counter and then incremented using the number of threads.

## 3/12/2018

I did some testing of my parallel brute force. I found it didn't work like I thought it did. Because I was using the thread ID as a starting point for my For-Loops and the number of threads to increment through each For-Loop. It wasn't doing every combination like I thought it was. I discovered this when I change my key. To overcome this, I only set the first For-Loop to use the thread ID as the position and number of threads to increment. This stopped the code from skipping certain words.

I then tested the timing of one thread versus four threads. I found that the one thread took 0.5 seconds and 4 threads took 0.7 seconds. I wasn't happy with this and I felt that the reason that it slowed down is that it was sharing data between threads. So, to fix this I used omp_lock_t. However, this made it slower as well because it would stop every thread from working when the lock is set. I then had a look at how I can remove the lock and stop data from being shared. To do this I used strcpy() to copy the key so that every variable has its own copy to work with. Unfortunately, this also didn't help and I still need to use a lock when copying the data.

## 4/12/2018

After taking some time to think about it and rerunning the code I noticed that one threaded method was slightly slower. This made me think that I should have a look at how my timer was working. I reimplement my timer using omp_get_wtime() instead of clock(). After retesting both methods I found four threads was faster with 0.187025 whereas one thread took 0.475636.

I tidied my files added them to GitHub and set up a .gitignore. To stop the execution files from getting uploaded.

I used a hello world program that was given to us and ran it on the cluster. From this, I was able to convert my OpenMP code to MPI and get it running. An issue I've found is that the whole program is run on each process on MPI not one like OpenMP. Where only the parts in between the #pragmas run multiple time. This causes the program to encrypt the plaintext on every machine and print out the cypher.

I fixed this by inserting an if statement to check if the current process is equal to zero. If it was it would print the cypher text. But now I've found the other process keep on going and will try and decrypt before it's been printed. Away I could overcome this is by inserting a barrier which will cause all the process's to wait until it's been printed.

Website used: https://www.mpich.org/static/docs/v3.2/www3/MPI_Barrier.html

Website used: http://mpi.deino.net/mpi_functions/MPI_Barrier.html

I now have a bug where all the threads will try keep running the brute force until the loop finishes. When using OpenMP I managed to get around this by creating a flag. Because it was a pointer that was created before the pragmas. it meant that all the other threads could see this and would also end their loops and return to main.

Website used: http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

I found out that MPI had some built-in function that would allow me to communicate between different process.

MPI_Send() and MPI_Recieve(): This would allow me to send a message from one process to one other process. The idea I had to do for this was for each process to send a message to the next process. I tried to implement this but unfortunately, I got an error saying bad id and decided to move onto another function.

MPI_Bcast(): This method would allow for one process to broadcast a message to every other process. I found when using this my programme would hang until the other threads received a message. I then made every process broadcast a message and receive a message on every loop. Unfortunately, I also got an error message and ran out of time to try and fix it.

After trying these two methods and not being able to fully implement either I decided to end the program once one of the threads had found a match using MPI_ABORT ()

After getting my program to run I did some further research into MPI and discovered that my program was only running on one computer within the cluster and creating four processes on it. I did this by printing the hostname of the machine from each process.

Whilst on the cluster I used the following command to identify the hostnames and the IP address of all the computers within the cluster.

```
cat /etc/hosts
```

In MPI you're able to choose how many computers you want to run your program on and how many threads you want on each of them. I found that there was two ways to do this. You can put it into the comment when you execute your program, or you can create a host file which contains all the hostnames/IP addresses and the number of slots each machine would have.

I prefer the idea of creating a host file. When I implement this, it asked me to enter my password for each of the machines that I was trying to connect to. This would be fine, but it would glitch and ask me for all the password on one line at the same time. This meant that I had to enter the password for all the machines. I then tried to specify the hosts using the execution command. When I had only specified two machines it didn't glitch, but I wasn't able to login. I discovered this was because I had to SSH into the machine using the master machine and reset my password before I could use the machine with MPI.

