

# **AES ENCRYPTION PASSWORD CRACKING THROUGH PARALLELISATION**

UFCFFL-15-M - Parallel computing 20jan\_1

## **Coursework Logbook**

Prepared by

Kiran Kumar Mohanraj - 20054954

MSc Cyber Security

University of the West of England, Bristol

## TABLE OF CONTENTS

Introduction	-----	3
Gitlab Links	-----	4
Logs of Progress	-----	5
Method of Parallelisation	-----	14
Performance Analysis	-----	15
Conclusion	-----	21
Reference	-----	21

## INTRODUCTION

Parallel computing has gained much significance in recent years to deal with the increasing complex computational problems. High computation power is yielded through parallelisation, large complex problems are divided into small chunks which are executed at same time, thus reducing the computational time (Belhaous et al., 2020). In Engineering field, parallelization has major impact in design, modelling and application development with the help of parallel computers. Even in commercial applications, parallel platforms like Linux clusters are widely for web and database servers (Grama et al., 2003).

Parallel Programming is the toughest part in parallelisation, since complexity increases than in sequential programming. The two important parallel programming languages are OpenMP and MPI. OpenMP is widely used by developers which works on threading and provides more control. Whereas, MPI provides more scalability. Implementing these programs reduces memory usage, load imbalance and communication costs (Chapman et al., 2008).

This report provides performance analysis of parallel programming against serial programming in cracking the password of AES encryption. AES encryption is one of the best encryption methods. It is even used for communicating top secret information in military. The only way to break the encryption is through brute-force approach. This report explains the method of parallelising the serial program. Finally, it compares the performance of serial programs with the parallel programs – OpenMP and MPI.

## **GITLAB LINKS**

Logbook:

[Documentation · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

Serial:

[Serial · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

OpenMP:

[OpenMP · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

OpenMPI:

[OpenMPI · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

Readme:

[README.md · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

Video:

[Video · master · kk2-mohanraj / Parallel Computing Coursework · GitLab \(uwe.ac.uk\)](#)

## LOGS OF PROGRESS

### Session-1 (OpenMP) – 20<sup>th</sup> March 2021

To implement the AES password cracking using OpenMP, I researched about the concepts of OpenMP. After gaining some knowledge on OpenMP, I saw Dr Kun Wei's video explaining the example serial code. I understood the code and then I thought that parallelisation needs to be done in the for loops because it is the only part of the code which runs repeatedly. I tried running the serial code and got the results as shown in Figure 2.

```
for(int i=0; i<dict_len; i++)
    for(int j=0; j<dict_len; j++)
        for(int k=0; k<dict_len; k++)
            for(int l=0; l<dict_len; l++)
                for(int m=0; m<dict_len; m++){
                    *password = dict[i];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];

                    initAES(password, salt, key, iv);
                    unsigned char* result = decrypt(ciphertext, cipher_len, key, iv);

                    if (success == 1){
                        if(checkPlaintext(plaintext, result)==0){
                            printf("Password is %s\n", password);

                            time_t end = time(NULL);
                            printf("Time elapsed is %ld seconds", (end - begin));

                            return 0;
                        }
                    }
                }

            free(result);
```

Figure 1 – Part of serial code where Parallelisation needs to be implemented

```
Password is 12345
Time elapsed is 18 seconds
```

Figure 2 – Result of Serial code

## Session-2 (OpenMP) – 30<sup>th</sup> March 2021

In this session, I wanted to try a different approach for generating the password before OpenMP implementation. So, I created a passwordGen() function (as shown in Figure 3) which uses rand() function for choosing the letters from the dictionary. For testing, the ciphertext with password – 12345 was used with the dictionary “0-9”. Instead of using the for loop repeatedly, main() will call the passwordGen() function for getting the password string. Figure 4 shows implementation of password cracking in main() function.

```
int passwordGen(char* password, char dit[], int dit_len)
{
    int i;
    for(i=0;i<5;i++)
    {
        int x = rand() % dit_len;
        *(password+i)=dit[x];
    }
}
```

