Distributed And Parallelized Image Encryption

Project Report

Course Code: CSE4001

(Parallel and Distributed Computing)

Slot: L1+L2 (Lab)

Professor: Dr. Murugan K

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

By,

Manjunath	**18BCE0039**
Pranjal Karn	**18BCE0134**
Mohd. Umar	**18BCE0196**
Anish Aggarwal	**18BCE0281**

Index

Topics **Pg. No.**

1. Abstract

When we think of necessity of parallel algorithms, one of the areas we can think of is cyber security. Data security is a major part of computational power consumption. Encryption is a way one can implement data security. Over the years many algorithms have been proposed. Many were cracked and easily solved by the brilliant of minds available, while some algorithms have still got hold of their ground and are still being implemented by big business houses in order to safeguard against unethical people. One of them being AES (Advanced Encryption Standard) encryption algorithm which was initially being brought as a replacement for DES (Data Encryption Standard) in 2001. As a result many companies started using this algorithm. Now, if you are aware of AES algorithm, then you might be knowing that AES consumes a lot of Processing power. AES takes a lot of time for larger files. On the other hand industries, and other Information Technology Companies are expanding its memory and data consumption day-by-day. Thus, sooner or later this algorithm will become obsolete for industrial usage. The fact that this algorithm cannot be decrypted easily and only known cases were due to some advanced brute-force mechanisms. Thus, this algorithm has still a lot of potential.

In the upcoming years, the industry is expecting tremendous growth in terms of processing capabilities. Multi-core CPUs have become common. Also, the front-runner in the field of processing capabilities is the Graphics Processing Unit (GPU).

In this paper, we focus on the application of general-purpose computation on GPU. We propose a new approach of fast parallel AES algorithm according to the architecture of GPU. We aim to improve the throughput and speed of AES encryption by optimizing the round function and large-scale parallel computing technology. According to our approach, we design a fast data encryption system based on parallel AES algorithm that can be implemented on GPU for further speedup. Our system can take full advantage of the high-performance computing capability of GPU and performing large-scale parallel computation of AES blocks, thus achieve fast AES encryption of information. Our system has the important significance in the practical application of computer information security and forensics.

2. Keywords:- Parallel, AES - encryption, Serial execution, Openmp, Symmetric Block cipher.

3. Introduction

Today, encryption of data has become an import part of data security. There are various encryption algorithms created like AES, DES, Bluefish, Triple DES algorithms,etc. In this project we aim towards creating a parallel version of the traditional AES - encryption algorithm and also test it on various performance measuring parameters.

The main reason behind choosing to parallelize the AES algorithm is because of the following reasons:

1. Robustness:

   The AES algorithm's security lies in its robustness under a brute force attack. This happens because the key space of AES-128 is 3.4 x 10^{38} keys in size. Even if it is continued to run forever, at 1 Tera keys/second, it would take around 1020 years to explore all of the key space.

2. Block cipher:

   The algorithm divides the plaintext into blocks of 16 bytes size and then perform the encryption processes. This can be thought of an opportunity of parallel handling of execution process.

3. Iterative nature:

   As discussed in the above 2 points, we know that the algorithm provides a near to perfect security, and it has its chances to be parallelized. The fact that the algorithm works in loops and that increases the memory consumption as well as the time taken for smaller sized data encryption (in comparison to other industrially available algorithms). It is sensible enough to enhance the performance of this algorithm as there is a lot of scope in its enhancement.

4. Literature Survey

4.1. Nishikawa N., Amano H., Iwai K. (2017) Implementation of Bitsliced AES Encryption on CUDA-Enabled GPU. In: Yan Z., Molva R., Mazurczyk W., Kantola R. (eds) Network and System Security. NSS 2017. Lecture Notes in Computer Science, vol 10394. Springer, Cham. https://doi.org/10.1007/978-3-319-64701-2_20

In this paper they have written about Table-based implementations which have been mainly reported in research related to high-performance AES on GPUs, in which tables are stored in the shared memory. Due to the increasing number of registers every year, GPU programming has enabled memory intensive applications such as bitsliced AES algorithm to be easily implemented. However, researches of implementation of bitsliced AES algorithm on GPU have not so far been conducted sufficiently in terms of several parameters. For this reason, in this paper, they have presented an implementation of bitsliced AES encryption on CUDA-enabled GPU with several parameters, especially focusing on three kinds of parallel processing granularities.

4.2. Shi J., Wang S., Sun L. (2020) A Parallel AES Encryption Algorithms and Its Application. In: Atiquzzaman M., Yen N., Xu Z. (eds) Big Data Analytics for Cyber-Physical System in Smart City. BDCPS 2019. Advances in Intelligent Systems and Computing, vol 1117. Springer, Singapore. https://doi.org/10.1007/978-981-15-2568-1_24

This paper presents a parallel AES encryption algorithm based on MapReduce architecture, which can be applied in large-scale cluster environment. It can improve the efficiency of massive data encryption and decryption by parallelization. And the paper designs a parallel cipher block chaining mode to apply AES algorithm. Experiments show that the proposed algorithm has good scalability and efficient performance, and can be applied to the security of massive data in cloud computing environment.

4.3. Sridevi Sathya Priya, S., KarthigaiKumar, P., Sivamangai, N.M. et al. High Throughput AES Algorithm Using Parallel Subbytes and MixColumn. Wireless Pers Commun 95, 1433–1449 (2017). https://doi.org/10.1007/s11277-016-3858-8

In this article, to achieve high speed in AES algorithm an eight stage Parallel accessing technique is used in SubByte transformation S-box and an eight stage parallel computation is applied in MixColumn transformation round. The results show that AES architecture with eight stage parallelism introduced in SubByte transformation and MixColumn transformation achieves high throughput than the other architectures. In S-box eight stage parallelism gives delay of 1.013 ns and in MixColumn it gives 0.835 ns delay. Parallel processing is used AES algorithm is used to increase the throughput with the trade off of increase in area. Using the proposed architecture 58.764 Gbps throughput is achieved with the expense of 6568 slices usage when implemented on virtex5 architecture which is recorded as a higher throughput than other architectures in the literature.

4.4. J. Ma, X. Chen, R. Xu and J. Shi, "Implementation and Evaluation of Different Parallel Designs of AES Using CUDA," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), Shenzhen, 2017, pp. 606-614, doi: 10.1109/DSC.2017.19.

This paper discusses how the performance of CBC-AES decryption based on GPU is influenced by 4 key parameters that include the size of input data, the number of threads per block, memory allocation style and parallel granularity. Further more, we compare the performance of AES on GPU to that of standard AES, AES-NI and find that when the size of input data is different, the implementations with different parameters setting achieve the best performance. So we provide several advices about how to implement CBC-AES on GPU aiming at different size of input data. In particular, our best performance of experiments on GPU(NVIDIA Tesla K40m) is about 112 times faster than the implementation of AES on CPU (Intel Xeon E5-2650) by using our optimization method.

5. H/W and S/W Requirements

5.1 Hardware

i. System type: x64-based
ii. Memory (RAM): Minimum 1 GB; Recommended 4 GB or above.
iii. Processor: Minimum 1 GHz; Recommended 2GHz or more.
iv. Hard Drive: Minimum 32 GB; Recommended 64 GB or more.
v. GPU (for better performance)

5.2 Software

i. GCC 9.0 (min 4.5)
ii. OpenMP
iii. Bash (for time analysis)

iv. Text Editor

v. Windows/ Linux OS

6. AES Algorithm

The core of AES algorithm is based upon the Rijndael cipher developed by Joan Daemen and Vincent Rijmen. The AES algorithm was announced as the new US government standard in November of 2001 and adopted as the standard for US encryption in May of 2002. On analyzing , we can see that AES is a symmetric key cryptography. It comprises of 3 block ciphers which are 128 bit, 192 bit, 256 bit. These are actually the block sizes. Each has different number of rounds associated. Number of rounds are the number of times the instructions processing is going to be done. Every round consists of various encryption steps including one which relies on the encryption key. For decryption, a set of reverse rounds are performed and the original plaintext is obtained.

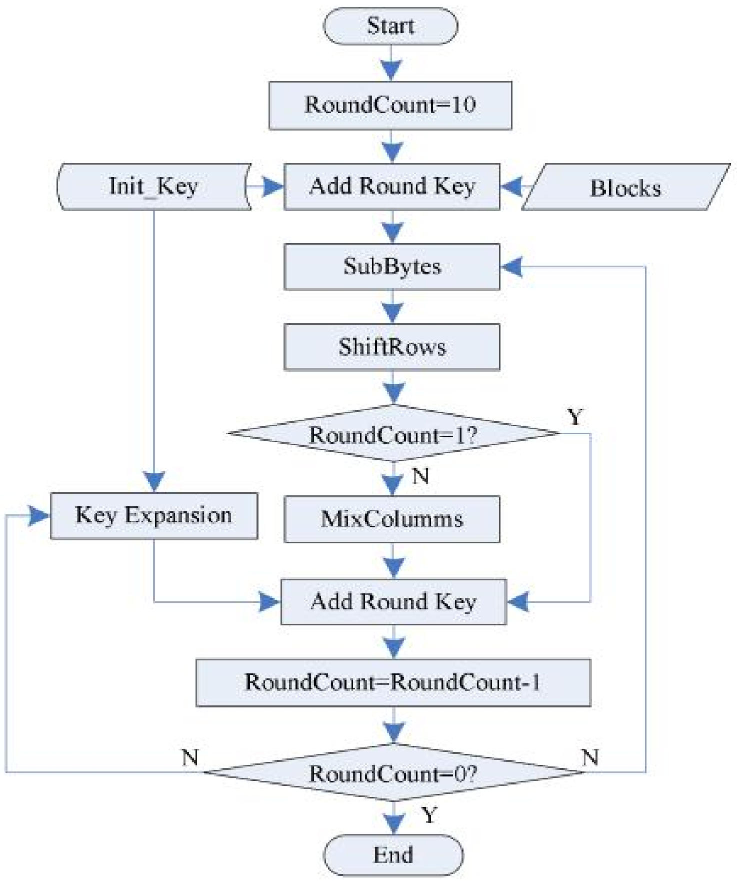


Figure 1: Algorithm Flowchart

In the above figure we can see that the AES encryption is an iterative algorithm and in case of 128-bit encryption the iteration goes up to 10 rounds. For each round the input is a block of data and the initial key. Each round has a sequence of transformations and the results of each are dependent on input from the previous rounds. The four transformations are SubBytes, ShiftRows, MixColumns and AddRound-Key. The result of any transformation of the intermediate processes is called a state. There is slightly difference in the last round. In the end we get a block of encrypted data.