## 5/12/2018

A major issue I found with MPI was that when you set up a Hostfile it asks you to enter your password. If you had more than two machines it would ask you to enter every password but all at once and end up glitching. I tried to generate an SSH key for each machine and add the public key to the host machine. This should have allowed for me to connect to each machine without entering a password. Unfortunately, I couldn't get this to work. Due to time restriction, I decided to only use two machines (the host and a client). This would stop the glitch as it would only ask for one password.

Now that I have managed to get it running on multiple machines (but not four) I found that I got an error message as I didn't have my code on the client. I SSH into the client and cloned my Git repo. I then had to run the make command to produce the execution files.

## Benchmarking and performance analysis

### Benchmarking

For the benchmarking and performance analysis each program was ran five times and then an average time was generated. OpenMP was run with 4 threads whereas MPI was run on 2 nodes with 4 processes on each node.

Below is a table with the results for brute forcing the key: 123456.

Serial

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.127662 | 0.0615431 |
| 2 | 0.128796 | |
| 3 | 0.138095 | |
| 4 | 0.112965 | |
| 5 | 0.107913 | |

OpenMP

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.087053 | 0.0463202 |
| 2 | 0.102497 | |
| 3 | 0.091007 | |
| 4 | 0.089601 | |
| 5 | 0.093044 | |

MPI

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.094368 | 0.0374779 |
| 2 | 0.059844 | |
| 3 | 0.078498 | |
| 4 | 0.058837 | |
| 5 | 0.083232 | |

Below is a table with the results for brute forcing the key: 999999.

Serial

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.127662 | 0.0615431 |
| 2 | 0.128796 | |
| 3 | 0.138095 | |
| 4 | 0.112965 | |
| 5 | 0.107913 | |

OpenMP

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.228099 | 0.1064114 |
| 2 | 0.203209 | |
| 3 | 0.200953 | |
| 4 | 0.217652 | |
| 5 | 0.214201 | |

MPI

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.084749 | 0.0459594 |
| 2 | 0.066953 | |
| 3 | 0.112216 | |
| 4 | 0.093739 | |
| 5 | 0.101937 | |

Below is a table with the results for brute forcing the key:969696.

Serial

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.127662 | 0.0615431 |
| 2 | 0.128796 | |
| 3 | 0.138095 | |
| 4 | 0.112965 | |
| 5 | 0.107913 | |

OpenMP

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.212668 | |
| 2 | 0.191975 | |
| 3 | 0.196291 | |
| 4 | 0.215808 | |
| 5 | 0.199365 | |

MPI

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.237731 | 0.0916044 |
| 2 | 0.13468 | |
| 3 | 0.141899 | |
| 4 | 0.219315 | |
| 5 | 0.182419 | |

Below is a table with the results for brute forcing the key: 765432.

Serial

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.127662 | 0.0615431 |
| 2 | 0.128796 | |
| 3 | 0.138095 | |
| 4 | 0.112965 | |
| 5 | 0.107913 | |

OpenMP

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.176171 | 0.0948393 |
| 2 | 0.195713 | |
| 3 | 0.186501 | |
| 4 | 0.189108 | |
| 5 | 0.2009 | |

MPI

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.417535 | 0.2260065 |
| 2 | 0.417535 | |
| 3 | 0.522297 | |
| 4 | 0.481587 | |
| 5 | 0.421111 | |

Below is a table with the results for brute forcing the key: 594232.