Figure 3 – Password generating function

```
while(flag=1){

    passwordGen(password,dict,dict_len);
    initAES(password, salt, key, iv);
    unsigned char* result = decrypt(ciphertext, cipher_len, key, iv);

    if (success == 1){
        if(checkPlaintext(plaintext, result)==0){

            printf("Password is %s\n", password);
            time_t end = time(NULL);
            printf("Time elapsed is %ld seconds", (end - begin));
            flag=1;
            return 0;
        }
    }

    free(result);
}
```

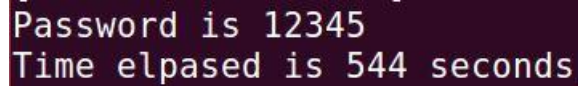
Figure 4 – Password cracking part in main() function

```
Password is 12345
Time elapsed is 0 seconds
```

Figure 5 – Result of the above testing.

### Session-3 (OpenMP) – (2<sup>nd</sup> – 8<sup>th</sup> April 2021)

In this session, I tried increasing the dictionary to “A-Z,” “a-z” and “0-9”. As shown in Figure 6, the password cracking time increased rapidly since password generation is random and the dictionary length has increased.



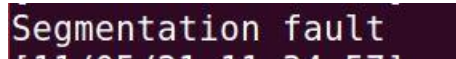
```

Password is 12345
Time elapsed is 544 seconds

```

Figure 6 – Result of testing with improved dictionary

With the serial version working perfectly, I planned to implement the OpenMP to parallelise the passwordGen(). Implementing a parallel region inside the while loop caused error. Even after the interaction with Professor Andrew McCathy, I wasn't able to rectify the error and implement the parallel region.



```

Segmentation fault

```

Figure 7 – Error while implementing OpenMP

With the failure of my approach, I thought of improving the example serial code and implement OpenMP. I used the Parallel loops concept for parallelising the for loops. Figure 8 shows the parallel for loop which breaks down the outer for loops and assigns equal work to the threads.



```

omp_set_num_threads(2);
#pragma omp parallel for
for( int i=0; i<dict_len; i++)
    for( int j=0; j<dict_len; j++)
        for( int k=0; k<dict_len; k++)
            for( int l=0; l<dict_len; l++)
                for( int m=0; m<dict_len; m++){

                    *password = dict[i];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];

                    printf("%s\n", password);
                }

```

Figure 8 – Parallel For loop

When the above code is executed, the terminal window crashed and stopped after running the code for long time. To test the execution, I printed the password to check the output. Then, I found out that each thread causes collision and does not match the plain text. When the password was printed, it had some time to execute the decryption and comparison without collision.

```
1233z
12340
12341
12342
12343
12344
12345
Password is 12345
Time elapsed is 265 seconds
```

Figure 9 – Password found without collision

#### Session-4 (OpenMP) – (9<sup>th</sup> – 11<sup>th</sup> April 2021)

To improve the performance of OpenMP, I used the schedule clause which breaks down the iterations of the for loops and assigns it to the threads. While executing this code, I got random results and even sometimes I got Segmentation fault.

```
omp_set_num_threads(2);
#pragma omp parallel for schedule(static,1)
for( int i=0; i<dict_len; i++)
    for( int j=0; j<dict_len; j++)
        for( int k=0; k<dict_len; k++)
            for( int l=0; l<dict_len; l++)
                for( int m=0; m<dict_len; m++){
                    *password = dict[i];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];
```