1. *Key Expansion:* In this stage we enter the key of 128/ 192/ 256 bits. This key is then expanded for usage in the further stages in the algorithm. The size of the key is related to the number of rounds performed in the algorithm. For example:
   i. 128-bits :- 10 rounds
      Expanded key size :- 352 bits
   ii. 192-bits :- 12 round
      Expanded key size :- 624 bits
   iii. 256- bits :- 14 rounds
      Expanded key size :- 960 bits
2. *AddRoundKey:* In this stage, the message is combined with the state using the appropriate expanded key version.
3. *SubBytes:* Here, modification is done to the block using an 8-bit substitution, or S-box. This is a non-linear transformation used to help avoid attacks based on algebraic manipulation.
4. *ShiftRows:* Cyclic shifting of data bits by certain offset is done.
5. *MixColumns:* This stage takes the four bytes of each column and applies a linear transformation to the data. The column is multiplied by the coefficient polynomial $c(x) = 3x^3+x^2+x+2$ (modulo x^4+1). This step, in conjunction with the ShiftRows step, provides diffusion in the original message, spreading out any non-uniform patterns.

7. The Parallel AES Algorithm

To parallelize our AES algorithm we first create an Encrypt and Decrypt functions. These 2 functions will separately handle the encryption processes of a block. This block will be disjoint from any other block and its alteration should not affect any other block. The 2 function will take plaintext as well as the encrypted text respectively. Then we fetch the message which has been split according to the bits chosen, for our case lets choose 256- bits. Now blocks of size 256-bits are passed into the Cipher function (Encrypt / Decrypt). Now handling the function becomes independent of external factors. So, the function can now be run in parallel with other the same function working on various blocks of data simultaneously.

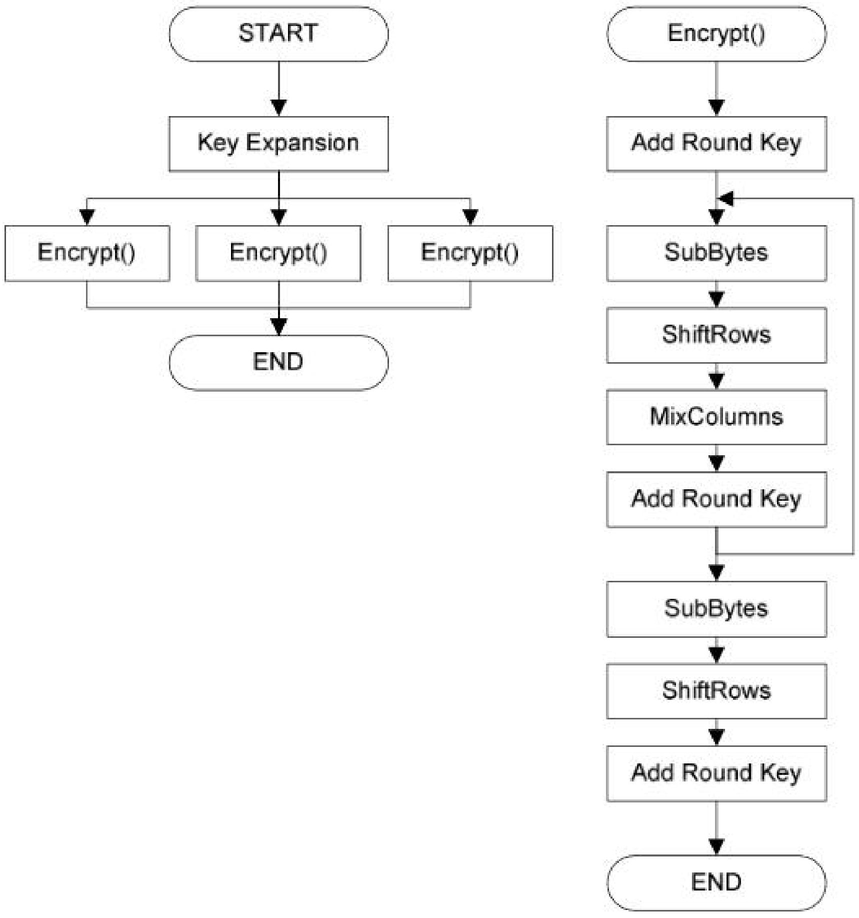


Figure 2: Parallel AES algorithm

The global memory access can optimized if we coalesce data accesses together to allow the system to perform wide memory reads.

This can be achieved by dividing the thread execution into 3 stages:

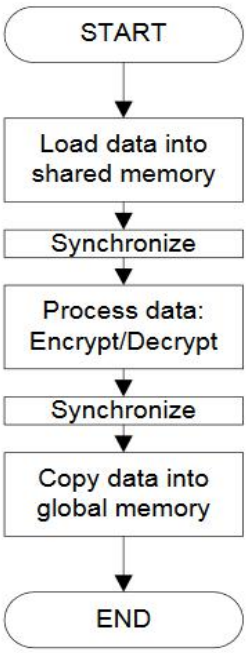

START
Load data into shared memory
Synchronize
Process data: Encrypt/Decrypt
Synchronize
Copy data into global memory
END

Figure 3: Stages of thread execution

Here, ordering of access is easy as every thread access the data from global memory before processing the data. Another optimization that can be done is creating a lookup table for faster access and remove that aspect of time consumption.

8. Experimental Analysis

8.1 Code

a) Encryption

```
#include<stdio.h>
#include<omp.h>
#include<string.h>

#define KB 1024
#define MB 1024*1024
#define CHUNK_SIZE 256*KB
#define KEY_LENGTH 128
// The number of columns comprising a state in AES. This is a constant in AES.
 Value=4
#define Nb 4

// The number of rounds in AES Cipher. It is simply initiated to zero. The act
ual value is recieved in the program.
int Nr=0;
```

```
// The number of 32 bit words in the key. It is simply initiated to zero. The
actual value is recieved in the program.
int Nk=0;

// The array that stores the round keys.
unsigned char RoundKey[240];

// The Key input to the AES Program
unsigned char Key[32];

int getSBoxValue(int num)
{
    int sbox[256] =   {
    //0     1     2      3     4     5     6     7      8     9     A      B     C
     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0x
fe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x
9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x
71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0x
eb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x
29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x
4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x
50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x
10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x
64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0x
de, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x
91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x
65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x
4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x
86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0x
ce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0x
b0, 0x54, 0xbb, 0x16 }; //F
    return sbox[num];
}

// The round constant word array, Rcon[i], contains the values given by
// x to th e power (i-1) being powers of x (x is denoted as {02}) in the field
 GF(28)
// Note that i starts at 1, not 0).
int Rcon[255] = {
    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0x
d8, 0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0x
ef, 0xc5, 0x91, 0x39,
    0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0x
cc, 0x83, 0x1d, 0x3a,
    0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x
1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0x
b3, 0x7d, 0xfa, 0xef,
    0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x
94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x
20, 0x40, 0x80, 0x1b,
    0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x
35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x
9f, 0x25, 0x4a, 0x94,
    0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x
04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x
63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0x
bd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x
8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x
2f, 0x5e, 0xbc, 0x63,
    0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x
72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x
74, 0xe8, 0xcb  };

// This function produces Nb(Nr+1) round keys. The round keys are used in each
 round to encrypt the states.
void KeyExpansion()
{
    int i,j;
    unsigned char temp[4],k;

    // The first round key is the key itself.
    for(i=0;i<Nk;i++)
    {
        RoundKey[i*4]=Key[i*4];
        RoundKey[i*4+1]=Key[i*4+1];
        RoundKey[i*4+2]=Key[i*4+2];
        RoundKey[i*4+3]=Key[i*4+3];
    }

    // All other round keys are found from the previous round keys.
    while (i < (Nb * (Nr+1)))
    {
        for(j=0;j<4;j++)
        {
            temp[j]=RoundKey[(i-1) * 4 + j];
        }
        if (i % Nk == 0)
        {
            // This function rotates the 4 bytes in a word to the left once.
            // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

            // Function RotWord()
            {
                k = temp[0];
                temp[0] = temp[1];
                temp[1] = temp[2];
                temp[2] = temp[3];
                temp[3] = k;
            }

            // SubWord() is a function that takes a four-byte input word and
            // applies the S-box to each of the four bytes to produce an outpu
t word.

            // Function Subword()
            {
                temp[0]=getSBoxValue(temp[0]);
                temp[1]=getSBoxValue(temp[1]);
                temp[2]=getSBoxValue(temp[2]);
                temp[3]=getSBoxValue(temp[3]);
            }

            temp[0] =  temp[0] ^ Rcon[i/Nk];
        }
        else if (Nk > 6 && i % Nk == 4)
        {
            // Function Subword()
            {
                temp[0]=getSBoxValue(temp[0]);
                temp[1]=getSBoxValue(temp[1]);
                temp[2]=getSBoxValue(temp[2]);
                temp[3]=getSBoxValue(temp[3]);
            }
        }
        RoundKey[i*4+0] = RoundKey[(i-Nk)*4+0] ^ temp[0];
        RoundKey[i*4+1] = RoundKey[(i-Nk)*4+1] ^ temp[1];
        RoundKey[i*4+2] = RoundKey[(i-Nk)*4+2] ^ temp[2];
        RoundKey[i*4+3] = RoundKey[(i-Nk)*4+3] ^ temp[3];
        i++;
    }
}

// This function adds the round key to state.
// The round key is added to the state by an XOR function.
void AddRoundKey(int round, unsigned char state[][4])
{
    int i,j;
    for(i=0;i<4;i++)
   {
        for(j=0;j<4;j++)
        {
            state[j][i] ^= RoundKey[round * Nb * 4 + i * Nb + j];
        }
    }
}

// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
void SubBytes(unsigned char state[][4])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSBoxValue(state[i][j]);
        }
    }
}

// The ShiftRows() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
void ShiftRows(unsigned char state[][4])
{
    unsigned char temp;
    // Rotate first row 1 columns to left
```

```
         temp=state[1][0];
         state[1][0]=state[1][1];
         state[1][1]=state[1][2];
         state[1][2]=state[1][3];
         state[1][3]=temp;
      // Rotate second row 2 columns to left

         temp=state[2][0];
         state[2][0]=state[2][2];
         state[2][2]=temp;

         temp=state[2][1];
         state[2][1]=state[2][3];
         state[2][3]=temp;
      // Rotate third row 3 columns to left

         temp=state[3][0];
         state[3][0]=state[3][3];
         state[3][3]=state[3][2];
         state[3][2]=state[3][1];
         state[3][1]=temp;
}

// xtime is a macro that finds the product of {02} and the argument to xtime m
odulo {1b}
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

// MixColumns function mixes the columns of the state matrix
void MixColumns(unsigned char state[][4])
{
    int i;
    unsigned char Tmp,Tm,t;
    for(i=0;i<4;i++)
    {
        t=state[0][i];
        Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i] ;
        Tm = state[0][i] ^ state[1][i] ; Tm = xtime(Tm); state[0][i] ^= Tm ^ T
mp ;
        Tm = state[1][i] ^ state[2][i] ; Tm = xtime(Tm); state[1][i] ^= Tm ^ T
mp ;
        Tm = state[2][i] ^ state[3][i] ; Tm = xtime(Tm); state[2][i] ^= Tm ^ T
mp ;
        Tm = state[3][i] ^ t ; Tm = xtime(Tm); state[3][i] ^= Tm ^ Tmp ;
    }
}

// in - is the array that holds the plain text to be encrypted.
// out -is the array that holds the key for encryption.
// state - the array that holds the intermediate results during encryption.
// Cipher is the main function that encrypts the PlainText.
void Cipher(unsigned char *in, unsigned char *out)
{
    int i,j,round=0;
   unsigned char state[4][4];
    //Copy the input PlainText to state array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++){

            state[j][i] = in[i*4 + j];
        }
    }

    // Add the First round key to the state before starting the rounds.
    AddRoundKey(0, state);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.
    for(round=1;round<Nr;round++)
    {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(round, state);
    }

    // The last round is given below.
    // The MixColumns function is not here in the last round.
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(Nr, state);

    // The encryption process is over.
    // Copy the state array to output array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            out[i*4+j]=state[j][i];
        }
    }
}
int main(int argc, char* argv[])
{
    int i;
    // Recieve the length of key here.
    while(Nr!=128 && Nr!=192 && Nr!=256)
    {
        // printf("Enter the length of Key(128, 192 or 256 only): ");
        scanf("%d",&Nr);
    }
   //Nr = KEY_LENGTH;

    // Calculate Nk and Nr from the recieved value.
    Nk = Nr / 32;
    Nr = Nk + 6;


    // The array temp_key stores the key.
    unsigned char temp_key[32] = {0x00  ,0x01  ,0x02  ,0x03  ,0x04  ,0x05  ,0x
06  ,0x07  ,0x08  ,0x09  ,0x0a  ,0x0b  ,0x0c  ,0x0d  ,0x0e  ,0x0f};

    // Copy the Key
    for(i=0;i<Nk*4;i++)
        Key[i]=temp_key[i];
```

```
//   ****************
// FILE READING CAPABILITY

    FILE *fp_input;
    FILE *fp_output;

    fp_input = fopen(argv[1], "rb");
    fp_output= fopen(argv[2], "ab");
   char chunk_input[CHUNK_SIZE];
   char chunk_output[CHUNK_SIZE];
   // The KeyExpansion routine must be called before encryption.
   KeyExpansion();
   while(!feof(fp_input)){
      //DO OPERATIONS
      fread(chunk_input, sizeof(chunk_input), 1, fp_input);
      #pragma omp parallel for private(i)
      for(i=0; i<CHUNK_SIZE; i += 16){
         unsigned char in[16], out[16];
         int block_no = i;
         memcpy(in, &chunk_input[block_no], 16);
         //The next function call encrypts the PlainText with the Key using AE
S algorithm.
         Cipher(in, out);
         memcpy(&chunk_output[block_no], out, 16);
      }

      fwrite(chunk_output, sizeof(chunk_output), 1, fp_output);
   }
   fclose(fp_output);
   fclose(fp_input);

   printf("Encryption of file %s complete. \n ", argv[1]);
}
```