Serial

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.156714 | 0.0881324 |
| 2 | 0.187735 | |
| 3 | 0.192742 | |
| 4 | 0.168023 | |
| 5 | 0.17611 | |

OpenMP

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.155492 | 0.0833797 |
| 2 | 0.170607 | |
| 3 | 0.176794 | |
| 4 | 0.165803 | |
| 5 | 0.165101 | |

MPI

| Attempt | Time | Average |
|---|---|---|
| 1 | 0.109277 | 0.050819 |
| 2 | 0.110154 | |
| 3 | 0.098767 | |
| 4 | 0.119271 | |
| 5 | 0.070721 | |

## Performance Analysis

To do the performance analysis Quinn's Law was used. From this, we can identify how efficient the it was to parallelise the brute fource with OpenMP and MPI. The following equations was used:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$

Result for key: 123456

OpenMP: 0.332161%
MPI: 0.205264636%

Result for key: 999999

OpenMP:  0.258527282
MPI: 0.299288611

Result for key: 969696

OpenMP: 0.27189213
MPI: 0.150795977

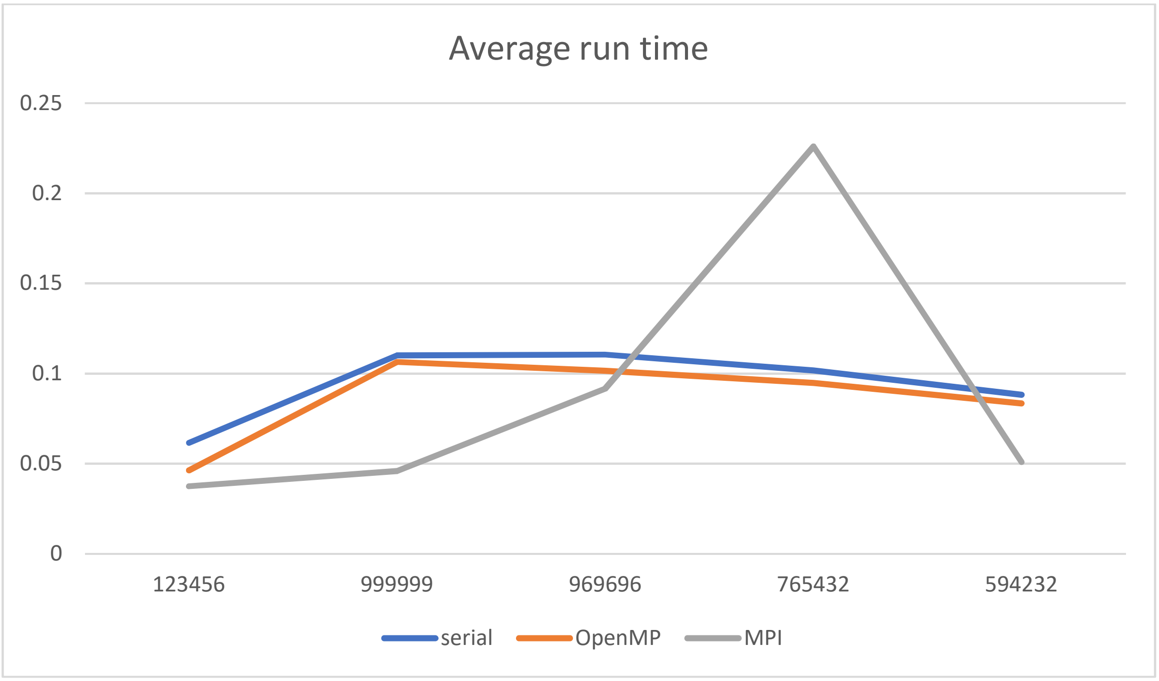
Result for key: 765432

OpenMP:  0.267924531
MPI:  0.0562147

Result for key: 594232

OpenMP: 0.26425
MPI: 0.21678

In the chart below, you can see average runtime for each program and each key.



From the results of the Quinn's law and the table above I can see that MPI is generally faster but there is an anomaly within my code as there is a huge spike with in the chart. This suggested that there is a bug within my code which results in some keys taking longer to be found.

## Conclusion

Throughout this assignment, a major part which should have been done differently was the comparison of the key to determine if you got the correct key because if you had the original key you wouldn’t need to brute force it. An alternative method would be to encrypt the plain text and check if the cypher text was the same. However, this wasn’t much better because if you knew the plaintext you wouldn’t need to decrypt it. A better solution is to decrypt the cypher text every time a new key was generated. Then check if the result was made up of normal ASCII characters.