Figure 10 – Code with schedule clause



After a session with Professor Andrew McCathy, I came to know that the parallel region uses the key and iv as shared variable because of which collision occurs while generating key and iv. To overcome this, I used private() clause for key, iv and ciphertext. This produced Segmentation Fault. Since the ciphertext is initialized above the parallel region, I changed the private() to firstprivate(), then the code produced expected results.

```

omp_set_num_threads(2);
#pragma omp parallel for schedule(static,2) firstprivate(ciphertext,key,iv)
    for( int i=0; i<dict_len; i++)
        for( int j=0; j<dict_len; j++)
            for( int k=0; k<dict_len; k++)
                for( int l=0; l<dict_len; l++)
                    for( int m=0; m<dict_len; m++){

                        *password = dict[i];
                        *(password+1) = dict[j];
                        *(password+2) = dict[k];
                        *(password+3) = dict[l];
                        *(password+4) = dict[m];

                        printf("%s\n", password);

```

Figure 11 – Code with firstprivate() clause

The above code works only when we print the password because printing the output provided some time to avoid collision. To overcome the problem, I tried using critical clause so that only one thread can access the critical region which is decryption and comparison. This too didn't provide the expected result and printed the random password.

After working out some concepts, I found that collision occurs in the password variable. So I initialized the password variable to 00000, whenever each thread access the variable it does not overwrite/collide with the other threads. This produced the expected result with reduced execution.

```

Password is 12345
Time elapsed is 1 seconds

```

Figure 12 – Result of crackaes.c using OpenMP

```

omp_set_num_threads(2);
#pragma omp parallel for schedule(static,2) firstprivate(ciphertext,key,iv)
for( int i=0; i<dict_len; i++)
    for( int j=0; j<dict_len; j++)
        for( int k=0; k<dict_len; k++)
            for( int l=0; l<dict_len; l++)
                for( int m=0; m<dict_len; m++){

                    unsigned char plainpassword[] = "00000";
                    unsigned char* password = &plainpassword[0];

                    *password = dict[i];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];

                    initAES(password, salt, key, iv);
                    unsigned char* result = decrypt(ciphertext, cipher_len, key, iv);
                    #pragma omp critical

                    if (success == 1){
                        if(checkPlaintext(plaintext, result)==0){

                            printf("Password is %s\n", password);
                            time_t end = time(NULL);
                            printf("Time elapsed is %ld seconds", (end - begin));
                            exit(0);
                        }
                    }
                    free(result);
                }
    }
}

```

Figure 13 – OpenMP parallel region

## Session-5 (OpenMP) – (17<sup>th</sup> April 2021)

Since the password is of variable length (4-6), I added two more parallel region, one for finding the 4-length password and another for 6-length password. To implement different dictionary, I had a idea of using Frequency Distribution of English letters. Figure 14 shows the graph of frequency distribution of English letters.

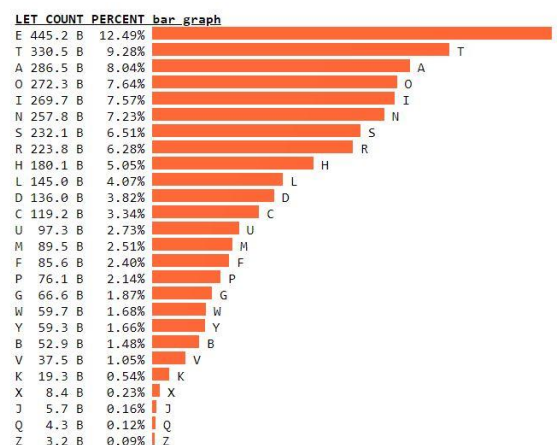
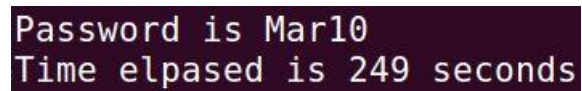


Figure 14 – Frequency Distribution of English Letters

The defined dictionary is

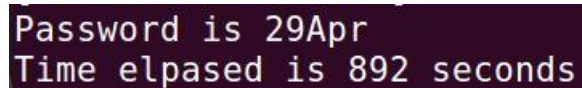
“ETANOISHRDLUCMWFYGPBVKXJQZetanoishrdlucmwfygpbvqxjqz0123456789”

To test out the code, I ran the code with various ciphertext got the results as shown below.