b) Decryption

```
#include<stdio.h>
#include<omp.h>
#include<string.h>

#define KB 1024
#define MB 1024*1024
#define CHUNK_SIZE 256*KB
#define KEY_LENGTH 128

// The number of columns comprising a state in AES. This is a constant in AES.
 Value=4
#define Nb 4

// The number of rounds in AES Cipher. It is simply initiated to zero. The act
ual value is recieved in the program.
int Nr=0;

// The number of 32 bit words in the key. It is simply initiated to zero. The
actual value is recieved in the program.
int Nk=0;

// The array that stores the round keys.
unsigned char RoundKey[240];

// The Key input to the AES Program
unsigned char Key[32];

int getSBoxInvert(int num)
{
int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4,
 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42,
 0xfa, 0xc3, 0x4e
```

```
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d,
 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7,
 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8,
 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01,
 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0,
 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c,
 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa,
 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78,
 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27,
 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83,
 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
 0x21, 0x0c, 0x7d };

return rsbox[num];
}

int getSBoxValue(int num)
{
    int sbox[256] =   {
    //0     1    2      3     4    5     6     7      8    9     A      B    C
      D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0x
fe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x
9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x
71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0x
eb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x
29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x
4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x
50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x
10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x
64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0x
de, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x
91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x
65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x
4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x
86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0x
ce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0x
b0, 0x54, 0xbb, 0x16 };
    return sbox[num];
}

// The round constant word array, Rcon[i], contains the values given by
// x to th e power (i-1) being powers of x (x is denoted as {02}) in the field
 GF(2^8)
// Note that i starts at 1, not 0).
int Rcon[255] = {
    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0x
d8, 0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0x
ef, 0xc5, 0x91, 0x39,
    0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0x
cc, 0x83, 0x1d, 0x3a,
    0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x
1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0x
b3, 0x7d, 0xfa, 0xef,
    0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x
94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x
20, 0x40, 0x80, 0x1b,
    0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x
35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x
9f, 0x25, 0x4a, 0x94,
    0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x
04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x
63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0x
bd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x
8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x
2f, 0x5e, 0xbc, 0x63,
    0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x
72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x
74, 0xe8, 0xcb  };

// This function produces Nb(Nr+1) round keys. The round keys are used in each
 round to decrypt the states.
void KeyExpansion()
{
    int i,j;
    unsigned char temp[4],k;

    // The first round key is the key itself.
    for(i=0;i<Nk;i++)
    {
        RoundKey[i*4]=Key[i*4];
        RoundKey[i*4+1]=Key[i*4+1];
        RoundKey[i*4+2]=Key[i*4+2];
        RoundKey[i*4+3]=Key[i*4+3];
    }

    // All other round keys are found from the previous round keys.
    while (i < (Nb * (Nr+1)))
    {
        for(j=0;j<4;j++)
        {
            temp[j]=RoundKey[(i-1) * 4 + j];
        }
        if (i % Nk == 0)
        {
            // This function rotates the 4 bytes in a word to the left once.
            // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

            // Function RotWord()
            {
                k = temp[0];
                temp[0] = temp[1];
                temp[1] = temp[2];
                temp[2] = temp[3];
                temp[3] = k;
            }

            // SubWord() is a function that takes a four-byte input word and
            // applies the S-box to each of the four bytes to produce an outpu
t word.

            // Function Subword()
            {
                temp[0]=getSBoxValue(temp[0]);
                temp[1]=getSBoxValue(temp[1]);
                temp[2]=getSBoxValue(temp[2]);
                temp[3]=getSBoxValue(temp[3]);
            }

            temp[0] =  temp[0] ^ Rcon[i/Nk];
        }
        else if (Nk > 6 && i % Nk == 4)
        {
            // Function Subword()
            {
                temp[0]=getSBoxValue(temp[0]);
                temp[1]=getSBoxValue(temp[1]);
                temp[2]=getSBoxValue(temp[2]);
                temp[3]=getSBoxValue(temp[3]);
            }
        }
        RoundKey[i*4+0] = RoundKey[(i-Nk)*4+0] ^ temp[0];
        RoundKey[i*4+1] = RoundKey[(i-Nk)*4+1] ^ temp[1];
        RoundKey[i*4+2] = RoundKey[(i-Nk)*4+2] ^ temp[2];
        RoundKey[i*4+3] = RoundKey[(i-Nk)*4+3] ^ temp[3];
        i++;
    }
}

// This function adds the round key to state.
// The round key is added to the state by an XOR function.
void AddRoundKey(int round, unsigned char state[][4])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[j][i] ^= RoundKey[round * Nb * 4 + i * Nb + j];
        }
    }
}
```

```c
// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
void InvSubBytes(unsigned char state[][4])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSBoxInvert(state[i][j]);

        }
    }
}

// The ShiftRows() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
void InvShiftRows(unsigned char state[][4])
{
    unsigned char temp;

    // Rotate first row 1 columns to right
            temp=state[1][3];
            state[1][3]=state[1][2];
            state[1][2]=state[1][1];
            state[1][1]=state[1][0];
            state[1][0]=temp;
            // Rotate second row 2 columns to right

            temp=state[2][0];
            state[2][0]=state[2][2];
            state[2][2]=temp;

            temp=state[2][1];
            state[2][1]=state[2][3];
            state[2][3]=temp;
            // Rotate third row 3 columns to right

            temp=state[3][0];
            state[3][0]=state[3][1];
            state[3][1]=state[3][2];
            state[3][2]=state[3][3];
            state[3][3]=temp;
}

// xtime is a macro that finds the product of {02} and the argument to xtime m
odulo {1b}
```

```c
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

// Multiplty is a macro used to multiply numbers in the field GF(2^8)
#define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 & 1) * xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))

// MixColumns function mixes the columns of the state matrix.
// The method used to multiply may be difficult to understand for the inexperienced.
// Please use the references to gain more information.
void InvMixColumns(unsigned char state[][4])
{
    int i;
    unsigned char a,b,c,d;
    for(i=0;i<4;i++)
    {

        a = state[0][i];
        b = state[1][i];
        c = state[2][i];
        d = state[3][i];


        state[0][i] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d, 0x09);
        state[1][i] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
        state[2][i] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
        state[3][i] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d, 0x0e);
    }
}

// InvCipher is the main function that decrypts the CipherText.
void InvCipher(unsigned char *in, unsigned char *out)
{
    int i,j,round=0;
    unsigned char state[4][4];
    //Copy the input CipherText to state array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[j][i] = in[i*4 + j];
        }
    }
    // Add the First round key to the state before starting the rounds.
    AddRoundKey(Nr, state);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.
    for(round=Nr-1;round>0;round--)
    {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(round, state);
        InvMixColumns(state);
    }

    // The last round is given below.
    // The MixColumns function is not here in the last round.
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(0, state);

    // The decryption process is over.
    // Copy the state array to output array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            out[i*4+j]=state[j][i];
        }
    }
}
int main(int argc, char* argv[])
{
    int i;

    // Recieve the length of key here.
    while(Nr!=128 && Nr!=192 && Nr!=256)
    {
        // printf("Enter the length of Key(128, 192 or 256 only): ");
        scanf("%d",&Nr);
    }
    // Calculate Nk and Nr from the recieved value.
    Nk = Nr / 32;
    Nr = Nk + 6;


// Part 1 is for demonstrative purpose. The key and plaintext are given in the
 program itself.
```

```
//  Part 1: ********************************************************

    // The array temp stores the key.
    // The array temp2 stores the plaintext.
    unsigned char temp[32] = {0x00  ,0x01  ,0x02  ,0x03  ,0x04  ,0x05  ,0x06  ,
0x07  ,0x08  ,0x09  ,0x0a  ,0x0b  ,0x0c  ,0x0d  ,0x0e  ,0x0f};

    // Copy the Key and CipherText
    for(i=0;i<Nk*4;i++)
        Key[i]=temp[i];

//         *********************************************************

//FILE READ AND WRITE FOLLOWS

   FILE *fp_input;
   FILE *fp_output;
   fp_input = fopen(argv[1], "rb");
   fp_output = fopen(argv[2], "ab");
   unsigned char source[CHUNK_SIZE];
   char chunk_input[CHUNK_SIZE];
   char chunk_output[CHUNK_SIZE];

   //The Key-Expansion routine must be called before the decryption routine.
   KeyExpansion();

   while(!feof(fp_input)){
      //DO OPERATIONS
      fread(chunk_input, sizeof(chunk_input), 1, fp_input);
      int i = 0;
      #pragma omp parallel for private(i)
      for(i=0; i<CHUNK_SIZE; i += 16){
         unsigned char in[16], out[16];
         int block_no = i;
         memcpy(in, &chunk_input[block_no], 16);
         //The next function call decrypts the PlainText with the Key using AE
S algorithm.
         InvCipher(in, out);
         memcpy(&chunk_output[block_no], out, 16);
      }

      fwrite(chunk_output, sizeof(chunk_output), 1, fp_output);

   }
   fclose(fp_output);
   fclose(fp_input);
   printf("Decryption of file %s complete. \n ", argv[1]);
}
```