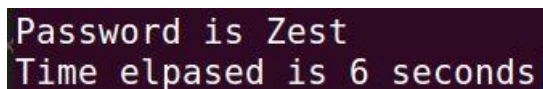
```
Password is Mar10  
Time elapsed is 249 seconds
```

Figure 15 – Result of the code with 2 thread



```
Password is 29Apr  
Time elapsed is 892 seconds
```

Figure 16 – Result of the code with 2 thread



```
Password is Zest  
Time elapsed is 6 seconds
```

Figure 17 – Result of the code with 2 thread

## Session-6 (OpenMPI) – (20<sup>th</sup> - 25<sup>th</sup> April 2021)

In this session, I researched about OpenMPI and ways of implementing it to parallelize the serial code. The OpenMPI uses a shared communication channel to communicate between the processor. To parallelise the for loop, I implemented the use of rank associated with each processor. Using MPI\_Irecv, we can receive the results posted by each process. The outer for loop is segregated using rank according to the number of processors provided. Once the password is found we need to inform all other processor to stop generating password. MPI\_Bcast is used to broadcast that a processor has found the right password. To quit the processor's work, we use MPI\_Abort to terminate the process.

```

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &inc);

MPI_Irecv(&rbuf, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &req);

for(int i = myrank; i < dict_len; i = i + inc){
    for(int j=0; j<dict_len; j++){
        for(int k=0; k<dict_len; k++){
            for(int l=0; l<dict_len; l++){
                for(int m=0; m<dict_len; m++){

                    *password = dict[i];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];

                    initAES(password, salt, key, iv);
                    unsigned char* result = decrypt(ciphertext, cipher_len, key, iv);

                    if (success == 1){
                        if(checkPlaintext(plaintext, result)==0){
                            MPI_Bcast(&sbuf, count, MPI_INT, myrank, MPI_COMM_WORLD);
                            printf("Password is %s\n",password);

                            time_t end = time(NULL);
                            printf("Time elapsed is %ld seconds", (end - begin));
                        }
                    }
                }
            }
        }
    }
}

```

Figure 18 – OpenMPI implementation in serial code

To test out the code, I ran the code with various ciphertext got the results as shown below.

```
Password is 12345  
Time elapsed is 1.441448 seconds  
-----  
MPI_ABORT was invoked on rank 1 in communicator MPI_COMM_WORLD  
with errorcode 1.  
  
NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.  
You may or may not see output from other processes, depending on  
exactly when Open MPI kills them.  
-----
```

Figure 19 – Result of the code with 4 processor

```
Password is Mar10  
Time elapsed is 199.417855 seconds  
-----  
MPI_ABORT was invoked on rank 1 in communicator MPI_COMM_WORLD  
with errorcode 1.  
  
NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.  
You may or may not see output from other processes, depending on  
exactly when Open MPI kills them.  
-----
```

Figure 20 – Result of the code with 4 processor

## **METHOD OF PARALLELISATION**

Parallelisation is the process of designing the logic of the algorithms with which parallel processing to be conducted. Based on the design of algorithms, parallelism can be classified into Data Parallelism and Task Parallelism. Data parallelism is breaking down the data and parallelising the algorithm. In Task parallelism, we decompose the task which can run independently and then implement parallelism. According to Flynn's Taxonomy, the computer architecture can be classified as SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data). To achieve Parallelisation, the algorithm is either decomposed as Data or Task and implemented in any of the above-mentioned computer architecture.

To crack the AES encryption, we use brute-force algorithm. This algorithm is based on trying out each and every possible password to crack the encryption. The flow of the program is, initialising the required variables, decoding the Base64, Salt removal and brute-force algorithm with decryption. In brute-force algorithm, a password string of variable length (4-6 length password) is generated with the defined dictionary. This password is used to decrypt the cipher text and if the decrypted plain text matches the original plain text, then that password is the original password used for the AES encryption.

From the above-mentioned flow of the program, the variable initialization, Base64 decoder and Salt removal can only be implemented in serial manner because these are done only once during the execution. Parallelisation that can be implemented in brute-force algorithm using the SIMD architecture is Data Parallelisation. Data Parallelisation is used because in the algorithm, password generation involves processing of different data from the defined dictionary. The SIMD architecture is implemented because the single instruction – password generation, decryption and comparison with original plaintext, is implemented parallelly and it involves using multiple data where each thread searches the dictionary and generates unique password for decryption.

## PERFORMANCE ANALYSIS

For analysing the performance of the parallel implementation, we consider 5-length password which is Mar10. The dictionary with Frequency distribution of English letters is used.

Forward dictionary –

"ETANOISHRDLUCMWFYGPBVKXJQZetanoishrdlucmwfypbvqxjqz0123456789"

Backward dictionary –

"9876543210zqxkvbpgyfwmculdrhsionateZQJXKVBPGYFWMCULDRHSIONATE"

The tables shown below provides the testing results for various number of threads and processor used for implementing parallelization.

### OpenMP Testing:

<i>Test</i>	<i>Threads</i>	<i>Forward</i>	<i>Backward</i>
<i>Serial</i>	-	258	1045
<i>OpenMP</i>	2	249	706
<i>OpenMP</i>	4	222	739
<i>OpenMP</i>	6	333	763
<i>OpenMP</i>	8	170	751

### OpenMPI Testing:

<i>Test</i>	<i>Processes</i>	<i>Forward</i>	<i>Backward</i>
<i>Serial</i>	-	258	1045
<i>OpenMPI</i>	2	207.143	564.942
<i>OpenMPI</i>	4	221.915	648.989
<i>OpenMPI</i>	6	184.970	675.012
<i>OpenMPI</i>	8	135.352	739.726