8.2. Screenshot

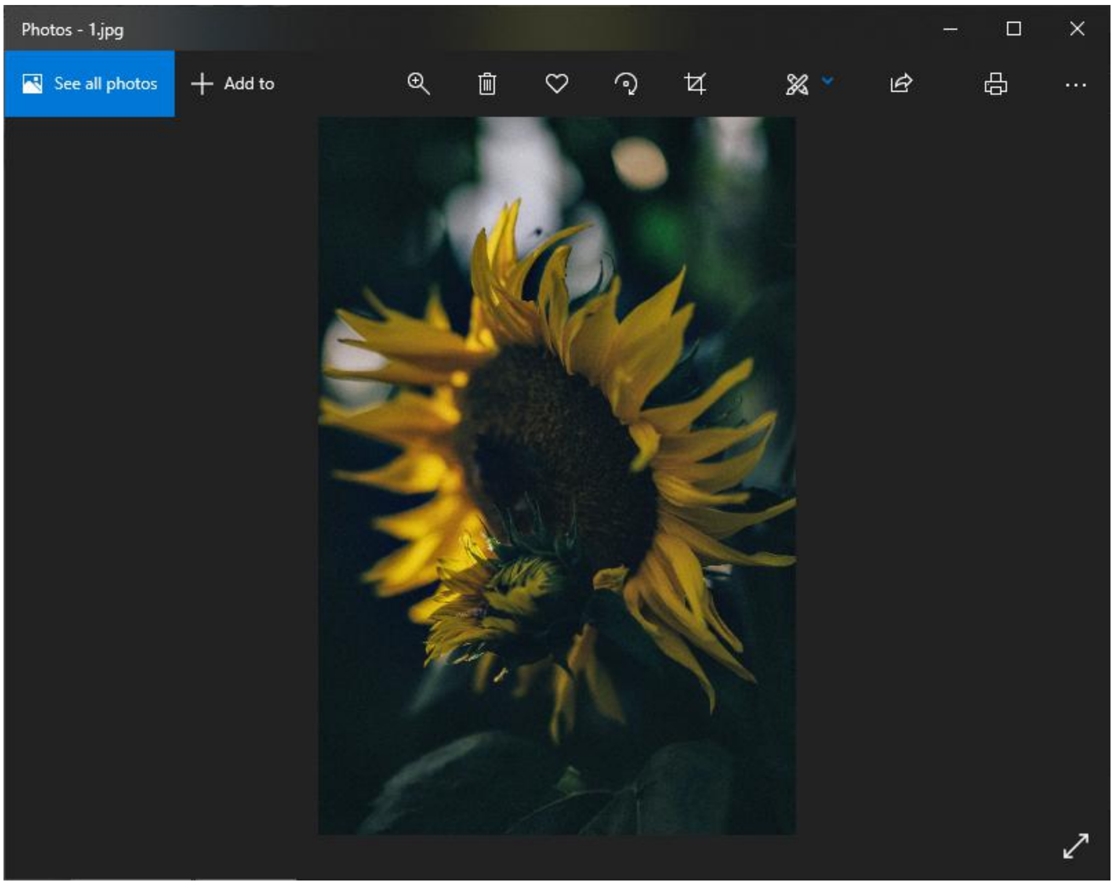
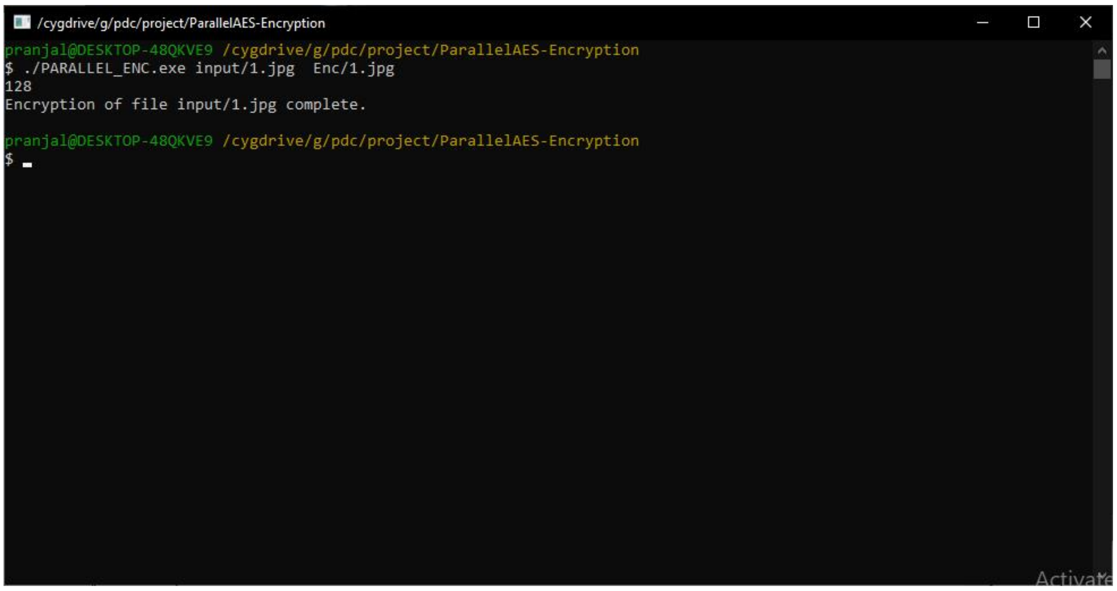


Figure 4: Input image file



```
/cygdrive/g/pdc/project/ParallelAES-Encryption
pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./PARALLEL_ENC.exe input/1.jpg  Enc/1.jpg
128
Encryption of file input/1.jpg complete.

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ 
```

Figure 5: Encryption done

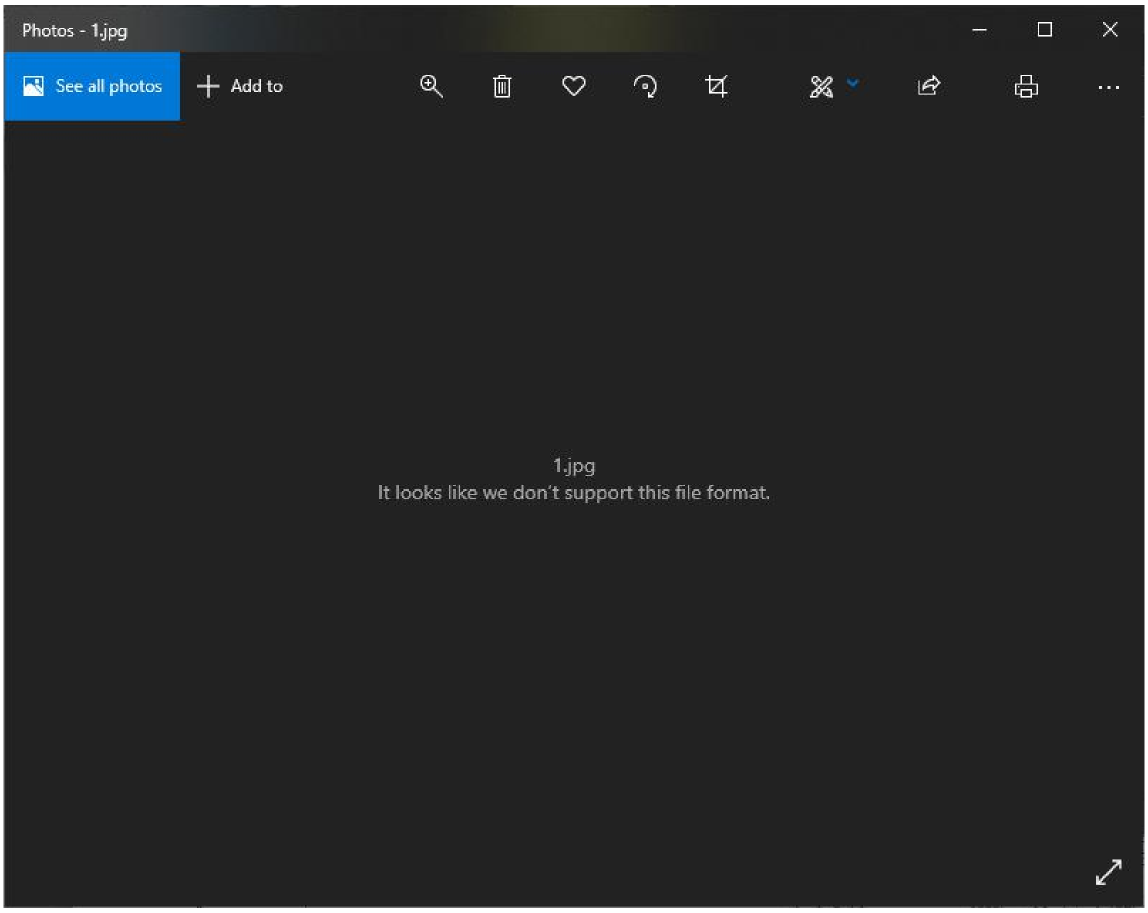
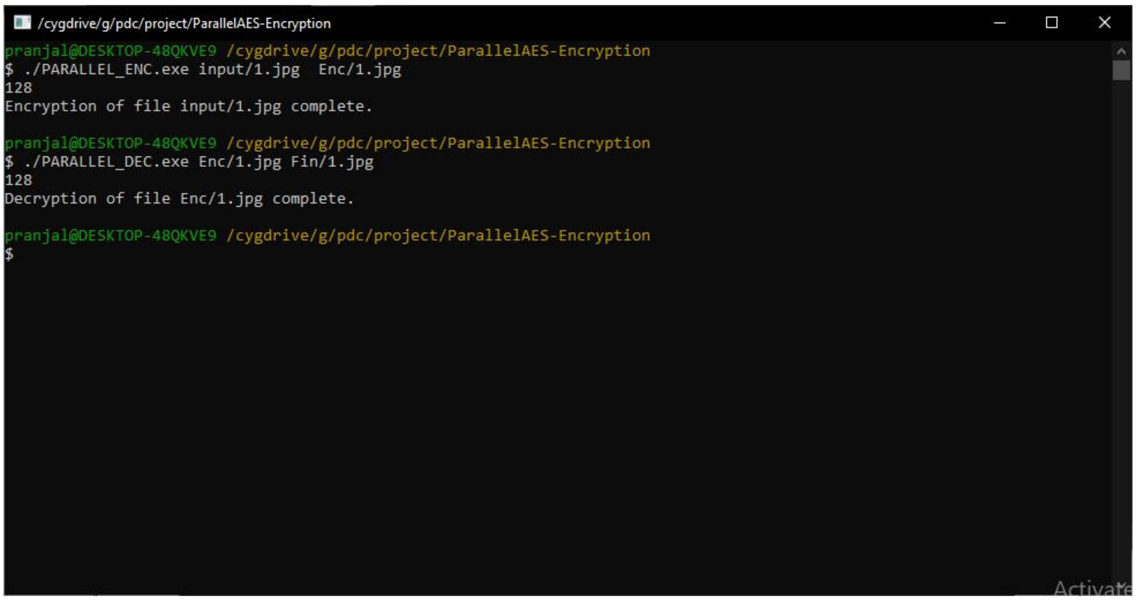