## OpenMP Analysis:

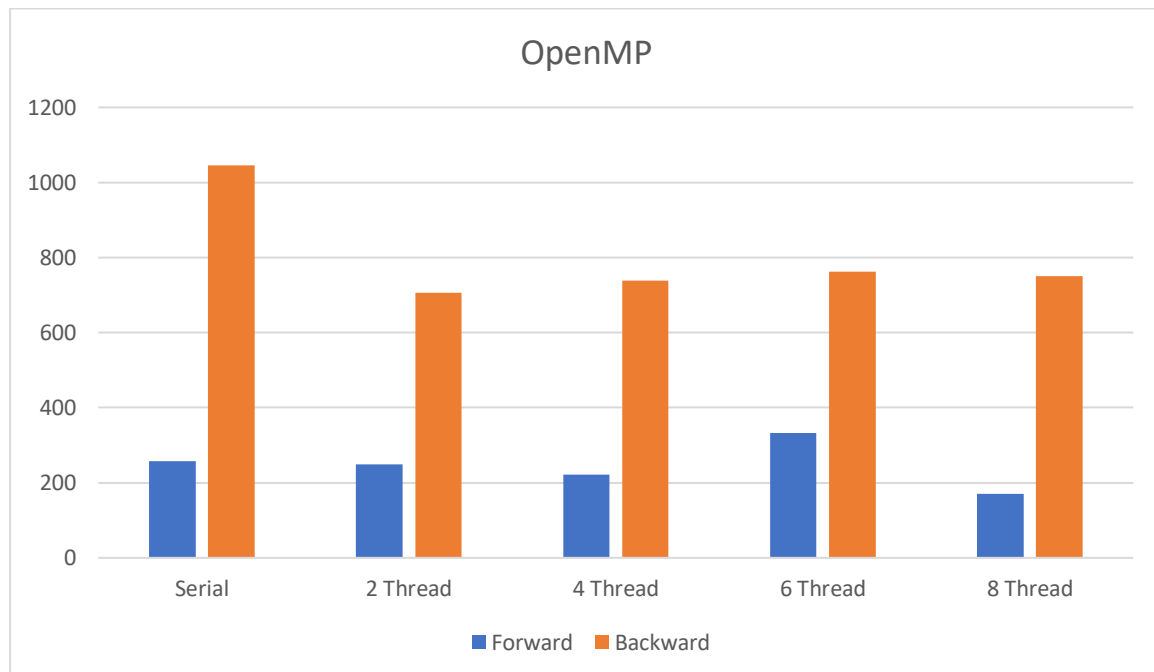


Figure 21 – Graph of Execution time vs Dictionary in OpenMP

At first, the serial version was tested with the forward and backward dictionary. The Backward dictionary took a long time to crack the password since the letter “M” of Password – Mar10 occurs towards the end of the dictionary. Then, the OpenMP version of the code was executed initially with 2 threads. The results of backward dictionary showed a drastic change in execution time. Whereas, the forward dictionary showed minimal time reduction.

The code was then executed with 4 threads and 6 threads, the execution time of both the dictionary increased slightly. In Theory, when the number of threads increases the execution time should decrease. This contradicts the results because of the hardware configuration of my laptop. When the code was executed with 8 threads, the forward dictionary produced the least execution time because the data processing has increased, it outweighs the instruction processing time and wait time. Whereas, the backward dictionary showed more time than 2 threads and 4 threads.

The one of the reasons for drastic time difference between the forward and backward dictionary is because of the use of `schedule(static,2)`, each thread takes the even position first executes all iteration and then moves to even position. For example, Thread 0 takes  $i=0$ , Thread 1 takes  $i=2$ , etc. Since the position of letter “M” occurs in even position in the forward vector, it is found faster.

From the above testing results, we can infer that 2 Thread implementation provided optimum execution for both the forward and backward dictionary.



## OpenMPI Analysis:

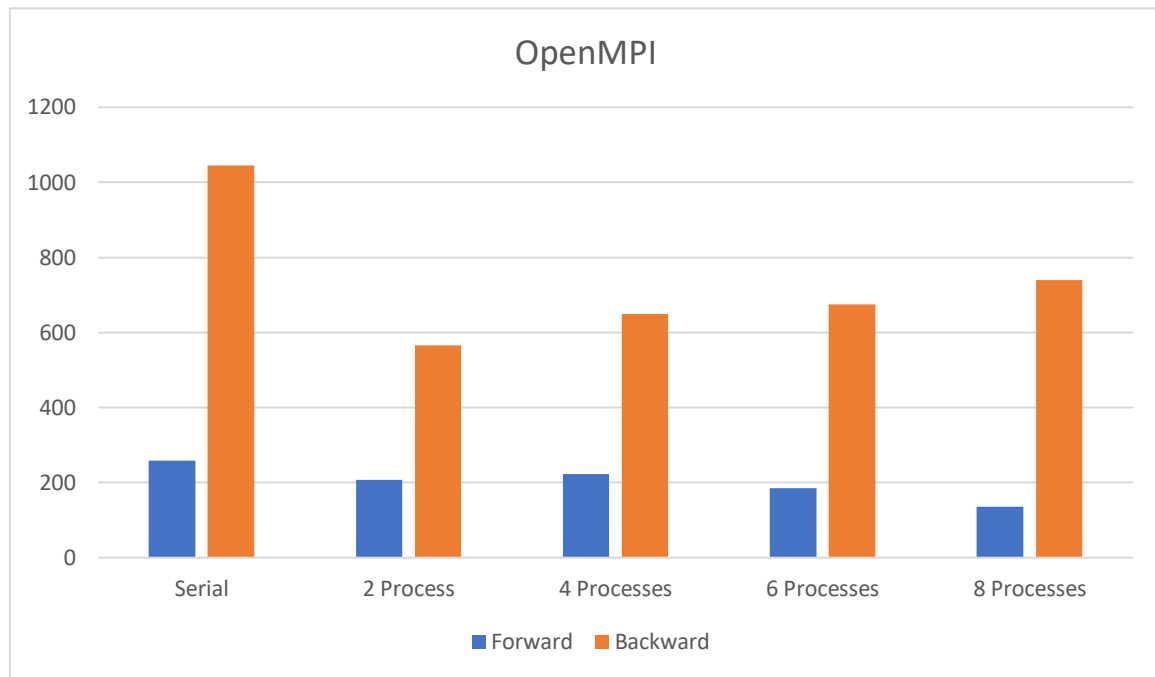


Figure 22 – Graph of Execution time vs Dictionary in OpenMPI

The MPI implementation was tested with various processes and the results were drawn. In Theory, MPI should produce faster results than serial and OpenMP. This is because of reduced memory usage by the CPU. The results of MPI implementation were faster than the serial and OpenMP version as expected. For the backward dictionary, execution time increased because of the limited number of cores available and repeated usage of the cores. For the forward dictionary, the execution time decreased significantly with increase in the number of processes.

## Performance Metrics:

### 1. Total Overhead:

It is the total time required to coordinate parallel tasks. The total overhead function is given by

$$T_o = pT_p - T_s$$

Where  $T_o$  is Total overhead function,  $T_p$  is the parallel time,  $T_s$  is the serial time and  $p$  is the number of processors.

The optimum number of threads/processors for the OpenMP/MPI implementation is 2 Threads/Processors. The Total Overhead time for these results is provided in the table below.

<i><b>Total Overhead Function</b></i>	<i><b>Forward</b></i>	<i><b>Backward</b></i>
<i><b>OpenMP</b></i>	240	367
<i><b>OpenMPI</b></i>	156.286	84.880

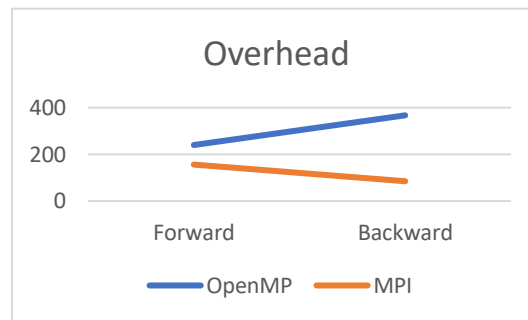


Figure 23 – Overhead function for optimum results

The OpenMP results has more overhead time than the MPI results. This can be because of waiting time of the threads for getting the resources or any background task causing the delay. This results in the Threads/processor take more to coordinate the parallel tasks.

## 2. Speedup:

It is the ratio of time taken for a sequential implementation to complete a task to that of the parallel implementation. Speedup measures the increase in time due to parallelism.

$$S = T_s/T_p$$

where  $T_p$  is the parallel execution time,  $T_s$  is the serial execution time.

<i><b>Speedup</b></i>	<i><b>Forward</b></i>	<i><b>Backward</b></i>
<i><b>OpenMP</b></i>	1.036	1.480
<i><b>OpenMPI</b></i>	1.245	1.849

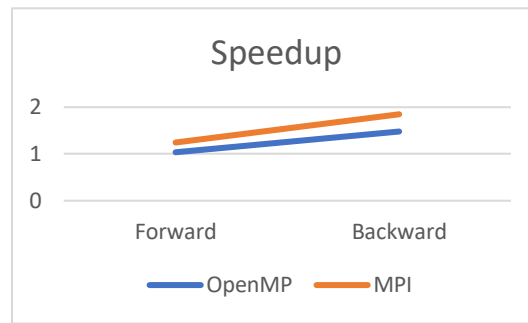


Figure 24 – Speedup for optimum results

The MPI implementation provide a better speedup compared to the OpenMP. As for the forward dictionary, both implementations produced somewhat similar speedup whereas the backward dictionary with MPI produced the highest speedup among the others.

### 3. Efficiency:

It is the ratio of speedup to the number of processors. It provides the information about how usefully the parallelization has been implemented.

$$E = S / p$$

where S is the Speedup and p is the number of processors. E ranges between 0 and 1.