Figure 6: Encrypted file



```
pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./PARALLEL_ENC.exe input/1.jpg  Enc/1.jpg
128
Encryption of file input/1.jpg complete.

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./PARALLEL_DEC.exe Enc/1.jpg Fin/1.jpg
128
Decryption of file Enc/1.jpg complete.

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$
```

Figure 7: Decryption Done

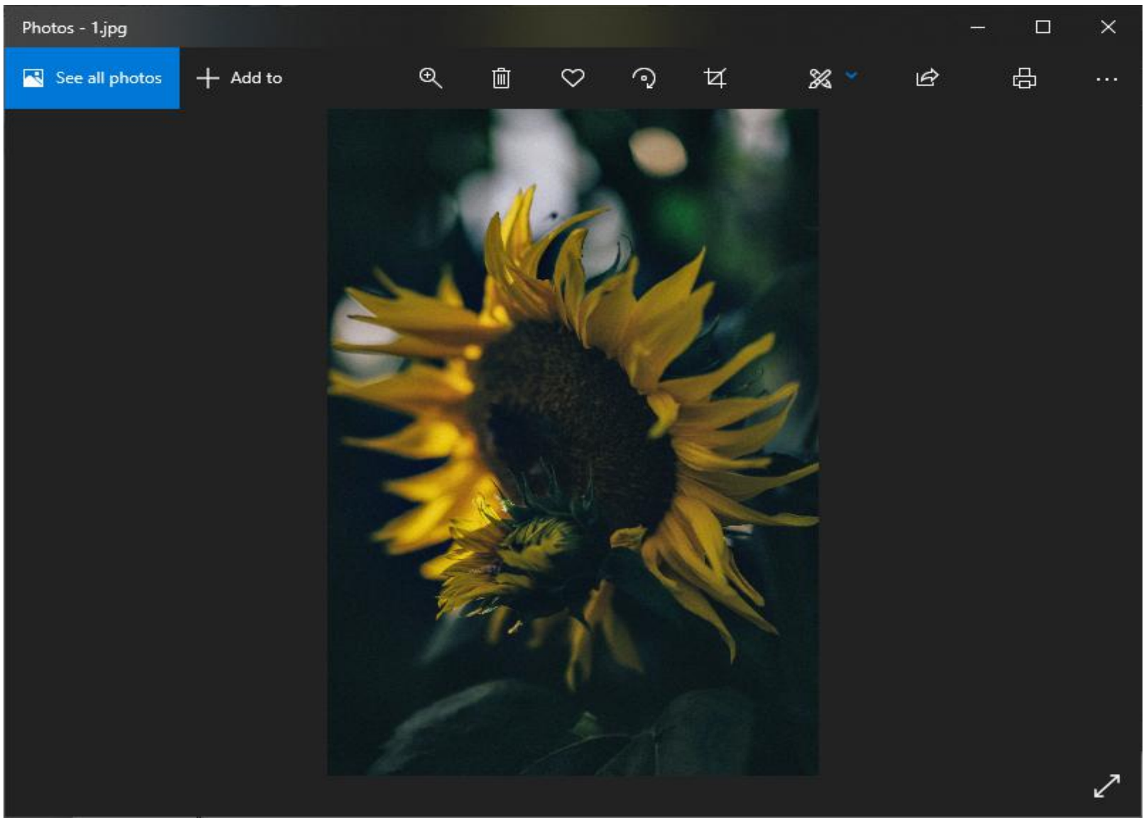


Figure 8: Decrypted File

9. Result and Analysis

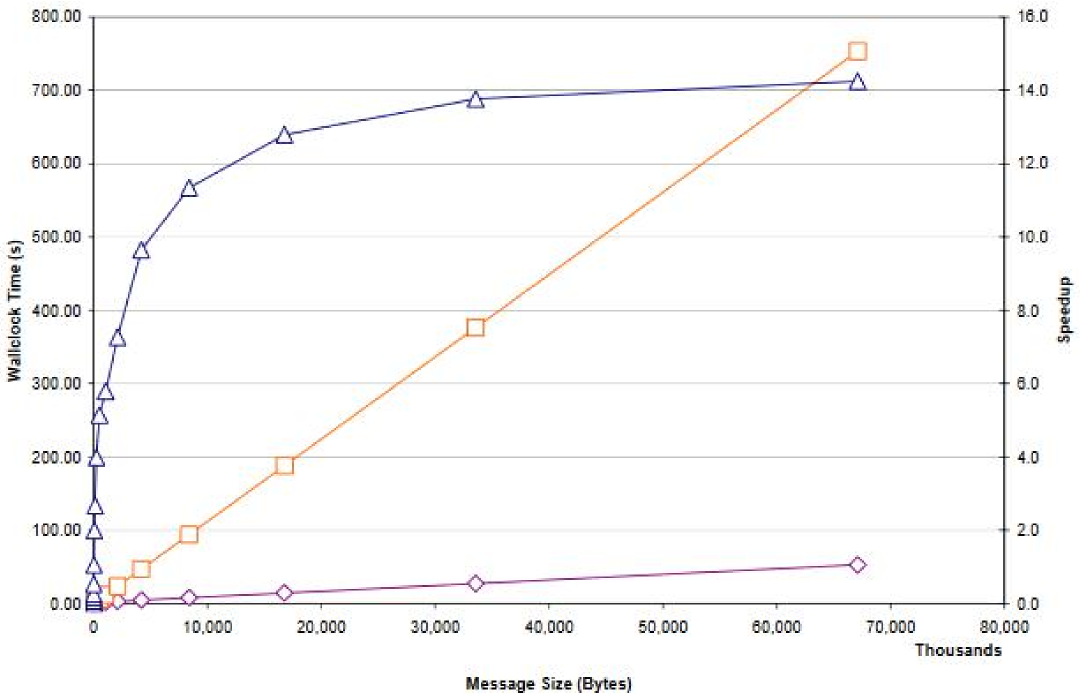


Figure 9: Performance Graph

Here,

Orange - Serial Execution
Blue - Speedup recorded
Purple - Parallel execution

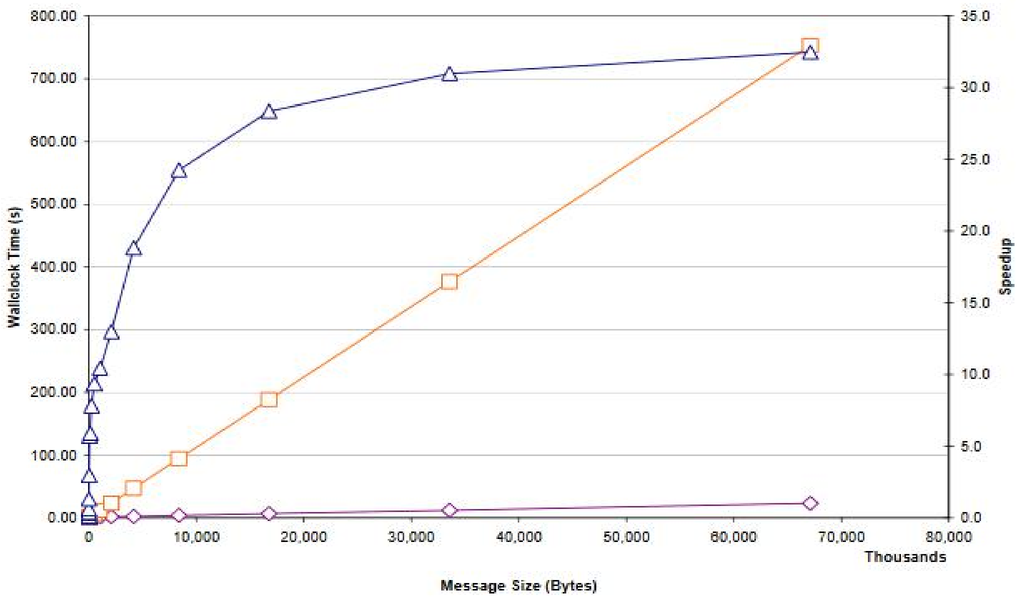


Figure 10: Without Memory Transfer and I/O time

From the above figures, we can say that the parallel version of the code gets massive speedups.

Screenshot for overall timings:

Let's think of industry application based scenario, like image encryption. Create a bash script of any script to run and calculate the algorithm run time and record it.