<i>Efficiency</i>	<i>Forward</i>	<i>Backward</i>
<i>OpenMP</i>	0.518	0.740
<i>OpenMPI</i>	0.622	0.924

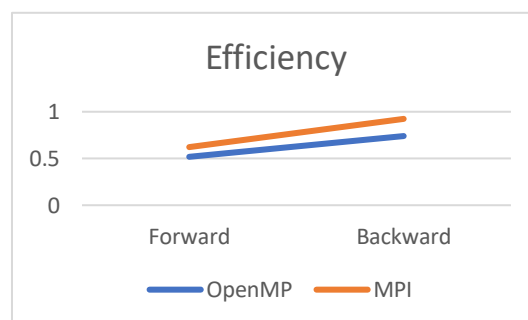


Figure 24 – Efficiency for optimum results

The backward dictionary is more efficient in both the implementation, where the MPI efficiency reached nearly ~1. The forward dictionary produced nearly 50% efficiency.

#### 4. Cost:

It is the product of time taken by parallel implementation to the number of processors. The parallel implementation is said to be effective if the cost of solving the task parallelly is less.

$$C = T_p * p$$

where,  $T_p$  is the parallel time and  $p$  is the number of processors.

<i>Cost</i>	<i>Forward</i>	<i>Backward</i>
<i>OpenMP</i>	498	1412
<i>OpenMPI</i>	414.286	1129.884

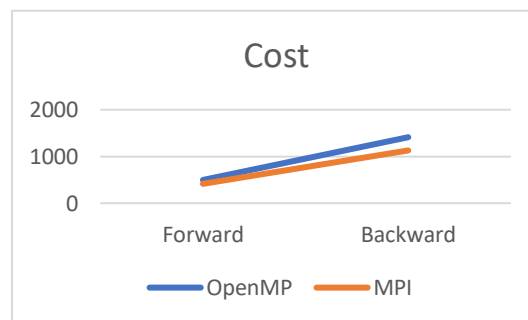


Figure 24 – Cost for optimum results

The OpenMP version requires more CPU usage which results in higher cost for both the dictionary.

#### Amdahl's Law and Gustafson Law:

Amdahl's law states that the speedup achieved by parallelisation can be expressed as

$$S = \frac{1}{(1 - f) + \frac{f}{N}}$$

where,  $f$  is the fraction of task which can be parallelized and  $N$  is number of processors.

This law deals with fixed computational problem size and fixed serial execution time. In our case, the algorithm uses sequential execution for decryption and comparison in the parallel region. Because of which the speedup will never be improved beyond the critical section execution time.

Whereas, Gustafson's law provides improvement in Amdahl's law, it states that the speedup can be improved beyond Amdahl's speedup when high performance hardware is used. The Speedup can be expressed as

$$S = (1 - f) + Nf$$

Thus, improving the hardware configuration provides more cores for execution which results in faster clock speeds and this will produce faster speedup for the parallel parts.

## CONCLUSION

The OpenMP and OpenMPI implementation used for parallelisation produced much faster results than the sequential program. Although, increasing the number of threads/processors doesn't improve the performance because of the hardware restrictions. OpenMPI with 2 process produced the fastest execution among the others. From the results, we can infer that more threads/processes does not really improve the performance rather it reduces it.

Thus, a more efficient OpenMP and MPI programs were created in this coursework, this can be implemented in various fields which requires high computational power and less execution time. According to Amdahl's law, more speedup is achieved for large size problems. This coursework can be further developed to implement more complex problems.

## REFERENCE

- Belhaous, S., Hidila, Z., Baroud, S., Chokri, S., & Mestari, M. (2020). An Execution Time Comparison of Parallel Computing Algorithms for Solving Heat Equation. *Communications in Computer and Information Science*, 1207 CCIS(January 2021), 283–295. [https://doi.org/10.1007/978-3-030-45183-7\\_22](https://doi.org/10.1007/978-3-030-45183-7_22)
- Chapman, B., Jost, G., & Pas, R. van der. (2008). *Using OpenMP*.
- Grama, A., Gupta, A., & Karypis, G. (2003). *Introduction to Parallel Computing*.