Parallel :-

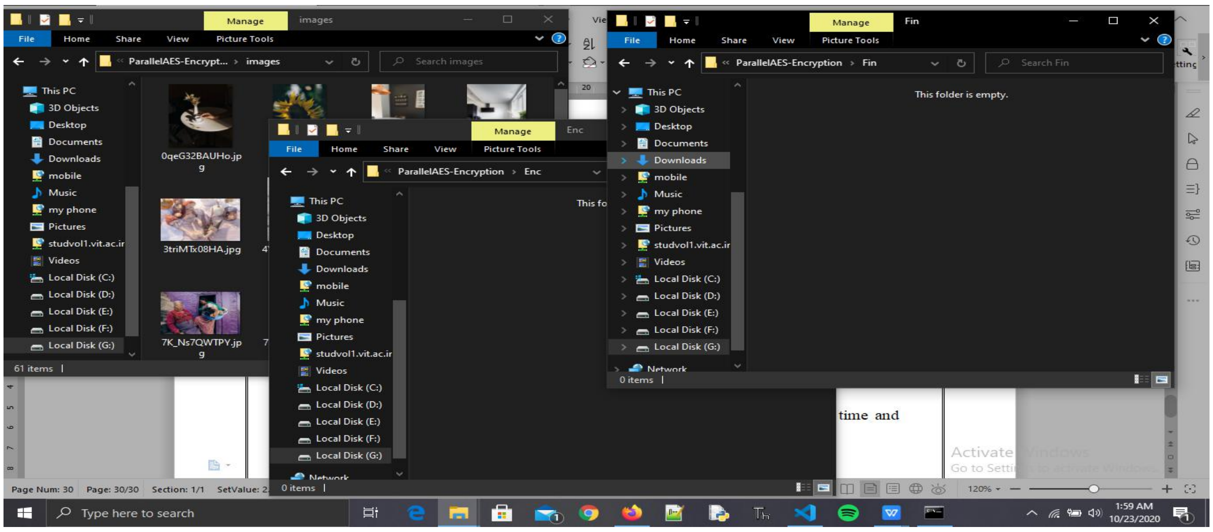
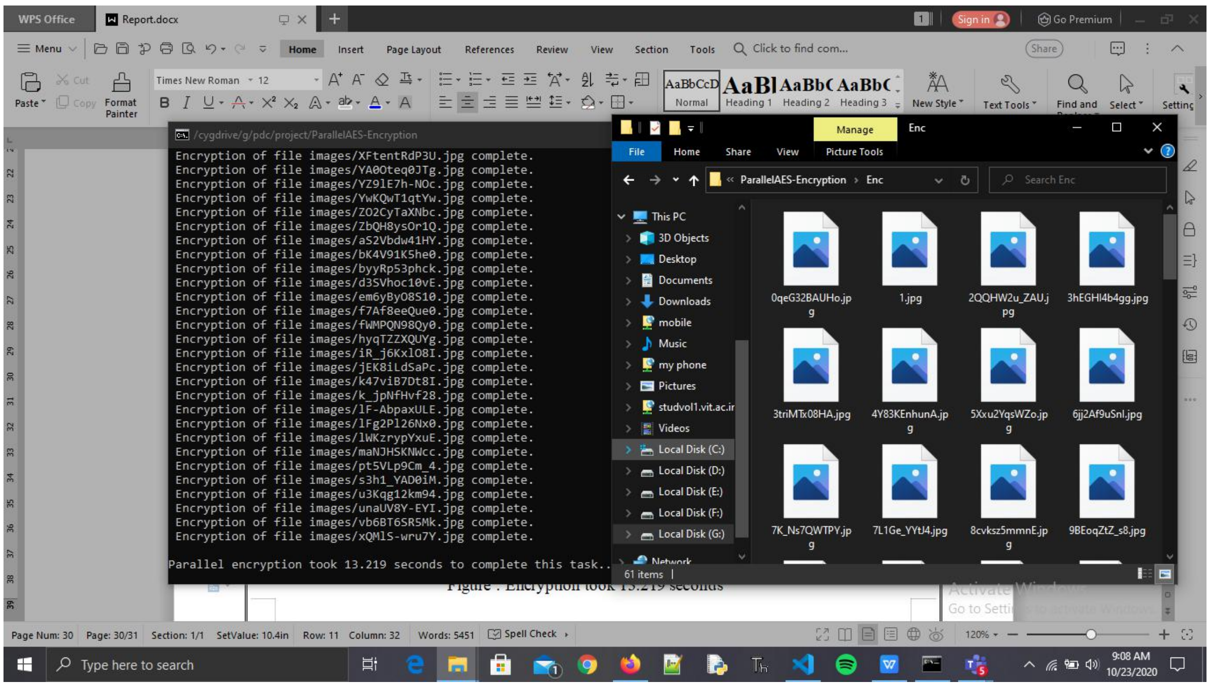
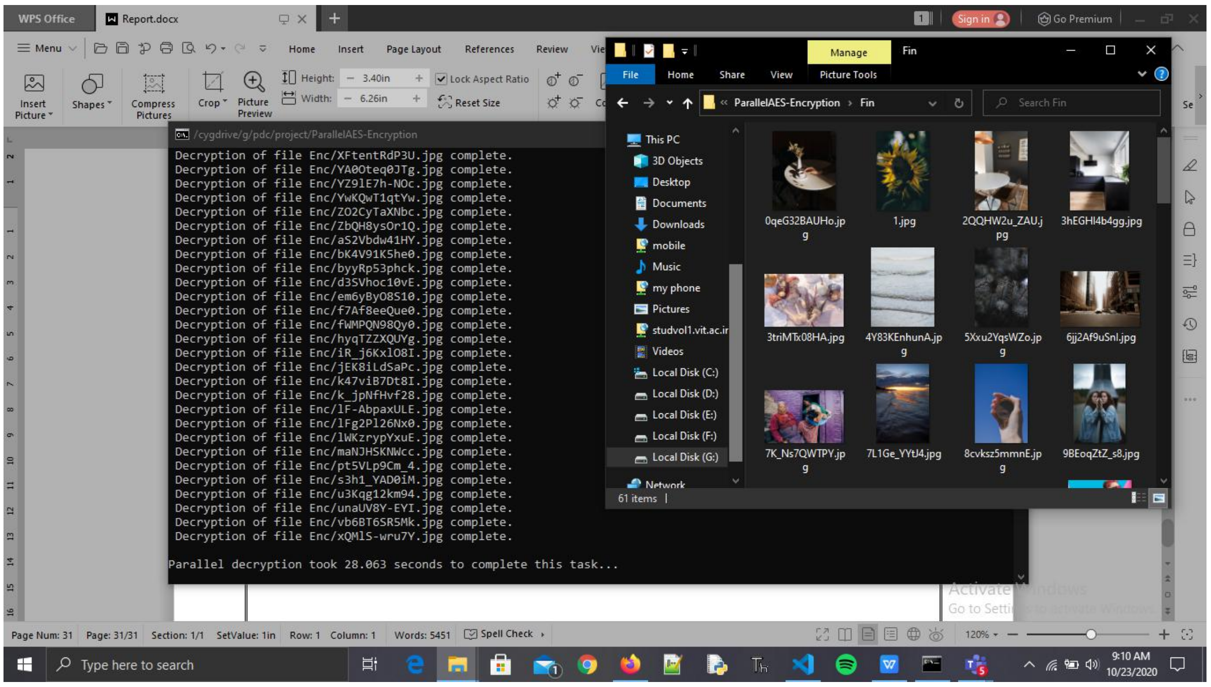


Figure 11: The encryption folders are initially empty



```
/cygdrive/g/pdc/project/ParallelAES-Encryption
Encryption of file images/XFtentRdP3U.jpg complete.
Encryption of file images/YA0Oteq0JTg.jpg complete.
Encryption of file images/YZ9lE7h-NOc.jpg complete.
Encryption of file images/YwKQwT1qtYw.jpg complete.
Encryption of file images/ZO2CyTaXNbc.jpg complete.
Encryption of file images/ZbQH8ysOr1Q.jpg complete.
Encryption of file images/aS2Vbdw41HY.jpg complete.
Encryption of file images/bK4V91K5he0.jpg complete.
Encryption of file images/byyRp53phck.jpg complete.
Encryption of file images/d3SVhoc10vE.jpg complete.
Encryption of file images/em6yByO8S10.jpg complete.
Encryption of file images/f7Af8eeQue0.jpg complete.
Encryption of file images/fWMPQN98Qy0.jpg complete.
Encryption of file images/hyqTZZXQUYg.jpg complete.
Encryption of file images/iR_j6KxlO8I.jpg complete.
Encryption of file images/jEK8iLdSaPc.jpg complete.
Encryption of file images/k47viB7Dt8I.jpg complete.
Encryption of file images/k_jpNfHvf28.jpg complete.
Encryption of file images/lF-AbpaxULE.jpg complete.
Encryption of file images/lFg2Pl26Nx0.jpg complete.
Encryption of file images/lWKzrypYxuE.jpg complete.
Encryption of file images/maNJHSKNWcc.jpg complete.
Encryption of file images/pt5VLp9Cm_4.jpg complete.
Encryption of file images/s3h1_YAD0iM.jpg complete.
Encryption of file images/u3Kqg12km94.jpg complete.
Encryption of file images/unaUV8Y-EYI.jpg complete.
Encryption of file images/vb6BT6SR5Mk.jpg complete.
Encryption of file images/xQMlS-wru7Y.jpg complete.

Parallel encryption took 13.219 seconds to complete this task..
```

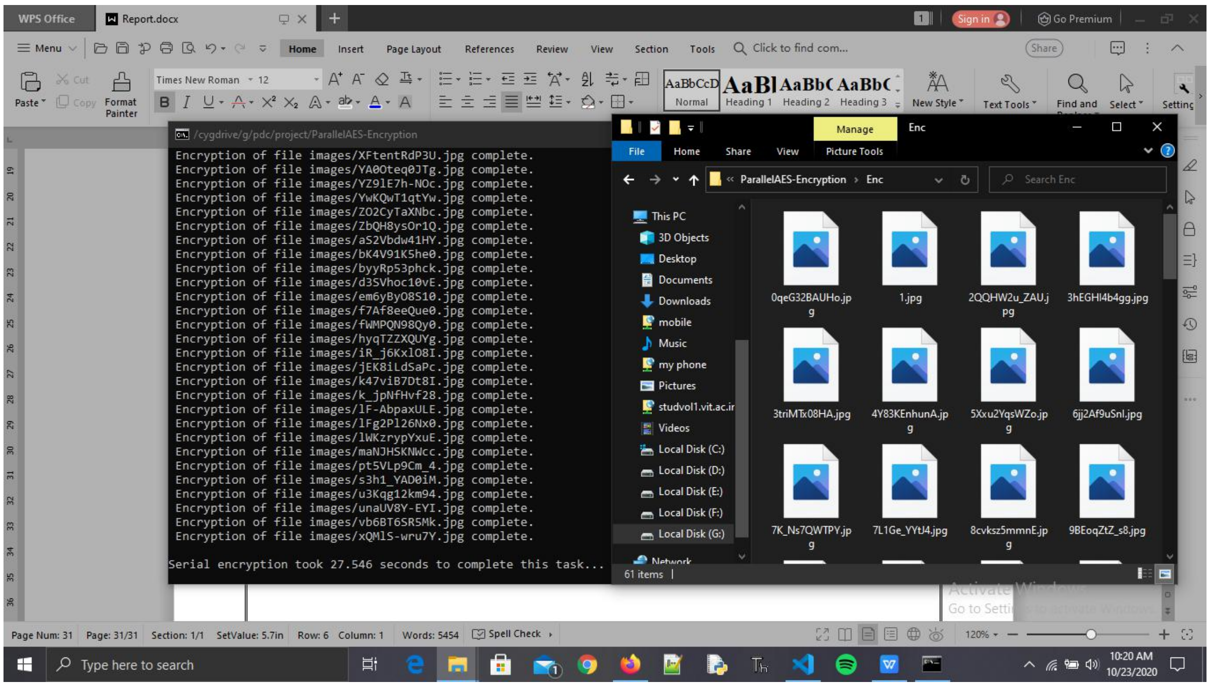
Figure 12: Parallel Encryption took 13.219 seconds



```
/cygdrive/g/pdc/project/ParallelAES-Encryption
Decryption of file Enc/XFtentRdP3U.jpg complete.
Decryption of file Enc/YA0Oteq0JTg.jpg complete.
Decryption of file Enc/YZ9lE7h-NOc.jpg complete.
Decryption of file Enc/YwKQwT1qtYw.jpg complete.
Decryption of file Enc/ZO2CyTaXNbc.jpg complete.
Decryption of file Enc/ZbQH8ysOr1Q.jpg complete.
Decryption of file Enc/aS2Vbdw41HY.jpg complete.
Decryption of file Enc/bK4V91K5he0.jpg complete.
Decryption of file Enc/byyRp53phck.jpg complete.
Decryption of file Enc/d3SVhoc10vE.jpg complete.
Decryption of file Enc/em6yByO8S10.jpg complete.
Decryption of file Enc/f7Af8eeQue0.jpg complete.
Decryption of file Enc/fWMPQN98Qy0.jpg complete.
Decryption of file Enc/hyqTZZXQUYg.jpg complete.
Decryption of file Enc/iR_j6KxlO8I.jpg complete.
Decryption of file Enc/jEK8iLdSaPc.jpg complete.
Decryption of file Enc/k47viB7Dt8I.jpg complete.
Decryption of file Enc/k_jpNfHvf28.jpg complete.
Decryption of file Enc/lF-AbpaxULE.jpg complete.
Decryption of file Enc/lFg2Pl26Nx0.jpg complete.
Decryption of file Enc/lWKzrypYxuE.jpg complete.
Decryption of file Enc/maNJHSKNWcc.jpg complete.
Decryption of file Enc/pt5VLp9Cm_4.jpg complete.
Decryption of file Enc/s3h1_YAD0iM.jpg complete.
Decryption of file Enc/u3Kqg12km94.jpg complete.
Decryption of file Enc/unaUV8Y-EYI.jpg complete.
Decryption of file Enc/vb6BT6SR5Mk.jpg complete.
Decryption of file Enc/xQMlS-wru7Y.jpg complete.

Parallel decryption took 28.063 seconds to complete this task...
```

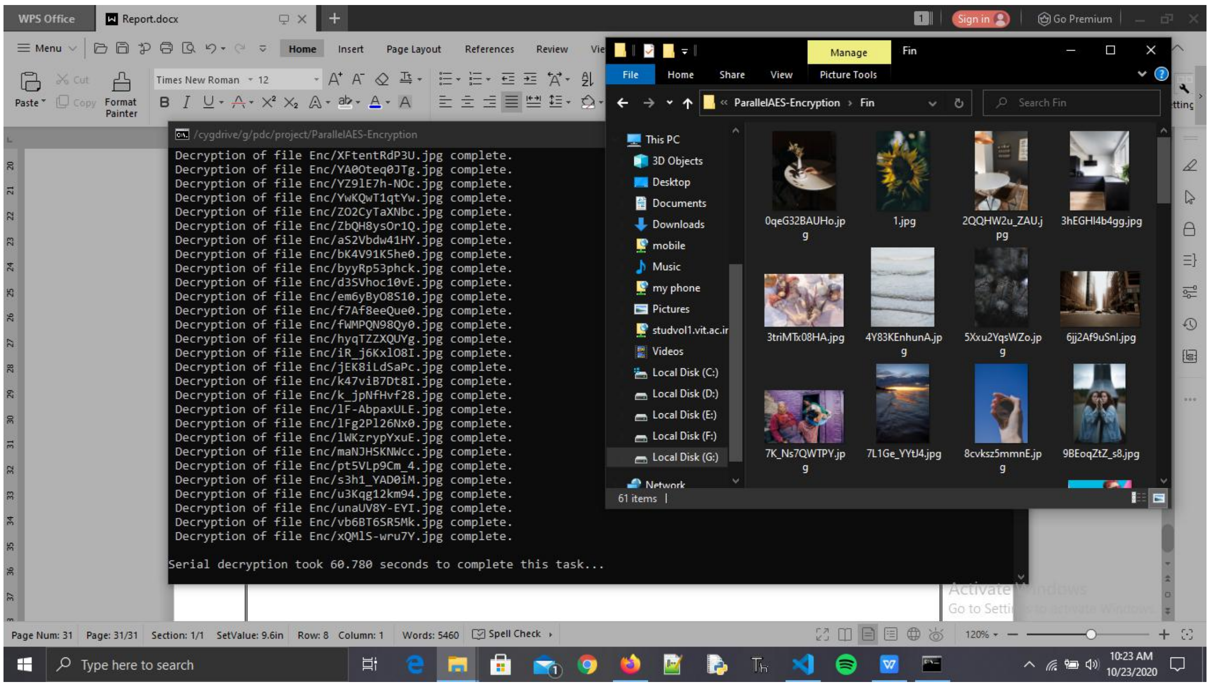
Figure 13: Parallel Decryption took 28.063 seconds

Serial:-



```
/cygdrive/g/pdc/project/ParallelAES-Encryption
Encryption of file images/XFtentRdP3U.jpg complete.
Encryption of file images/YA0Oteq0JTg.jpg complete.
Encryption of file images/YZ9lE7h-NOc.jpg complete.
Encryption of file images/YwKQwT1qtYw.jpg complete.
Encryption of file images/ZO2CyTaXNbc.jpg complete.
Encryption of file images/ZbQH8ysOr1Q.jpg complete.
Encryption of file images/aS2Vbdw41HY.jpg complete.
Encryption of file images/bK4V91K5he0.jpg complete.
Encryption of file images/byyRp53phck.jpg complete.
Encryption of file images/d3SVhoc10vE.jpg complete.
Encryption of file images/em6yByO8S10.jpg complete.
Encryption of file images/f7Af8eeQue0.jpg complete.
Encryption of file images/fWMPQN98Qy0.jpg complete.
Encryption of file images/hyqTZZXQUYg.jpg complete.
Encryption of file images/iR_j6KxlO8I.jpg complete.
Encryption of file images/jEK8iLdSaPc.jpg complete.
Encryption of file images/k47viB7Dt8I.jpg complete.
Encryption of file images/k_jpNfHvf28.jpg complete.
Encryption of file images/lF-AbpaxULE.jpg complete.
Encryption of file images/lFg2Pl26Nx0.jpg complete.
Encryption of file images/lWKzrypYxuE.jpg complete.
Encryption of file images/maNJHSKNWcc.jpg complete.
Encryption of file images/pt5VLp9Cm_4.jpg complete.
Encryption of file images/s3h1_YAD0iM.jpg complete.
Encryption of file images/u3Kqg12km94.jpg complete.
Encryption of file images/unaUV8Y-EYI.jpg complete.
Encryption of file images/vb6BT6SR5Mk.jpg complete.
Encryption of file images/xQMlS-wru7Y.jpg complete.

Serial encryption took 27.546 seconds to complete this task...
```

Figure 14: Serial Encryption took 27.546 seconds



```
/cygdrive/g/pdc/project/ParallelAES-Encryption
Decryption of file Enc/XFtentRdP3U.jpg complete.
Decryption of file Enc/YA0Oteq0JTg.jpg complete.
Decryption of file Enc/YZ9lE7h-NOc.jpg complete.
Decryption of file Enc/YwKQwT1qtYw.jpg complete.
Decryption of file Enc/ZO2CyTaXNbc.jpg complete.
Decryption of file Enc/ZbQH8ysOr1Q.jpg complete.
Decryption of file Enc/aS2Vbdw41HY.jpg complete.
Decryption of file Enc/bK4V91K5he0.jpg complete.
Decryption of file Enc/byyRp53phck.jpg complete.
Decryption of file Enc/d3SVhoc10vE.jpg complete.
Decryption of file Enc/em6yByO8S10.jpg complete.
Decryption of file Enc/f7Af8eeQue0.jpg complete.
Decryption of file Enc/fWMPQN98Qy0.jpg complete.
Decryption of file Enc/hyqTZZXQUYg.jpg complete.
Decryption of file Enc/iR_j6KxlO8I.jpg complete.
Decryption of file Enc/jEK8iLdSaPc.jpg complete.
Decryption of file Enc/k47viB7Dt8I.jpg complete.
Decryption of file Enc/k_jpNfHvf28.jpg complete.
Decryption of file Enc/lF-AbpaxULE.jpg complete.
Decryption of file Enc/lFg2Pl26Nx0.jpg complete.
Decryption of file Enc/lWKzrypYxuE.jpg complete.
Decryption of file Enc/maNJHSKNWcc.jpg complete.
Decryption of file Enc/pt5VLp9Cm_4.jpg complete.
Decryption of file Enc/s3h1_YAD0iM.jpg complete.
Decryption of file Enc/u3Kqg12km94.jpg complete.
Decryption of file Enc/unaUV8Y-EYI.jpg complete.
Decryption of file Enc/vb6BT6SR5Mk.jpg complete.
Decryption of file Enc/xQMlS-wru7Y.jpg complete.

Serial decryption took 60.780 seconds to complete this task...
```

Figure 15: Serial Decryption took 60.780 seconds

10. Extended Work

With slight modification we can even encrypt images in all formats, videos, pdf files and all files which can be viewed visually. This we implemented using the following Bash script

```
START=$(date +%s%3N) //time calculation begins
for i in input/*
do
echo <AES type (128/192/256)> | ./PARALLEL_ENC.exe $i "Enc/$(basename -- $i)"
done
END=$(date +%s%3N) //time calculation ends
DIFF=$(echo "scale=3;($END-$START)/1000" | bc)
echo -e "\nParallel <task> took $DIFF seconds to complete this task..."
```

Examples :

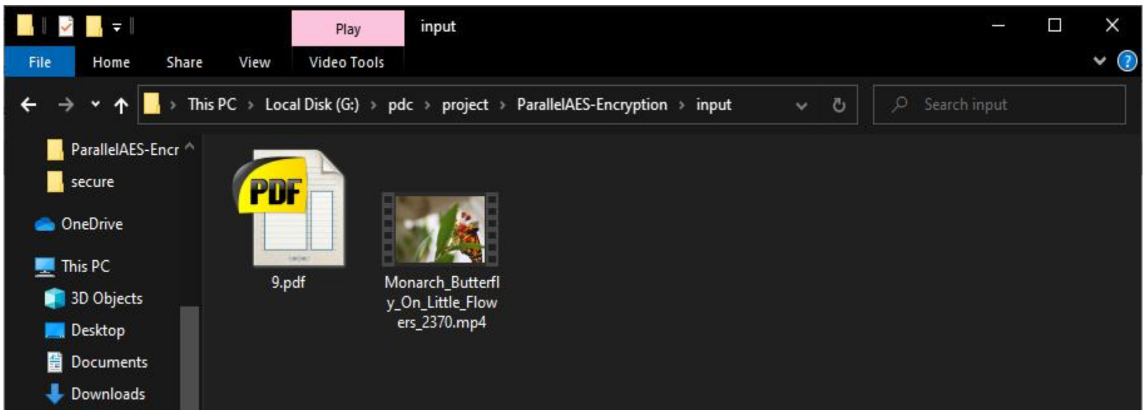


Figure 16: Input files (.pdf and .mp4)

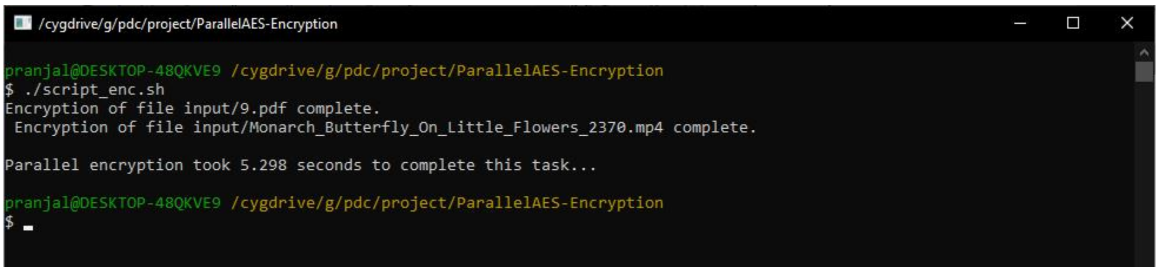
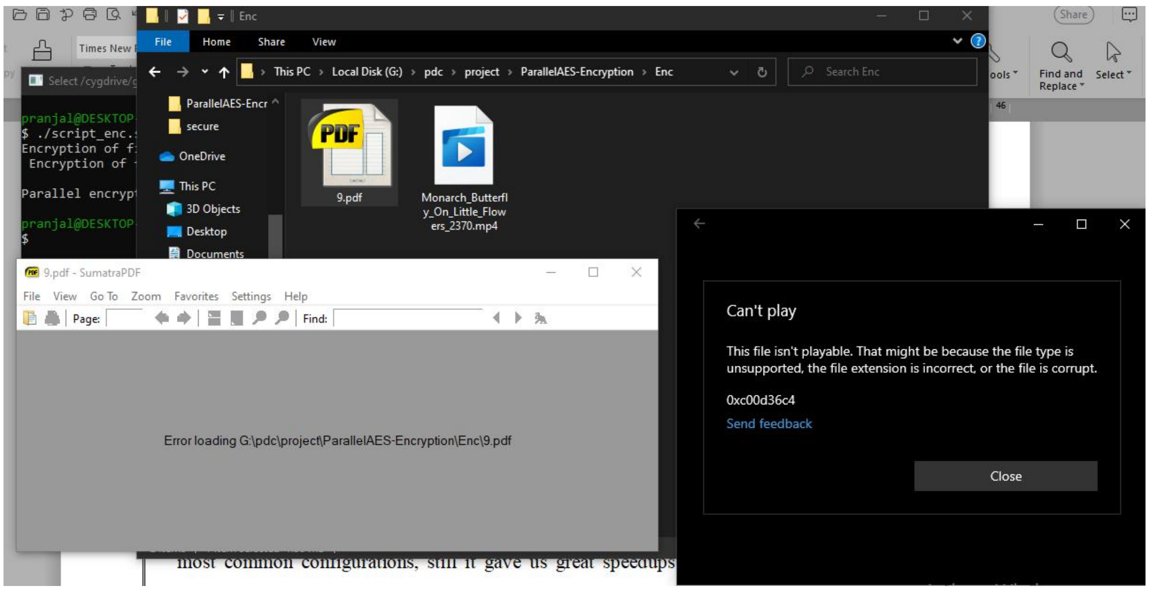

```
/cygdrive/g/pdc/project/ParallelAES-Encryption

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./script_enc.sh
Encryption of file input/9.pdf complete.
 Encryption of file input/Monarch_Butterfly_On_Little_Flowers_2370.mp4 complete.

Parallel encryption took 5.298 seconds to complete this task...

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$
```

Figure 17: Parallel encryption done



Error loading G:\pdc\project\ParallelAES-Encryption\Enc\9.pdf

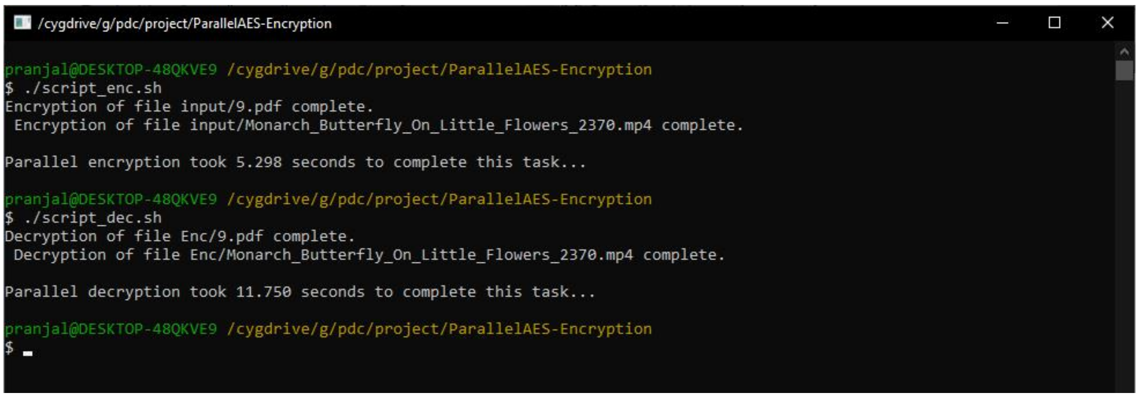
Can't play

This file isn't playable. That might be because the file type is unsupported, the file extension is incorrect, or the file is corrupt.

0xc00d36c4

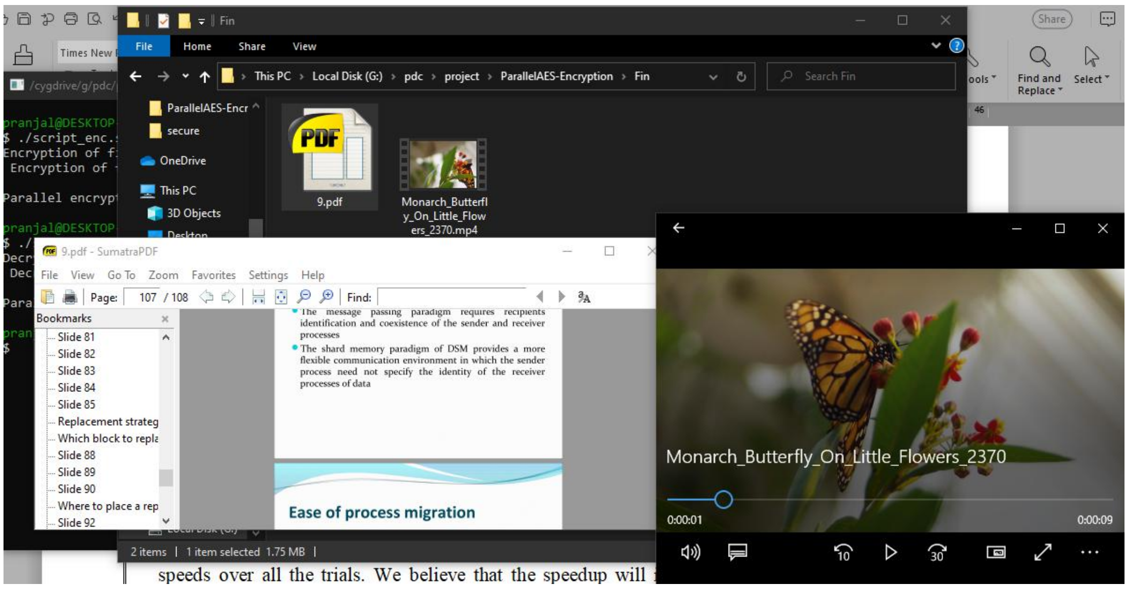
This PC > Local Disk (G:) > pdc > project > ParallelAES-Encryption > Enc
9.pdf
Monarch_Butterfly_On_Little_Flowers_2370.mp4
9.pdf - SumatraPDF
Send feedback
Close

Figure 18: Encrypted Files



```
pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./script_enc.sh
Encryption of file input/9.pdf complete.
 Encryption of file input/Monarch_Butterfly_On_Little_Flowers_2370.mp4 complete.

Parallel encryption took 5.298 seconds to complete this task...

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ ./script_dec.sh
Decryption of file Enc/9.pdf complete.
 Decryption of file Enc/Monarch_Butterfly_On_Little_Flowers_2370.mp4 complete.

Parallel decryption took 11.750 seconds to complete this task...

pranjal@DESKTOP-48QKVE9 /cygdrive/g/pdc/project/ParallelAES-Encryption
$ _
```

Figure 19: Decryption done



- The message passing paradigm requires recipients identification and coexistence of the sender and receiver processes
- The shard memory paradigm of DSM provides a more flexible communication environment in which the sender process need not specify the identity of the receiver processes of data

Ease of process migration

Figure 20: Decrypted file

11. Conclusion

For this experiment we conducted the experiments on our local systems having the most common configurations, still it gave us great speedups and more than double the speeds over all the trials. We believe that the speedup will increase even further if we perform the same on a GPU -enabled device. This way we successfully implemented and analyzed the parallel AES encryption algorithm.

12. References

[1] Nishikawa N., Amano H., Iwai K. (2017) Implementation of Bitsliced AES Encryption on CUDA-Enabled GPU. In: Yan Z., Molva R., Mazurczyk W., Kantola R. (eds) Network and System Security. NSS 2017. Lecture Notes in Computer Science, vol 10394. Springer, Cham. https://doi.org/10.1007/978-3-319-64701-2_20

[2] Shi J., Wang S., Sun L. (2020) A Parallel AES Encryption Algorithms and Its Application. In: Atiquzzaman M., Yen N., Xu Z. (eds) Big Data Analytics for Cyber-Physical System in Smart City. BDCPS 2019. Advances in Intelligent Systems and Computing, vol 1117. Springer, Singapore. https://doi.org/10.1007/978-981-15-2568-1_24

[3] Sridevi Sathya Priya, S., KarthigaiKumar, P., Sivamangai, N.M. et al. High Throughput AES Algorithm Using Parallel Subbytes and MixColumn. Wireless Pers Commun 95, 1433–1449 (2017). https://doi.org/10.1007/s11277-016-3858-8

[4] J. Ma, X. Chen, R. Xu and J. Shi, "Implementation and Evaluation of Different Parallel Designs of AES Using CUDA," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), Shenzhen, 2017, pp. 606-614, doi: 10.1109/DSC.2017.19.

[5] Advanced Encryption Standard, Wikipedia, http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[6] Block cipher modes of operation, Wikipedia, http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation