A PARALLEL IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD FOR HIGH PERFORMANCE COMPUTING PLATFORM USING MPJ EXPRESS

By

MEHREEN TAHIR

NUST201463280MRCMS64214F

Masters of Science in Systems Engineering

RESEARCH CENTER FOR MODELING AND SIMULATION (RCMS)

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

(NUST), ISLAMABAD, PAKISTAN

2017

A Parallel Implementation of Advanced Encryption Standard for High Performance Computing Platform Using MPJ Express

Supervised by

Dr. Muhammad Junaid Hussain

Research Center for Modeling and Simulation (RCMS)

A thesis submitted to the National University of Sciences and Technology in partial fulfillment of the requirement for the degree of Masters of Science

2017

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis written by Ms. **Mehreen Tahir**, Registration No. **NUST201463280MRCMS64214F** of **RCMS** has been vetted by undersigned, found complete in all aspects as per NUST Statutes/Regulations, is free of plagiarism, errors, and mistakes and is accepted as partial fulfillment for award of MS/MPhil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature with stamp: _____

Name of Supervisor: Dr. Muhammad Junaid Hussain

Date: _____

Signature of HoD with stamp: _____

Date: _____

Countersign by

Signature (Dean/Principal): _____

Date: _____

APPROVAL

It is certified that contents of the thesis entitled "**A Parallel Implementation of Advanced Encryption Standard for High Performance Computing Platform using MPJ Express**" submitted by Ms. **Mehreen Tahir**, Registration No. **NUST201463280MRCMS64214F** of **RCMS** have been found satisfactory as partial fulfillment for award of MS/MPhil degree.

Name of Supervisor: **Dr. Muhammad Junaid Hussain**

Signature:_____

Date: _____

Name of GEC member 1: **Ammar Mushtaq**

Signature: _____

Date: _____

Name of GEC member 2: **Fawad Khan**

Signature: _____

Date: _____

Name of GEC member 3: **Muhammad Tariq Saeed**

Signature: _____

Date: _____

Dedication

Dedicated to my beloved parents whose prayers and sacrifices made me what I am today. This achievement is a part of their dream to give me the best education as they could. And to my dear husband who supported and encouraged me at every step.

STATEMENT OF ORIGINALITY

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

Date

Mehreen Tahir

Acknowledgments

I fail to find enough words to acknowledge the greatest source of support and knowledge Allah Almighty; for giving me the knowledge, determination and health to achieve my goals, whose expert guidance and divine presence always strengthens me.

I would like to show my deepest gratitude to my supervisor Dr. Muhammad Junaid Hussain for his support, guidance and mentorship throughout this project.

I sincerely appreciate the efforts of my GEC members Engr. Fawad Khan and Dr. Ammar Mushtaq for giving their insightful input at every step. Specially Mr. Muhammad Tariq Saeed without his assistance and vision it was impossible for me to complete this project.

I am also thankful to Engr. Muhammad Hassan for his help and all those people who supported me indirectly during my project.

I am highly obliged to my all family members, without their unending support, tolerance and prayers the very idea of this study was impossible.

Contents

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HPC	High Performance Computing
MPICH	Message Passing Interface Chameleon
MPI	Message Passing Interface
MPJ	Message Passing Interface Java
NIST	National Institute of Standards and Technology
OpenMP	Open Multi-Processing

List of Tables

List of Figures

Abstract

The use of Java language for High Performance Computing (HPC) is becoming increasingly popular due to appealing language features and availability of parallel programming libraries and tools. In this work, we use MPJ (Message Passing Interface Java)-Express, a Java based library to accelerate Advanced Encryption Standard (AES) algorithm. MPJ-Express is an MPI (Message Passing Interface)-like implementation that supports acceleration of Java code on multicore and cluster computer systems. We have partitioned the problem at two levels. By employing a data parallel approach, we first divide the data length among available processors and then data at each processor is further divided among processor cores. The experimental results show almost linear throughput in case of multicore platform (1 node or stand alone system) and non linear throughput for cluster platform. These experimental results are compared with the AES algorithm accelerated by separately using other parallel programming tools in C language such as OpenMP API (Open Multi Processing Application Program Interface), MPICH (Message Passing Interface Chameleon) and CUDA (Compute Unified Device Architecture) programming model. Parallel AES implementation using MPJ Express provides high speed up factor and efficiency for multicore and cluster platform as compared to AES accelerated in C using OpenMP and MPICH. But the speed up of GPU based implementation of AES in C using CUDA (1 node) out performed AES in MPJ Express using multcore platform. Overall performance of AES accelerated in C on all platforms is best as compared to AES accelerated using MPJ Express. Accordingly it is concluded that this implementation is suitable for applications that are platform independent.

Chapter 1

Introduction

1.1 High Performance Computing

High Performance Computing or Supercomputing emerged in late 1970s and dramatically took over the whole world. Initially it was employed by defense and high tech industries but nowadays it is used in every type of industry. Supercomputing underwent substantial changes from technology point of view in 1990s. The most noticeable development was the architectural innovation in processor technology[23]. Many chip manufacturers began developing processors on single chip with power efficient computing units or cores. Multicore chips were produced to address the high power consumption issue of high clock speed systems that were unreliable. Cluster systems, built from multiple nodes (computers acting as servers) are extensively available and are now frequently used for high performance computing [64]. Software based advancements like Parallel Virtual Machine (PVM) also surfaced. PVM, a group of workstations and supercomputers is a single high performance parallel machine and network computing used to solve large scale problems. Of similar importance is a single portable message passing library having functions to be called from different programming languages (C, FORTRAN) to create parallel programs.[23]

Parallel programming is a significant attribute of high performance computing. Parallel processing is the practice that has been used for increasing the efficiency of processing large data. Conventionally, parallel processing signifies the idea of decreasing the execution time of a program by apportioning it into multiple portions, each of which is executed concurrently on its own processor. Theoretically, concurrent execution of these portions on n separate processors reduce the execution time by n times. The terms concurrent and parallel are used to refer to the condition where the period for executing two or more processes overlap in time, even if they start and stop at different times. It is possible to perform parallel processing by connecting multiple computers in a network and distributing fragments of the program to different computers on the network [78]. These high performance systems are very helpful in running large simulations as they lead to improved results. A suitable programming language is required to formulate algorithms for parallel programs. Execution of these programs is frequently managed through particular run time libraries or compiler directives which are included in a standard programming language like FORTRAN, Java or C [64].

1.2 Encryption and Decryption

Encryption is the process of transforming data into illegible format, also called as cipher, such that only authorized persons can read it. Decryption, on the other hand, is the process of reforming illegible data into its legible form, also called as decipher [58]. Encryption algorithms can be classified as:

1. Symmetric Algorithm

2. Asymmetric Algorithm

1.2.1 Symmetric Algorithm

Symmetric algorithm or Symmetric Key Cipher as the name indicates, uses identical keys for the encryption and decryption processes i.e. same key is used while transferring (send/receive) the data (if communication is done over the network) as shown in Figure 1.1 . The input data (plaintext) can be encrypted as a block (block ciphers) or a stream of bits (stream ciphers). Symmetric algorithms like Data Encryption Standard (DES), AES, blowfish, twofish are block ciphers. Whereas Rivest cipher 4 (RC4) is a stream cipher.

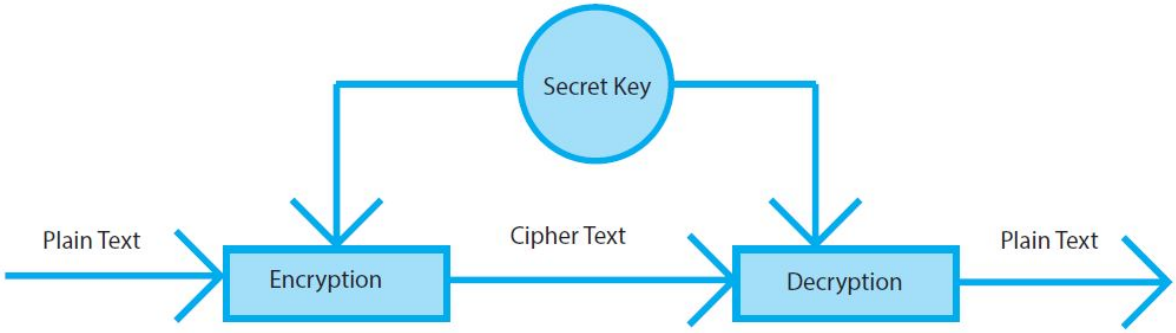

Figure 1.1 : Symmetric Key Cipher [57]
Symmetric Algorithm takes secret key and converts plain text into cipher text. Same key is used for encryption and decryption.

1.2.1.1 Data Encryption Standard (DES)

DES is derived from a symmetric block cipher called Lucifer and became the first Federal Information Processing Standard (FIPS) in 1976 to get rid of various attacks. It has a key length and block size of 56 bits and 64 bits respectively [41]. Even though it had the necessary security features, DES algorithm was unable to resist several attacks due to small key size i.e. 56 bits. Many variations of DES were produced to pawn these attacks but they did not provide a fully secure algorithm. Therefore, DES was superseded by Advanced Encryption Standard in 2001 and became new FIPS [57].

1.2.1.2 Advanced Encryption Standard (AES)

Unlike DES, AES has variable key lengths. Each key length has different number of encryption and decryption rounds. It has a block size of 128 bits. The whole process becomes more secure due to variable key size as it solves the key length issue in DES. The key sizes are: 1) 128-bits with 10 rounds. 2) 192-bits with 12 rounds. 3) 256-bits with 14 rounds. In 1998, a competition for replacing DES was organized by the National Institute of Standards and Technology (NIST) USA. In 2001, NIST announced that an algorithm called Rijndael [21] has been nominated as the new standard. AES is one of the most popular symmetric key encryption algorithms. It is frequently employed as a benchmark for encrypting/decrypting data throughout the world due to its security [30].

1.2.1.3 Implementation of Symmetric Algorithms

In networked applications that provide security, data encryption and decryption are frequent actions. But in order to match high data input rate of networked applications such as real time data processing or multimedia streaming very fast encryption and decryption schemes are required [74]. HPC is being used to increase the efficiency of processing large data at run time by executing symmetric algorithms in parallel.

1.2.2 Asymmetric Algorithm

Asymmetric/public key algorithm or Asymmetric Key Cipher is a technique which uses dissimilar keys for the encipher and decipher process as shown in Figure 1.2 . The major problem with symmetric key algorithm is the management of keys. Secure communication of key between sender and receiver requires a secure network. The risk increases with increase in number of users in the network. To solve this problem, asymmetric ciphers were developed. Two mathematically related different keys are used by sender and receiver. One is public key (can be

distributed publicly) and the other is private key. Examples of asymmetric key algorithms are Elliptic Curve Cryptography (ECC), Ron Rivest, Adi Shamir and Leonard Adleman (RSA), and Digital Signature Algorithm (DSA) etc.

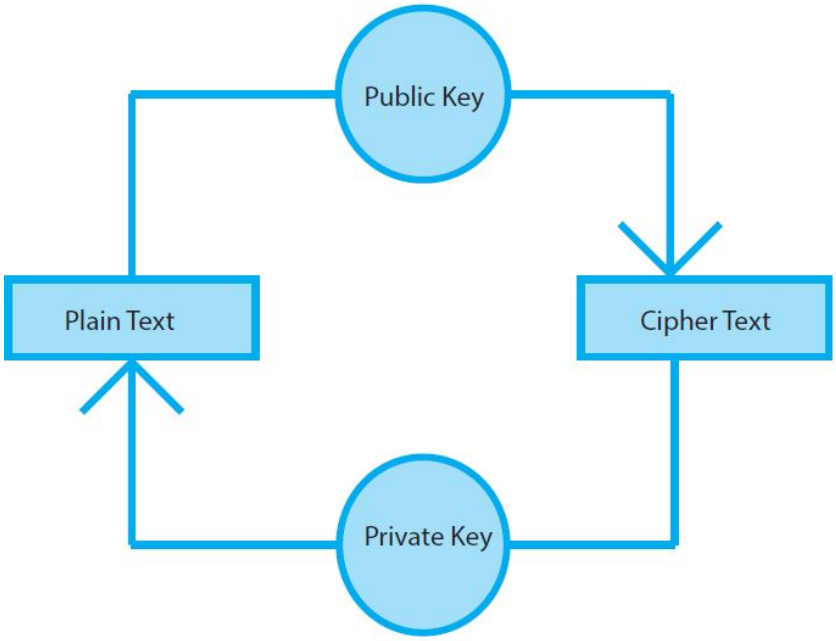

Figure 1.2 : Asymmetric Key Cipher [57]
Asymmetric Algorithm uses pubic key and converts plain text into cipher text. On the other hand, decryption requires private key to convert cipher text into plain text.

1.3 Research Objectives

The objectives of this research endeavor are to:

- Parallelizing AES using MPJ Express by utilizing multicore and cluster configuration on HPC platform in Java
- Parallelizing AES using OpenMP threads and MPICH on HPC platform in C
- Parallelizing AES by utilizing CUDA on GPUs
- Comparing the results of above mentioned implementations of AES

1.4 Methodology of Research

Figure 1.3 highlights the steps of research methodology of this thesis work. First step is to select domain of the research and formulate research theme which in our case lies between parallel computing and encryption. After this, literature is reviewed and objectives are defined. On the basis of literature and defined objectives a methodology is developed. Implementation of this methodology must satisfy the objectives. Results of this implementation are evaluated at the end and conclusion is drawn.

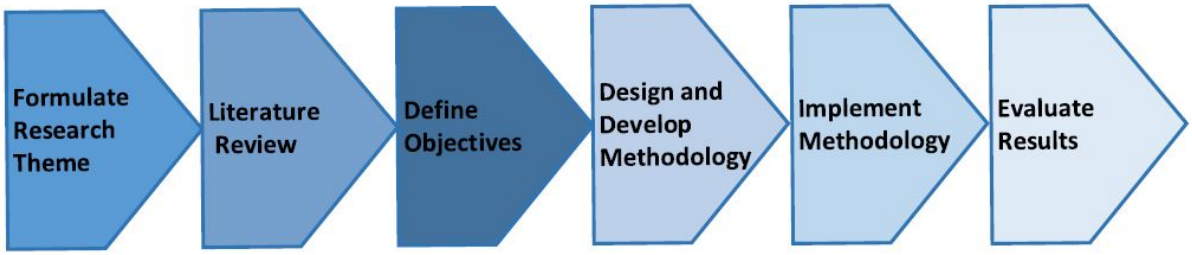

Figure 1.3 : Research Methodology
The research work carried out in current thesis is done by using this six step method. Domain of research is decided first and literature is reviewed accordingly. This literature review provides gap in literature. Research objectives are formulated to fill this gap and methodology is developed to achieve these objectives. Methodology is implemented and results are evaluated.

1.5 Contributions

Symmetric ciphers also known as bulk ciphers [5], are used to encrypt large amount of data. Their common use is found in the static storage of large amounts of sensitive data, as well as its communication over the network or across Internet. Bulk encryption offers safe and effective methods for protecting data from being compromised and stolen as they are encrypted with one symmetric key and cannot be decrypted with any other key. However, data encryption of bulk data requires very large response times.

HPC provides raw computational power to solve this problem. Parallel processing is the practice to utilize this computational power for increasing the efficiency of processing large data.

Parallel implementation of AES on HPC platform using MPJ Express provides a platform independent and scalable implementation that can be used at enterprise level to solve the problem of bulk encryption. Same implementation can be used on any HPC platform for bulk encryption purpose.

1.6 Organization of Thesis

A brief outline of the chapters included in this thesis is presented below:

Chapter 1 gives an introduction of HPC and Encryption. Objectives of the research along with the research methodology are explained afterwards. The chapter concludes with the contribution of this study in practical applications. Chapter 2 provides detailed explanation of AES algorithm starting from its inception. A detailed literature review is presented in Chapter 3 including both international as well as local scope of work. Some of the missing links are also identified. All the terms and concepts of HPC used in our methodology are discussed in Chapter 4. Last section of this chapter explains our proposed methodology. In Chapter 5, implementation of proposed methodology along with its comparison to AES parallelized in other platforms are presented. A comprehensive discussion of observed results along with potential reasoning is also stated. Chapter 6 is the Conclusion in which results of current study are presented. Recommendations for future research are also proposed in this chapter.

Chapter 2

Advanced Encryption Standard (AES)

2.1 Introduction

Two Belgian cryptographers, Joan Daemen and Vincent Rijmen, developed Advanced Encryption Standard Algorithm. This algorithm is frequently used as a benchmark for encrypting/decrypting data throughout the world due to its security [30]. AES is also required in numerous industrial standards and is utilized in several commercial applications and systems [21]. This chapter provides detailed description of Advanced Encryption Standard algorithm beginning from its formulation. Section 2.2 discusses the AES Selection procedure in detail. It consists of three rounds and results in Rijndael as final choice for AES. Section 2.3 highlights differences between AES and Rijndael. Section 2.4 is a high level view of AES. Section 2.5 comprises of detailed working of AES. Section 2.6 summarizes the chapter.

2.2 AES Selection Procedure

In September of 1997, the final invitations for nominee proposals of AES were issued. NIST stated that they are looking for an algorithm that must be more efficient and secure than 3DES. The author of selected algorithm must agree to make it open source. These were minimum

requirements for the cipher that must be met to qualify for the competition. Qualifying ciphers must have fixed 16 bytes (128 bits) block size but key length should be variable (128, 192 and 256 bits). The variable key length requirement was later dropped. But some candidates opted for the variable key length requirement in their design (e.g. RC6 and Rijndael) as NIST revealed that additional functionality would be received positively.

2.2.1 First Round

From all the submitted designs, 15 were selected to go into the first round. First round was a conference named as The First Advanced Encryption Standard Candidate Conference. It was organized on 20-22 August 1998 at Ventura, California where all accepted nominees were presented. All the designs were evaluated by international cryptographers.

Table 2.1: 15 Submissions for the First Round of AES Selection [61]

Submissions	**Submitter(s)**	**Submitter Type**
CAST-256	Entrust (CA)	Company
Crypton	Future Systems (KR)	Company
DEAL	Outerbridge, Knudsen (USA-DK)	Researchers
DFC	ENS-CNRS (FR)	Researchers
E2	NTT (JP)	Company
Frog	TecApro (CR)	Company
HPC	Schroeppel (USA)	Researcher
LOKI97	Brown et. Al (AU)	Researcher
Magenta	Deutsche Telekom (DE)	Company
Mars	IBM (USA)	Company
RC6	RSA (USA)	Company
Rijndael	Daemen and Rijmen (BE)	Researchers
SAFER+	Cylink (USA)	Company
Serpent	Anderson, Biham, Knudsen (UK-IL-DK)	Researchers
Twofish	Counterpane (USA)	Company

In the first round, all candidates were evaluated on the basis of cost, security and implementation characteristics. NIST invited cryptology experts to evaluate the candidates by mounting attacks and crypt analyse them or evaluate the implementation cost if anyone is interested. The first

round consisted of 2 conferences. Second AES Conference was joined with the yearly Fast Software Encryption Workshop. It was organized at Rome, Italy in March 1999. The workshop ended in August 1999. NIST selected the following 5 algorithms out of 15:

- RC6
- MARS
- Twofish
- Serpent
- Rijndael

2.2.2 Second Round

In April, 2000 at New York, 3rd AES conference was held. Fast Software Encryption Workshop was again joined with it. Three sessions were held:

1. Cryptographic attacks
2. Software implementation
3. Field Programmable Gate Arrays (FPGA) and Application-Specific Integrated Circuits (ASIC)

Attendants of conference were asked about their choice of AES through a questionnaire. Rijndael was unquestionably elected as the people's choice.

2.2.3 Final Round: Selection

NIST, on 2 October, 2000, formally announced Rijndael to be AES, without any modifications. NIST issued an excellent report of 116 pages in which they summarized all contributions. Motivations behind using Rijndael algorithm were

1. It performs really well in wide variety of both hardware and software environments
2. Very well suited for memory restricted environments
3. Internal structure of this algorithm makes it an excellent candidate for instruction level parallelism
4. Provide defense against power and timing attacks without impacting its own performance

2.3 Dissimilarities between Rijndael and the AES

The value for accepted block size and supported key length, is the only difference that separates Rijndael from AES.

Table 2.2: Differences between AES and Rijndael

Parameters	AES	Rijndael
Block Size	128 bits	Variable: Any multiple of 32, ranging from 128-256 bits
Key Length	128/192/256 bits	

The variable block size and key lengths are not a part of current FIPS standard as it was not assessed in AES selection process.

2.4 Overview of AES

Figure 2.1 shows the basic input output of AES. 128 bit plaintext x is the input of AES black box along with key k (use anyone of them 128/192/256 bits) and 128 bit cipher text y is the output. AES key length governs the number of round function of the algorithm.

DES does not encrypt complete plaintext block per iteration as it has feistal structure, unlike AES. Fesital networks are common in block ciphers but AES does not possess it, due to which it encrypts the entire block per iteration. Hence, it takes comparatively small number of rounds

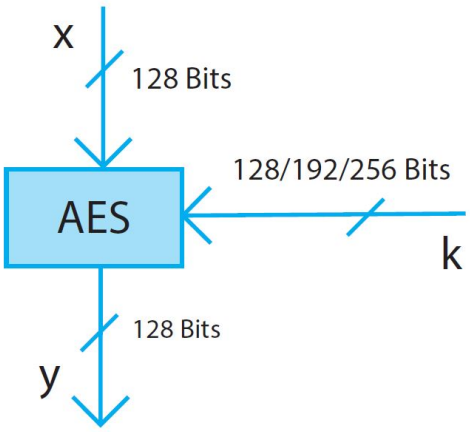

Figure 2.1 : AES Input/Output Parameters [61]
AES takes 128 bits of block and variable key length (128/192/256 Bits) as input. The output block size is 128 Bits.

Table 2.3: AES Key Lengths with Corresponding Number of Rounds [61]

Key Lengths	**# Rounds = nr**
128 bit	10
192 bit	12
256 bit	14

as compared to DES. Figure 2.2 shows block diagram of AES Encryption. Each round of AES comprises of three layers, except first. Furthermore, last round does not have mix column, which makes the encryption and decryption arrangement symmetrical. Three layers are as follows:

- **Key Addition layer**: A key scheduler is present that makes round keys/sub-keys out of main key. Each 128 bit sub key is then XORed (Exclusive OR) with the 128 bit of plaintext.

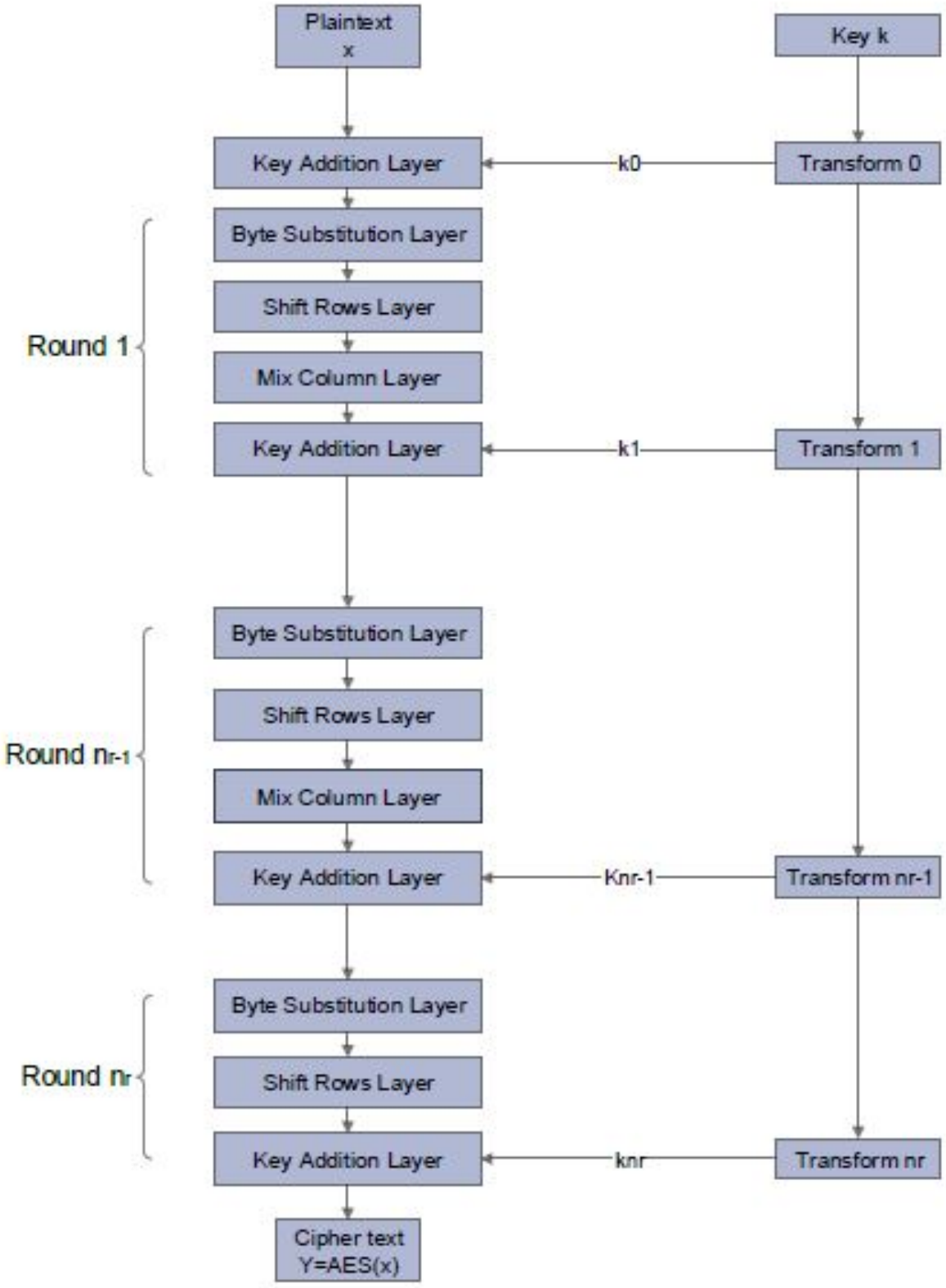

Figure 2.2 : AES Encryption Block Diagram [61]
Plaintext x is input and y ciphertext is output of AES. It has variable rounds depending upon the key length used. Each round has three layers. Last round misses mix column. Sub keys are generated from key k.

- **Byte Substitution layer (S-Box)**: An S-Box (Substitution Box) is formed with special mathematical properties that non linearly transforms data which introduces confusion in it.

- **Diffusion layer**: This layer consists of 2 sub layers. Each layer performs different linear operation.

  - *Shift Rows layer*: byte level permutation of the data
  - *Mix Column layer*: matrix operation that mixes 4x4 matrix of input with another 4x4 matrix

We will not discuss Key Scheduler as it is not in the scope of this thesis.

2.5 Working of AES

Figure 2.3 is interior structure of AES which shows that A0,...,A15 is a 16 byte input to S-Box. B0,...,B15 , the 16 byte output of S-Box is fed into the Shift Row layer that permutes it and then Mix Column layer mixes it to form 16 byte C0,...,C15 intermediate result. Key Addition layer XORs 128 bit round key to this result to complete one round of AES.

One major difference between DES and AES is that latter is byte oriented. To properly understand how data flows through each layer of AES, it is important to know how data is arranged. Each block (16 bytes A0,A1,...,A15) of input data is in the form of 4x4 byte matrix:

$$\begin{bmatrix} A0 & A4 & A8 & A12 \\ A1 & A5 & A9 & A13 \\ A2 & A6 & A10 & A14 \\ A3 & A7 & A11 & A15 \end{bmatrix}$$

In the same way key bytes are arranged depending on the key length. 128-bit key forms 4x4

matrix, 192-bit key forms 4x6 matrix and 256-bit key 4x8 matrix. Following is an example of a 192-bit key matrix:

$$\begin{bmatrix} K0 & K4 & K8 & K12 & K16 & K20 \\ K1 & K5 & K9 & K13 & K17 & K21 \\ K2 & K6 & K10 & K14 & K18 & K22 \\ K3 & K7 & K11 & K15 & K19 & K23 \end{bmatrix}$$

We will not discuss the mathematics of AES as it is out of the scope of this thesis.

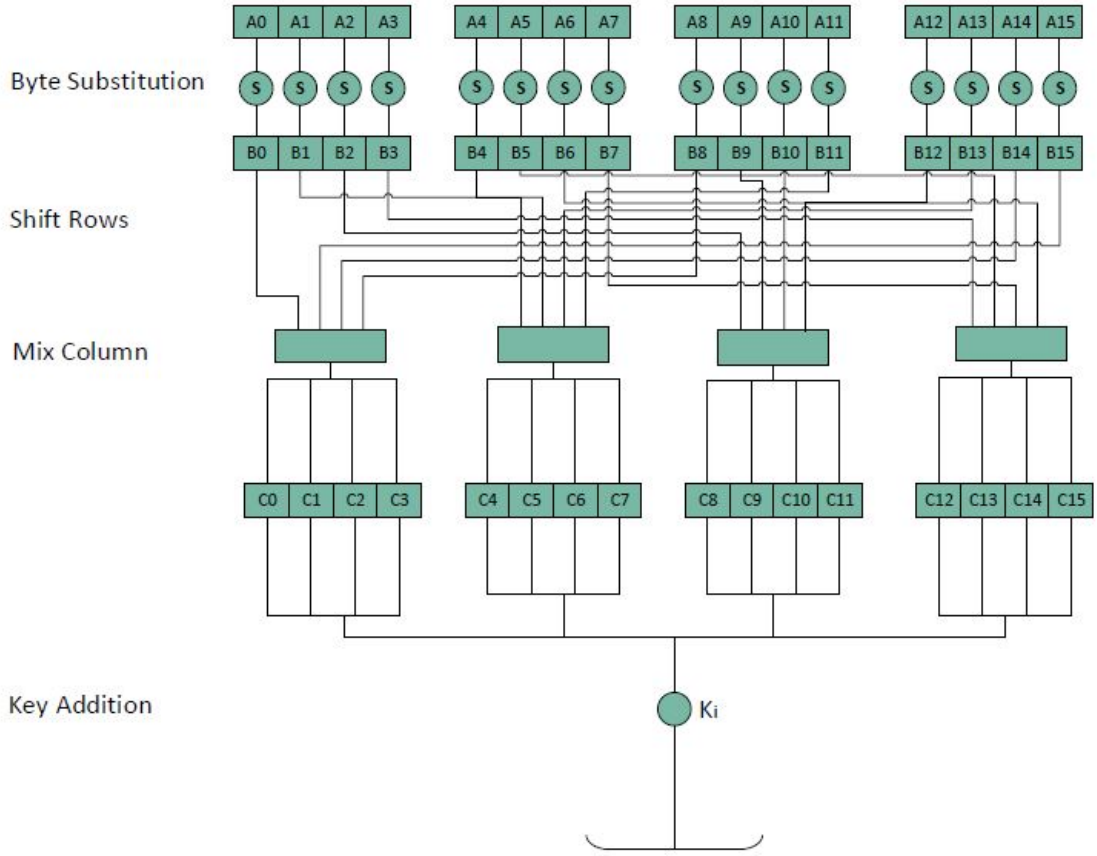

Figure 2.3 : AES Round Functions from 1-nr [61]
Block diagram of internal structure of AES shows each layer with its respective product . A is the input. B and C are intermediate products of byte substitution and diffusion (shift row and mix column) respectively. Ki is the sub key of respective round.

2.5.1 Byte Substitution

During encryption and decryption, each value of the current block is replaced by its corresponding S-Box and Inverse S-Box value respectively. Figure 2.4 is an S-Box, for example 19 (HEX) is one value of the current block. It will be replaced by D4 (HEX). Go straight to the intersection of 1 (vertical index) and 9 (horizontal index) which is D4. Figure 2.5 is the corresponding inverse S-Box. During decryption, D4 (HEX) will be replaced by 19 (Intersection of D and 4).

	0	1	2	3	4	5	6	7	8	9	A	B
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F

Figure 2.4 : Substitution Box [61]
S-Box used by Byte Substitution layer during encryption. Each value in plaintext block is replaced by its corresponding value of S-Box.

2.5.2 Shift Row

The Shift Row layers performs circular shift on 4x4 matrix. The matrix is formed column wise but shifted row wise. If we have a 16 byte block with following values:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Its 4x4 matrix will be:

$$\begin{array}{ll} \text{Row 1} & \begin{bmatrix} 1 & 5 & 9 & 13 \end{bmatrix} \longrightarrow \text{No shift} \\ \text{Row 2} & \begin{bmatrix} 2 & 6 & 10 & 14 \end{bmatrix} \longrightarrow \text{One position right shift} \\ \text{Row 3} & \begin{bmatrix} 3 & 7 & 11 & 15 \end{bmatrix} \longrightarrow \text{Two position right shift} \\ \text{Row 4} & \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix} \longrightarrow \text{Three position right shift} \end{array}$$

	0	1	2	3	4	5	6	7	8	9	A	B	C	D
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21

Figure 2.5 : Inverse Substitution Box [61]
Inverse S-Box used by Byte Substitution layer during decryption. Each value in ciphertext block is replaced by its corresponding value of Inverse S-Box.

Resultant matrix:

$$\begin{bmatrix} 1 & 5 & 9 & 13 \\ 6 & 10 & 14 & 2 \\ 11 & 15 & 3 & 7 \\ 16 & 4 & 8 & 12 \end{bmatrix}$$

2.5.3 Mix Columns

Mix Column operation is all about matrix multiplication. We have two 4x4 matrices. One is input and another is multiplication matrix.

16 byte Input Matrix **Multiplication Matrix**

$$\begin{bmatrix} b1 & b5 & b9 & b13 \\ b2 & b6 & b10 & b14 \\ b3 & b7 & b11 & b15 \\ b4 & b8 & b12 & b16 \end{bmatrix} \qquad \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Each value of the input matrix is multiplied with all the values of the multiplication matrix. Matrix multiplication is carried out in traditional way. (*) is Galois Field Multiplication and instead of adding values we XOR them. Multiplication of one column of input matrix is given below:

b1 = (b1*2) XOR (b2*3) XOR (b3*1) XOR (b4*1)

b2 = (b1*1) XOR (b2*2) XOR (b3*3) XOR (b4*1)

b3 = (b1*1) XOR (b2*1) XOR (b3*2) XOR (b4*3)

b4 = (b1*3) XOR (b2*1) XOR (b3*1) XOR (b4*2)

2.5.3.1 Galois Field Multiplication

Mathematics of Galois Field is beyond the scope of this thesis. We will discuss multiplication with the help of 2 HEX look up tables. Matrices must be in HEX format with maximum two

digits. Look up table L and E give values of multiplication and addition respectively. Figure 2.6 and 2.7 are L and E Tables respectively.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

Figure 2.6 : L Table [61]
Instead of calculating value of each Galois field multiplication, a look up table is generated that gives values of multiplication.

For example, if we have (b1*2)

From table L, b1=AF

Go straight to the intersection of A (vertical index) and F (horizontal index) which is B7. Now check for 02, which is 19.

So, AF*02=B7+19=D0

If the result is greater than FF, e.g.

7B+B4= 12F > FF

Then

12F-FF=30

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

Figure 2.7 : E Table [61]
Instead of calculating value of each Galois field multiplication, a look up table is generated that gives values of multiplication.

Exclusions:

Above procedure will not be applied in two cases:

1. 1*x=x

2. 0*x=0

After performing all the multiplication operations and XORs, we will get one value. For example, if we have below mentioned equation

b1 = (b1 * 2) XOR (b2*3) XOR (b3*1) XOR (b4*1)

And after performing all operations we get 8A

b1=8A

Now, we will use look up table E

8A=32

Hence, b1=32

All these operations will be done on whole input matrix, and then we will get the result of Mix Column.

2.5.4 Key Addition

Key Scheduler generates sub keys. Each sub key is XORed with 16 byte block of input. Each round uses new sub key. In first round, 1-16 indices of key are used then in second round, 17-32 indices are used and so on. Figure 2.8 shows key addition of first two rounds.

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 2.8 : Key Addition
Sub keys are generated through key scheduler and each sub key is of 16 byte. In each round, 16 byte input block is XORed with 16 byte sub key. Every round has its own sub key.

2.6 Summary

In this chapter, Advanced Encryption Standard is explained in detail. We discussed how AES came into being along with the internal structure and working of AES. Mathematics is not discussed in this chapter as it is out of the scope of this thesis.

Chapter 3

Literature Review

3.1 Introduction

This literature review provides research background on parallel implementation of Advanced Encryption Standard Algorithm. In section 3.2, literature is divided into two approaches; hardware approach and software approach. A brief summary of research work regarding hardware implementation of AES is given in Section 3.2.1 and software implementation of AES in Section 3.2.2. Software approach is partitioned into two types; Central Processing Units (CPUs) and Graphical Processing Units (GPUs) in sections 3.2.2.1 and 3.2.2.2 r respectively. Summary of this chapter in Section 3.3, categorizes gap in the literature as a framework for the unique contribution of this study.

3.2 Parallel Implementation of AES

With the inception of AES, researchers are working to enhance the performance of AES. There are two methods used for speedup:

1. Software Approach

2. Hardware Approach

3.2.1 Software Approach

CPU and GPU have tremendous computational power depending on their configuration. They are used in wide variety of applications where computationally expensive tasks are done. Cryptography is one of the most important applications. Symmetric and asymmetric algorithms require considerable computational power for their execution. Besides security, efficiency of an algorithm plays a vital role in real world applications where we need faster encryption and decryption process. GPUs or CPUs can execute AES in parallel with multiple threads as it operates on 16-bytes of independent blocks. Work of some researchers is discussed in following section.

3.2.1.1 Central Processing Unit

Parallelizing AES on a CPU has been thoroughly explored in recent years. This is because there are still some computers that lack GPUs or equipped with old GPUs that do not support general computing.

Holget et al parallelized AES using OpenMP on ARM MPCore featuring four ARM11 processor cores. They investigated the parallelization effect on performance to power ratio by inspecting parameters like number of threads in a running program. They derived functional and instruction level power model for MPCore with power efficient codes. They achieved average speed up factor of 3.4 and efficiency gain factor of 1.8 [16].

Ortega et. al. implemented parallel AES with Open MP on multicore CPU system and CUDA on GPU. For openMP, data is divided into 16 bytes of blocks and assigned to threads. These threads go to specified cores and encrypt data. Their study proved that OpenMP is a good for paralleling AES [54].

Pachori et. al. presented the parallel implementation of AES using Java Parallel Programming Framework (JPPF). They used control and data parallelism and their design provided performance improvement. Four layers of AES are divided into two independent parts thus modifying structures of original AES. These independent parts are assigned to separate processors. These processors exchange intermediate results during processing and further process them. After that single processor combines output of these independent parts and encrypts them. Their study showed significant improvement [56].

Pousa et. al. followed the same line and realized 3 parallel AES algorithm versions by making use of MPI, CUDA and Open MP respectively. Researchers parallelized AES with OpenMP by initializing number of threads equal to total number of cores available. Key is generated sequentially. Each block of data is assigned to each thread which is in turn assigned to each core. Each core encrypts 16 bytes of block independently. In case of MPI, number of processes equal to number of available processors is generated. Data is divided in sets of 16 bytes and each block is assigned to each process. This processes goes to separate processor and encrypts respective blocks independently [60].

Duta et. al. realized parallel AES by making use of CUDA, Open MP, and Open Computing Language (OpenCL) respectively. They not only parallelize AES but also emphasize importance of I/O operations by implementing AES in Cipher Block Chaining mode and Interleaved Cipher block chaining mode. They achieved remarkable performance by using parallel programming models and APIs [24].

Navalgund et. al. parallelized AES by dividing it into parallel and non-parallel parts and implemented parallel parts with OpenMP. It was also used to deal with synchronization and data dependency problems. It was done by using join and fork model. Results have shown significant improvement and authors claim this implementation to be useful for hardware implementation [52].

As we are discussing parallel implementation of AES, parallelism is of different types. Most common types are task, instruction, loop and data level parallelism. Liu et. al. examined granularity of task and data level parallelism and mapped sixteen implementations of AES with offline and online key expansion on a fine-grained multicore system. Their smallest design requires 6 processors. Fastest design achieved throughput of 4.375 cycles/byte. They optimized area of all 16 implementations by examining load on each processor and reduced it by 18%. They have shown efficiency in terms of throughput and energy as compared to available AES implementations on multicore systems and demonstrated fine-grained multicore system to be an excellent solution for AES software implementation [47].

Nagendra et. al. analyzed the efficiency of the AES using Open MP. They compared sequential and parallel algorithm execution time and found that parallel AES cipher implemented on Dual Core system is faster than sequential AES. They observed that performance was less for small file size but increased by increasing file size and became constant after particular file size. Time reduced by 38% for a small size and as much as 45% for large files. Subsequently time reduced with increasing file size. Their research work proved that multi-core processors can be efficiently used to speed up algorithms[51].

Elkabbany et. al. used multi processors to parallelize AES. It is scalable and suitable for real time applications. As AES has inherent qualities of parallelism, so they took advantage of this characteristic and used pipelining along with parallelization to accelerate AES. Most of the researchers use pipelined parallelization or MixColumn parallelization. But these researchers pipelined all rounds and parallelized MixColumn and Add Round Key transformations. In order to enhance system performance, instead of pipelining 9 stages unlike previous studies, they pipelined 11 stages. Their results demonstrated that pipelining improved results upto 95% while introducing parallelization of MixColumn and Add Round Key elevate it upto 98%. These results were achieved by using 8 and 16 processors respectively for encryption and decryption.

But it should be noted that after a certain limit, performance of algorithm degrades by increasing number of processors[26].

Pendli et. al. analyzed the effectiveness of the AES using Open MP on dual core processor. This process is validated using JAVA platform. They compared sequential and parallel algorithm execution time and found that parallel AES cipher implemented on Dual Core system works 40-45% faster than sequential AES. Their research work proves that multi-core processors can be efficiently used to speed up algorithms[59].

3.2.1.2 Graphical Processing Unit

Graphics processors (GPUs) are hardware accelerators that perform computationally intensive tasks that general purpose processors (CPUs) cannot perform efficiently. GPUs possess large number of simple deeply multithreaded cores. GPU architectures are fully programmable processors. Today the raw computational power of GPUs has exceeded that of high end general purpose CPU.

Manavski was the first to use CUDA for parallelizing AES on GPU. The performance of newly developed algorithm was compared with CPU implementation. This research opened new ways for parallelizing algorithms. The proposed implementation offered same range of performance as other hardware based implementation and was 20 times faster than OpenSSL implementation [48].

Biagio et. al. used coarse and fine grained approaches to implement AES using CUDA programming model. They inferred that GPU as a co-processors works efficiently for AES and this is cost effective as compared to CPU based AES implementation on OpenSSL [15].

Bos et. al. implemented AES on GPUs and reported problems that were encountered in this process. They also evaluated if GPU was a good option for cryptography or not and concluded that GPU was a good option for cryptography as compared to CPU. Although there

were problems but they could be alleviated by using different programming models like CUDA and AMD/ATI's Close to Metal (CTM) [17].

Le et al devised a way for overcoming the issue of low throughput of AES encryption process on CPUs. They proposed and implemented AES on GPU and gained a speed up of 7x as compared to CPU. They claimed that this scheme can be used in digital forensics to achieve high encryption and decryption rates [43].

To encounter the problem of low throughput of AES encryption on CPUs, Tran et. al proposed and implemented new algorithm for parallel AES encryption on GPU. Their algorithm showed significant performance gain[74]. Ortega et. al. concluded from their research that GPU encrypts data with AES in economical way and is faster as compared to sequential code or code that has been parallelized using OpenMP [54].

GPUs are very helpful in processing multiple types of data concurrently. The processing power of CPUs cannot be compared with GPUs. But we can combine CPUs and GPUs to take advantage of their power. CPUs have large and fast cache memories and are of great benefit if data transfer rate is greater than processing time or instruction branching is obstructing continuous processing of instruction on all GPU cores. Barlas et. al. proposed a framework for optimal utilization of CPU and GPU resources combined for encryption and decryption in block ciphers like AES. They have highlighted all the aspects that must be kept in mind while using GPUs in any application [14].

Li et. al. proposed a CUDA implementation of AES in CBC and ECB mode on GPU. One important concept that we will discuss in next chapter is granularity which is division of task into number of instructions. Granularity of the proposed implementation is 16 bytes (one block) per thread. It showed 50 times speed up than sequential implementation. They also discussed optimization techniques for real world application [44].

Nhat et. al. proposed a technique for parallel execution of AES in counter mode on GPUs.

They increased the block size (16 bytes) of plaintext across the boundaries of data by a factor E to create larger blocks. This approach helps in decreasing the distribution time of blocks to cores, their synchronization, and number of encryption function calls by factor E at the end of whole process as compared to coarse grained approach. It gives significant performance gain on general purpose multicore system and GPU [53].

Mocanu et. al. implemented parallel AES on General Purpose Graphical Processing Unit (GPGPU) platforms from different manufacturers using C++ Accelerated Massive Parallelism (C++ AMP) and compared results with CPU based parallel and sequential implementation [50].

Guo et. al tested sequential implementation with improved fast AES implementation which was 50 times faster. After that they tested Intel Advanced Encryption Standard New Instructions (AES-NI) extended instruction sets with fast implementation. And Intel AES-NI extended implementation was 50 times faster. Lastly, they parallelized AES on GPU with CUDA and found that it is 18 times faster than fast AES implementation. AES has lot of practical applications [35].

Fei et. al. proposed parallel AES algorithm for encryption on cloud for massive users' data by using GPU parallelism or CPU parallelism. They decomposed data into small chunks and encrypted them. They proposed 6 algorithms with different schemes. These 6 algorithms were implemented on two different platforms and found that CPU and GPU parallelism improved results greatly [29].

Fei et. al. worked on another application of AES. They proposed a secure file system using parallel AES and secure hashing algorithm. They implemented their proposed system by using CPU parallelism, GPU parallelism and hybrid of both. They tested on a pair of representative platforms [30].

3.2.2 Hardware Approach

After NIST approved AES in 2001, most of research effort was headed towards effective hardware implementations of AES [20] algorithm. The former schemes mostly focused on high speed thoroughly pipelined implementations, and after that research shifted on low area and low power architectures keeping in view economical devices and feedback operational modes. Current research work is mostly focused on decreasing the execution time of algorithm.

FPGAs are a smart choice [9] when it comes to hardware implementation of AES. FPGAs were primarily used as glue logic. Due to incredible growth of technology in past few years, FPGAs are being used in many complex applications like cryptographic applications are one of them. Latest cryptographic applications are being increasingly realized on FPGAs, varying from fully parallel pipelined to compact, low cost and low power architectures. FPGAs are used most commonly because ASICs require more time to market as compared to FPGAs. Secondly, developmental process of FPGAs is a lot more effective and economical. Furthermore, unlike ASICs, the reconfigurable nature of FPGAs provides the provision to modify the already implemented algorithm. This characteristic is helpful in reconfiguring the device at run time by addressing flaws from already implemented design and can be optimized for fixed set of requirements. Contributions of different researchers in this regard have been discussed below.

Gaj et. al implemented 5 AES finalists by using iterative architecture. This type of architecture was appropriate for feedback cipher modes. The FPGA device used was Xilinx Virtex XCV-1000. They also implemented four non-feedback cipher mode AES algorithms by using full mixed inner-round pipelining and outer-round pipelining. All AES nominees achieved roughly same throughput. Their research work supported Rijndael as the new AES [32].

Fischer et al. assessed two different approaches of mapping Rijndael algorithm to FPDs. They also worked on suitability of these approaches to available FPD families. Results of these scheme were implemented on Altera FLEX, ACEX and APEX FPD. They were contrasted

with the efficient known Xilinx FPGA implementation at that time and result came out to be faster than others. The Altera ACEX FPD proved to be a tremendous device for implementing very fast AES on reconfigurable hardware. ACEX FPD family fits cost sensitive encryption applications. A new solution based on T-boxes allowed AES implementation with the same performance of encryption and decryption [31].

Chitu et. al. implemented finite state machine-based encryption/decryption of AES (128 bit block and key size) on FPGA that took 84% of total area. The architecture was implemented on Virtex-II XC2V100-4 device which achieved throughput of 0.739Gb/s [19].

Chodowiec et. al. proposed and implemented compact 128 bit AES architecture on FPGA for embedded applications. They used folded register concept and implemented it on low cost Spartan II device that has achieved 1.3Gb/s throughput. They explored new way of implementing MixColumn and Inverse MixColumn transformations [20].

Compact and high speed architectures were proposed by Standaert et. al. They not only did algorithmic optimizations of S-Box but also proficient arrangements between Mix Column, Shift Row and key addition layer. To cater place and route constrictions, heuristic rules were applied for optimized efficiency. It was implemented on VIRTEX-E technology. This design yielded throughput up to 18.5 Gb/s. The throughput/area ratio improved with area requirements limited to 542 slices and 10 RAM blocks [73].

Rouvroy et. al. made an assembly suitable for small embedded applications, as key was precomputed before the encryption or decryption process. This design was implemented on Virtex-II device that delivered 0.358Gb/s throughput [65].

Zhang et. al. used sub-pipelining to increase the efficiency of design. The design did not use lookup tables instead they used combinational logic. Composite field arithmetic reduced the area of design and achieved 21.566Gb/s throughput on Xilinx XCV1000 e-8 device [80].

Hodjat et. al. worked on inner and outer round pipelining along with loop unrolling methods

and achieved throughput of 21.54Gb/s. This design was implemented on Virtex-II Pro device [38].

Yoo et. al. proposed a parallel pipelined high speed architecture. They achieved 29.77Gb/s throughput by employing an effective inter and intra round pipeline scheme [77].

Kotturi et al increased the throughput of AES algorithm by proposing a hardware efficient design. It was high speed parallel pipelined architecture. It utilized inter and intra round pipelining. Their design yielded encryption throughput of 29.77 Gb/s [42].

Hamalainen et. al. introduced 8 bit low area and power architecture. Area was reduced by coalescing partial storage of state and Shift Row operation into the byte permutation unit. Power was reduced by executing Mix Columns operation in parallel with remaining of the cipher. The architecture attained throughput of 0.121Gb/s [36].

128 bit key AES algorithm was proposed by Liberatori et. al. Key was precomputed and stored in the memory. The circuitry to store keys was included in the architecture. The 8 bit FPGA implementation was done on Altera Flex 10K EPF10K20 that delivered 1.1Gb/s throughput [45].

128 bit AES cipher processor had been designed by Fan et. al. They used new high speed and hardware functional sharing blocks technique. The AES functional calculations havd been applied on four layers, which were SubBytes, ShiftRows, MixColumns and AddRoundKey. The FPGA tool used was Xilinx ISEtrade 7.1 with XSTtrade synthesizer. Proposed sequential AES design can reach operational frequency of 75.3 MHz and throughput of 0.876 Gb/s while in fully pipelined AES design it can reach 250 MHz and 32 Gb/s respectively [27].

Gielata et. al proposed pipeline architecture that was used to increase the efficiency of implementation in terms of performance and flexibility of AES algorithm. It was implemented in Virtex4 series of Xilinx using VHDL language. They accomplished throughput of 21.2 Gb/s in case of encryption and 16.6 Gb/s for decryption [33].

Qu et. al developed a pipelined AES architecture in Counter mode to provide the high through-

put. This was done while implementing byte transformation in 1 clock cycle by placing some registers in appropriate places to shorten the delay. It was implemented on Xilinx Foundation ISETM 10.1 FPGA. It achieved throughput of 73.737Gbps in 576.07MHz frequency with resource efficiency of 3.21Mbps/LUT [62].

Van et. al. worked on three low power encryption schemes that tried to attain best power results without degrading the throughput of design. These schemes were then compared with each other in terms of area, and power consumption [75].

FPGA possesses qualities of hardware and software. Granado et. al. took advantage of these qualities and implemented AES on FPGA by using 3 hardware languages Handel-C, VHDL and JBits. This methodology had partial and dynamic reconfiguration along with parallel and pipelined architecture. This methodology can be applied to other cryptographic algorithms. This design achieved throughput of 22.922 Gb/s and efficiency (throughput/ratio area) of 6.97 Mb/s per slice [34].

Soliman et. al. proposed a high throughput crypto coprocessor for a general purpose processor implemented on FPGA for encryption/decryption. It was implemented on Xilinx Virtex V FPGA with VHDL programming language. This co-processor hid memory latency as it was built on a decoupled architecture. Architecture also consisted of AES pipelines. Four pipelines could yield throughput of 222 Gb/s at 444 MHz. While decreasing the frequency could reduce power utilization and offered scalable system [71].

Hoang et al. implemented AES that resulted in high throughput. It is best for applications that require high performance and high speed. The architecture was 128 bit AES cipher based on iterative looping approach and S-Box lookup table implementation. This simplified architecture and effortlessly achieved low latency and high throughput [37].

Banu et. al worked on increasing the throughput of 128 bit AES algorithm by employing different hardware and software techniques. They used pipelining technique in hardware to

speed up algorithm by executing multiple rounds in parallel. It was implemented using Xilinx xc5vlx110t-1 device. It achieved throughput of 31.25Gbps [13].

Folded parallel architecture is proposed by Rahimunisa et. al, with the aim to reduce delay through implementation of byte transformation step in one clock cycle. They inserted registers in suitable places. They achieved throughput of 73.737Gbps [63].

Farashah et. al. presented digital design of 128 bit AES cantered on the 2-slow re-timing method. They had attained 86Gb/s throughput and 671.5241 MHz maximum operational frequency on Virtex-5 device [28].

Anwar ey. al. proposed parameterized crypto coprocessor based on AES which was fully pipelined. Different number of AES rounds encountered different latencies varying to the application requirement. If low-latency design is application constraint, the number of rounds is increased whereas if we have limited area, number of rounds is reduced. Effect of number of rounds on area, latency and throughput had been explored by performing different experiments. Parameterized crypto coprocessor worked according to application demand [9].

Soltani et. al. proposed a fully pipelined architecture for AES that yielded excessively high throughput on FPGA. They used full and sub pipelining, loop-unrolling methods in their implementations. Their finest implementation on Virtex-6 device achieved 260Gb/s throughput [72].

AES can be implemented in hardware with minimum hardware utilization; it is more secure and costs less. This approach was presented by James et. al. Lightweight block ciphers were efficiently implemented in hardware. An approach, to make AES a lightweight block cipher, was being discussed such that designing the steps of AES such as mix columns, substitute byte in AES was to be implemented in a parallel manner. The latency in this implementation was considered to be less comparing to the conventional implementation of AES. The conventional and the new approach were simulated in XILINX 14.2 and compared in the aspects of area and latency. The design was implemented in SPARTAN 6 FPGA [39].

Oukili et. al. used pipelining to enhance speed and maximum operating frequency. The implementations had been successfully done by virtex-6 (xc6vlx240t) FPGA. Proposed unmasked and masked architectures in this technique were very fast. They achieved a throughput of 93.73Gbps and 58.57Gbps, respectively [55].

3.3 Summary

A brief review of the literature has been discussed to give an overview of the techniques that have been used to parallelize AES. These techniques are divided into hardware and software approaches. The literature review has shown that although hardware approach gives high throughput but cost of implementation is high too. On the contrary, software implementation is economical but does not yield as high throughput as hardware implementation. Depending on the nature of application, hardware or software approach is selected. Software implementation can be done on CPUs and GPUs. If we compare CPUs and GPUs, GPUs yield higher throughput but there are some computers that lack GPUs or equipped with application specific GPUs. CPU implementation is economical as CPUs are readily available. Researchers have worked on variety of languages and platforms to parallelize AES. A parallel implementation of AES using MPJ Express is of very preliminary nature in literature [25]. So we did a detailed study on parallel implementation of AES using MPJ Express and accessed its performance.

Chapter 4

Parallel Implementation of AES using MPJ Express on HPC Platform

4.1 Introduction

Performance is an ongoing topic in the field of information technology. It is an unending race. Parallelization is a common technique for increasing performance. In this chapter, we will discuss the proposed parallelization methodology for increasing the performance of AES. We begin with the introduction of multicore and cluster systems in Section 4.2. Section 4.3 is related to the concepts of parallel programming that leads to our methodology. This section is further divided into subsections which are Parallel Programming Model, Data Parallelism, Message Passing Programming and Message Passing Model which consists of MPJ Express Library. Section 4.4 comprises of motivation for this research work. Section 4.5 provides Methodology that explains the parallelization algorithm. Section 4.6 summarizes the chapter.

4.2 Multicore and Cluster Systems

Earlier computers were uniprocessor in which single central processing unit was used to execute all computer tasks sequentially. While multiprocessors, as opposed to single processor, refers to executing multiple contemporary processes in parallel [12]. Top most chip producers started to create chips, each chip having processors with numerous power effective computing elements. There were physical causes that resulted in the technological growth towards multicore processors. Overheating restraints the clock speed as transistors on a chip increases. These processors have independent control and memory units. Generally, core is referred as single computing unit unlike multicore that refers to complete processor possessing numerous cores. Multicore processors deliver enhanced performance as they carry multiple cores per processor. Therefore, multicore processors form a small parallel computer system. Figure 4.1 is showing a typical multicore architecture.

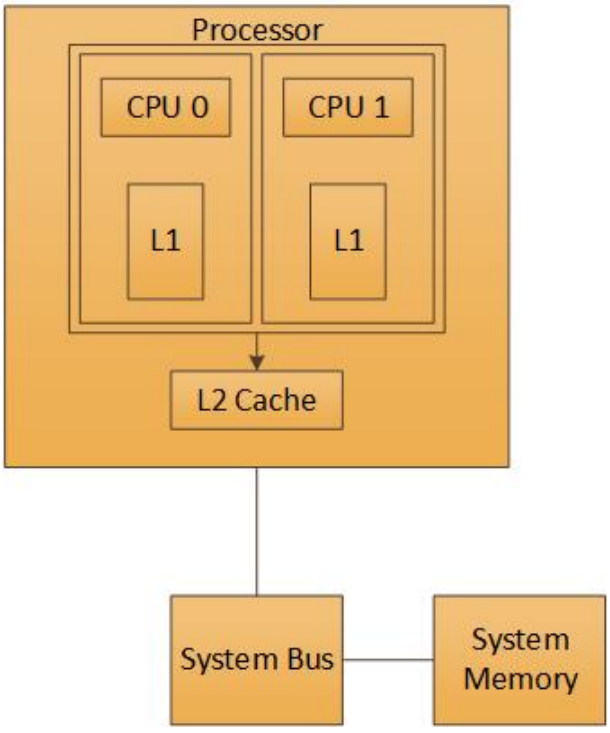

Figure 4.1 : Multicore System Architecture [22]
A multicore system consists of N processing cores. Each core has its exclusive Level 1 (L1) cache. All this is integrated on a single chip. They share a common Level 2 (L2) cache. All processing cores share the bandwidth among main memory and L2 cache.

To completely exploit the power of multicore processors, thread level parallelism must be used [67]. A multicore system consists of N processing cores. Each core has its exclusive Level 1 (L1) cache. All this is integrated on a single chip. They share a common Level 2 (L2) cache. All processing cores share the bandwidth among main memory and L2 cache. Multicore system supports concurrent multithreading. Figure 4.2 shows a multicore system with multithreading. Individual threads can run concurrently on same core. Each thread can execute different task independently [40]. Integrating multiple multicore systems connected by a high speed network makes up a cluster system as shown in Figure 4.3 . Each multicore system is referred as a node. Nodes can work independently or work together to perform a task in parallel. Such cluster systems provide enhanced performance, hence called as HPC platform.

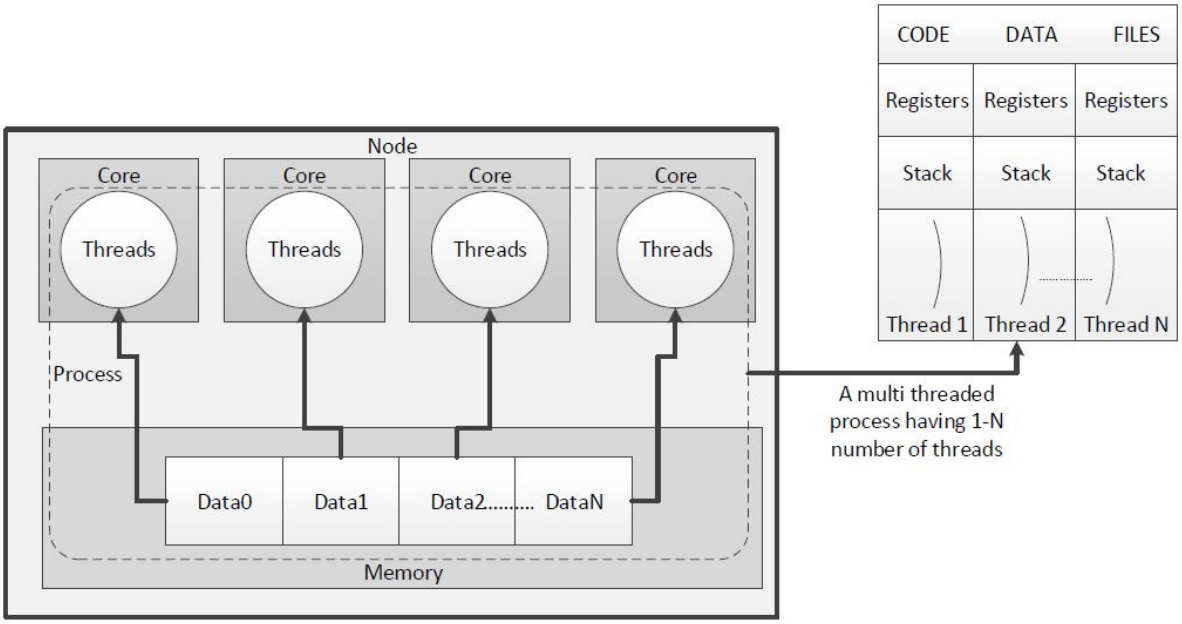

Figure 4.2 : Multicore System with Multithreading [69, 4]
A multicore system consists of N processing cores. All processing cores share the bandwidth among main memory. Multicore system supports concurrent multithreading. Individual threads can run concurrently on same core. Each thread can execute different task independently.

Symmetric and asymmetric algorithms are complex and require high computational power. The sequential execution of the algorithms would need a significant amount of execution time. This may not be practicable for most of the applications that require faster encryption and decryption

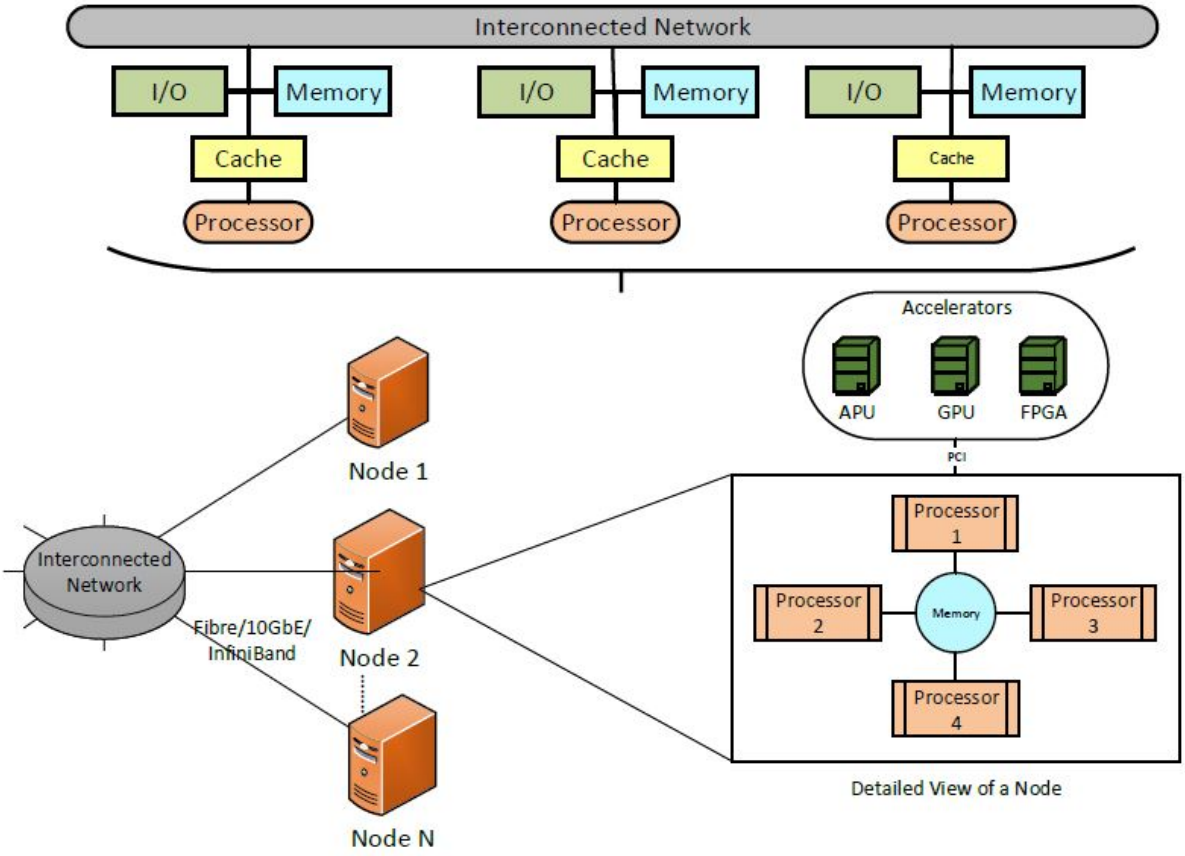

Figure 4.3 : Modern HPC Cluster Platform [7, 3]
Integrating multiple multicore systems connected by a high speed network makes up a cluster system. Cluster system can be homogeneous or heterogeneous depending on the processing power of processors.

rate to meet the required data flow [12]. For the sake of increased speed and enhanced experience, significance of performance can not be neglected. There are two ways to increase the speed of computer systems; increase CPU frequency/clock speed or increase number of cores. Former method provides poor reliability and high power consumption [30].

Multiple cores are global microprocessor chips. It enables the chip manufacturers to gain high collective performance while circumventing the high power consumption of high frequency CPUs. As shown by the systems on the TOP500 list [6], todays sophisticated computer manufacturing companies take benefit of economies of scale by using multiple multicore chips to build large HPC systems. The leading HPC architectures consists of clusters of such nodes joined together through high speed network [79].

4.3 Parallel Programming

Primary phase of parallel programming is to convert the given application problem into parallel algorithm. First of all divide application into several parts with respect to computations, called tasks. Execution of these tasks can be done in parallel on cores/processors in multicore or cluster system. The size of tasks (number of instructions) depends on the programmer, it is referred as granularity. Granularity is of two types

- Coarse grained $\longrightarrow$ Tasks with numerous instruction
- Fine grained $\longrightarrow$ Tasks with few instructions

If granularity of the task is extremely fine grained, the cost of mapping tasks to processors and their scheduling is enormous and yields a substantial total execution time. Therefore, task division needs to be a right compromise of number of tasks and granularity. The tasks of a program are then allocated to threads or processes, which are then allocated to cores/processors for execution. Based on the memory structure of execution environment either thread or process is used. Thread is more appropriate for shared memory environment like multicore processors while processes are suitable for distributed environment like cluster system. A process can consists of several threads but vice versa is not possible. We can say that thread is a lightweight task while process is a heavy weight task.

Scheduling is the allocation of tasks to processes/threads and defines the order of task execution. The tasks of a program may be independent or dependent on each other. Dependent tasks surface the issue of data and control dependencies, which means that tasks must execute in a specific order. Management and Synchronization of threads/processes is required for orderly execution of tasks. Figure 4.4 illustrates the synchronization of threads/processes. Associated with all these is the time for data synchronization and computation on processors/cores called as parallel execution time. Speed up and efficiency are calculated by comparing the sequential

execution time on one processor with the resultant parallel execution time, for a quantifiable assessment of the parallel execution time of programs. [64]

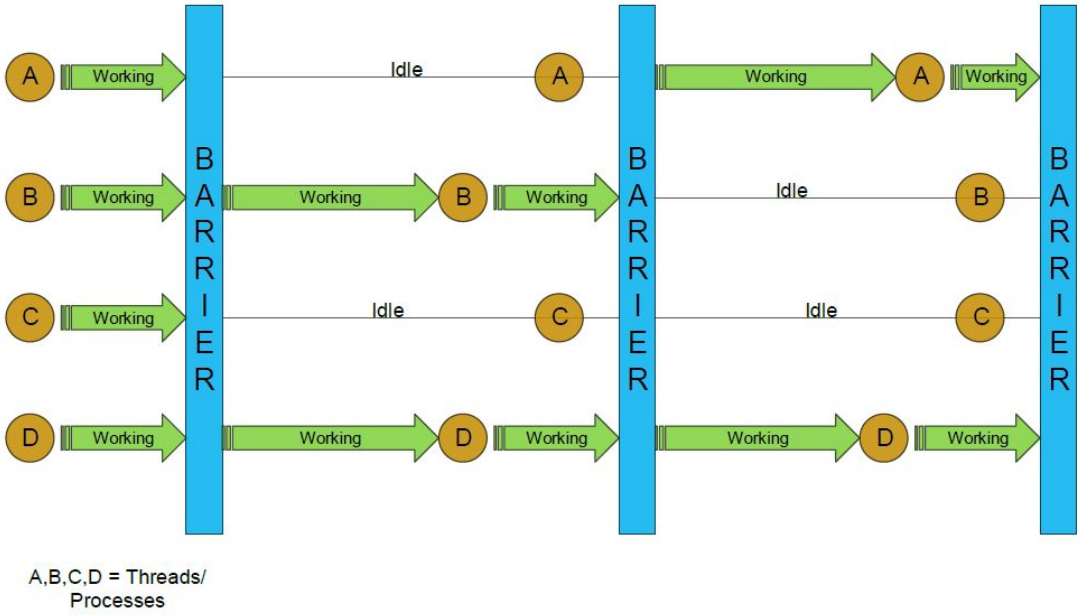

Figure 4.4 : Synchronization of Threads/Processes [1]
Management and Synchronization of threads/processes is required for orderly execution of tasks. All the threads/processes stop at a barrier and wait for other threads/processes in the group to reach barrier before proceeding further.

4.3.1 Parallel Programming Model

There are many models of parallel systems like machine, architectural, computation and programming model [64]. Parallel programming model is an abstraction of parallel computing architecture and defines a parallel computing environment with respect to semantics of the programming language/environment. This vision of parallel computing environment is affected by the language, runtime libraries, complier and architectural design. Therefore, same architecture possesses many different parallel programming models. Parallel programming models can differ in many ways:

- The level of parallelism (statement , parallel loops , instruction or procedure)
- Parallel program design is implicit or explicit

- The specification of parallel program units
- The execution mode of parallel units (Single Program Multiple Data /Single Instruction Multiple Data, synchronous/asynchronous)
- The interchange of information between computing units (shared variables/ explicit communication)
- Synchronization between parallel parts

4.3.2 Data Parallelism

In many applications, different components of a larger data require same operation. This can be done in parallel, if components are independent of each other. The components are divided equally among the processors. Each processor is the owner of its data and applies same operation on it. This kind of parallelism is same as Single Instruction Multiple Data Model and is known as data parallelism. Data parallelism is used via extending sequential programming languages by incorporating special constructs to express parallel operations. It can be utilized for both distributed and shared memory. For a distributed address space, each processor needs to get the program data in its local memory on which it has to do computations.

The programming models for distributed memory space are different. These are message passing programming models. Communication operations are performed by exchange of data between participating processes. This exchange of data between participating processes is done through messages, such that one process receives the data stored in another process's local memory. Dozens of parallel programming languages have been introduced in the past. Many of them are high level languages that simplify various aspects of managing parallelism. However, no single high level parallel language has gained widespread acceptance in the parallel programming community. As a result, most parallel programming made use of FORTRAN or

C, enhanced with message passing functionality. MPI is the most popular message passing standard backing parallel programming. Nearly, every commercial parallel computer supports MPI and other open libraries supporting MPI standard. [49]

4.3.3 Message-Passing Programming

Message passing library interface specifications are standardized and referred as message passing interface. Figure 4.5 is a Message passing model. It outlines the library routines for standard communication patterns. There is a group of processors, each processor having its private memory with data and instructions stored in it. Processors are linked by an interconnection network. Processor A can send its data through message to processor B, thus providing indirect access of its data to processor B. In this model, all processors can communicate with each other. The user provides number of processes upon program execution that remains constant throughout the program. Each process carries its own unique ID. Typically all processes perform same operation but they can perform different functions as well depending on application. Each process works on its local data and communicates with other processes or I/O devices. Message passing model has variety of advantages over other parallel programming models.

- Message passing programs are suitable for Multiple Instruction Multiple Data architectures. They are a natural fit for multi computers. Message passing programs can be executed on multiprocessors by using shared variables as message buffers but have no support for global memory. Message passing model provides the multiprocessor programmer with the tools needed to manage the memory hierarchy.

- Debugging message passing program is easier than debugging shared memory program. As each process controls its own memory, it is not possible for one process to by chance overwrite a variable contained by another process, a common bug in shared variable programs.

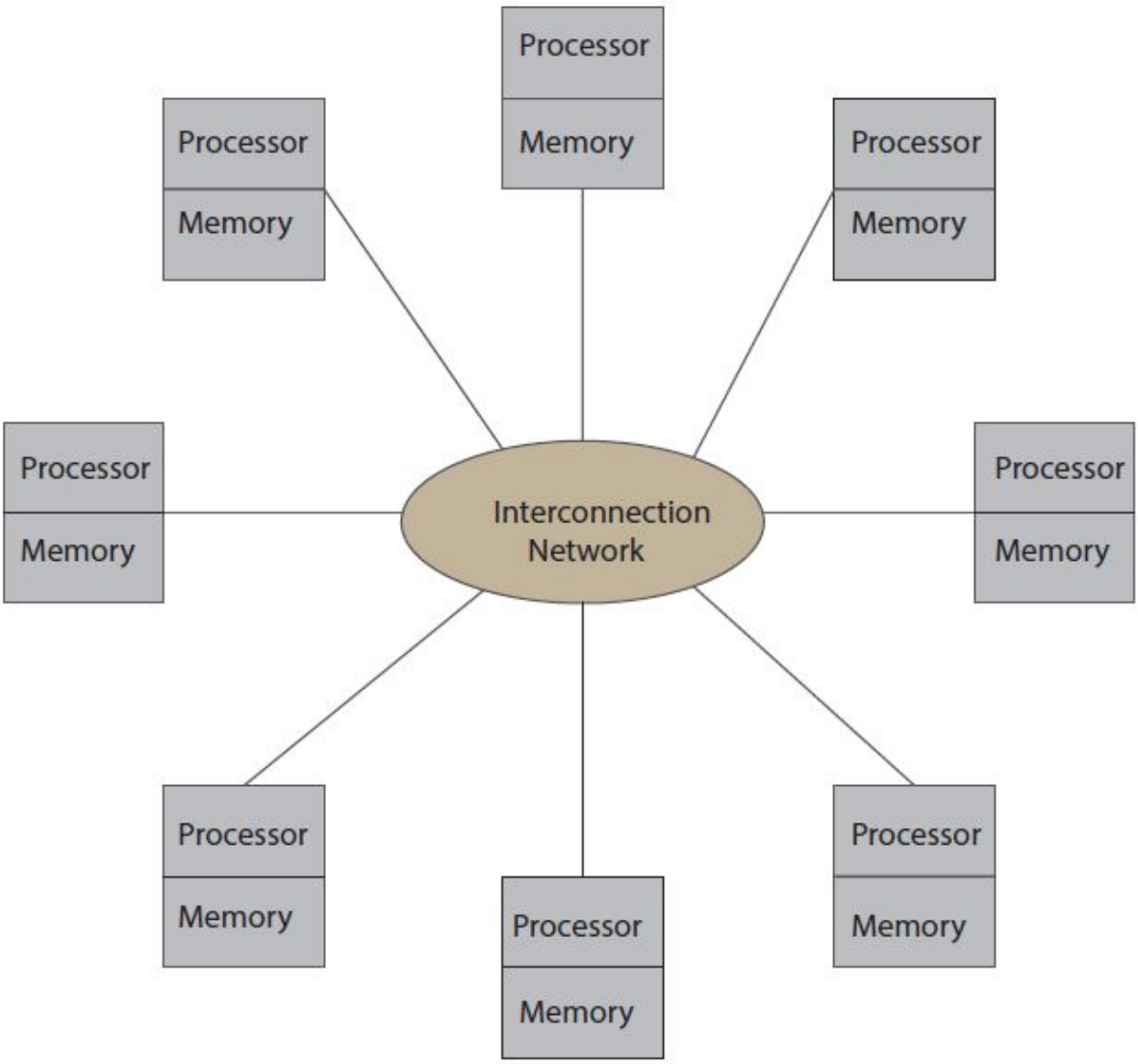

Figure 4.5 : Message Passing Model [49]
This model is used in distributed memory computers or cluster systems for communication. Each node of a cluster or computer of distributed memory system communicate with each other through processes.

4.3.3.1 MPJ Express Library

Java is proving itself to be a successful platform for all kind of simulations. Its success can be accredited to its good performance, portable characteristic, and innate support for security, objects, threads, visualization. [66] introduced a message passing interface called MPJ Express, which is pure java message passing interface. MPJ Express provides support for parallelizing extensive Java simulations on distributed and multicore platforms [10], a middleware that provides a channel for communication between processors of a cluster or multicore system.

MPJ Express supports a programming model which is Single Program Multiple Data (SPMD). Though MPJ Express is intended for distributed memory environment but it is possible to effectively execute parallel user applications on shared memory or multicore systems.

Configuration There are 2 main configurations of MPJ Express, as shown in Figure 4.6 .

- Multicore configuration $\longrightarrow$ for running MPJ Express programs on laptops/ desktops
- Cluster configuration $\longrightarrow$ for running MPJ Express programs on clusters and network of computers. The cluster configuration makes use of communication devices. MPJ Express supports 4 devices
  - Java New I/O (NIO) device, niodev: niodev for clusters using Ethernet
  - Myrinet device, mxdev: mxdev for clusters using Myrinet express interconnects
  - Hybrid device, hybdev: hybdev for clusters of multicore computers
  - Native device, native: native above MPI native library (Open MPI, MPICH, or MS-MPI)

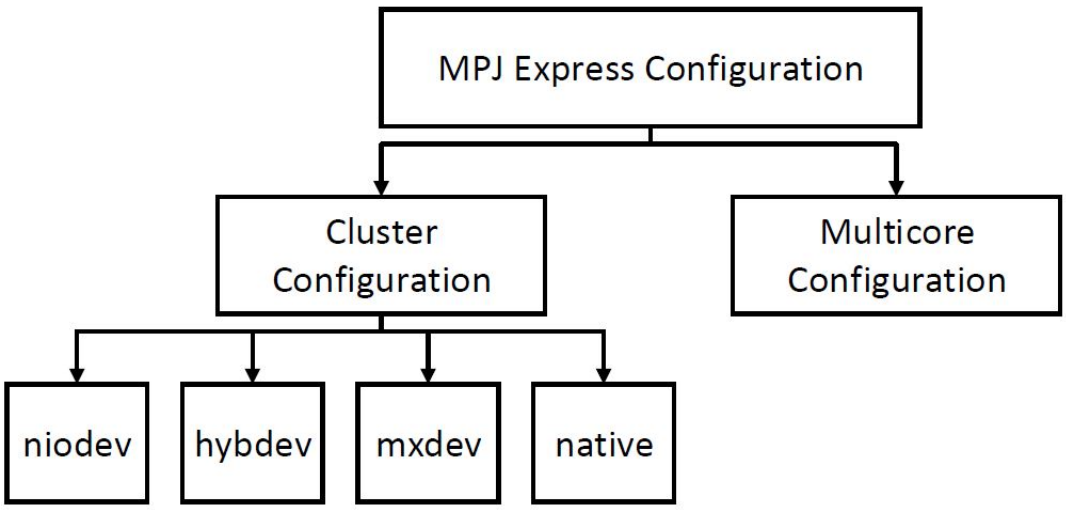

Figure 4.6 : MPJ Express Configurations [2]
MPJ Express works in multicore and cluster configurations. Cluster configuration supports 4 devices; niodev, mxdev, hybdev and native.

Multicore Configuration Multicore configuration is intended for the devices that have share memory and multi processors. Users can create their MPJ Express message passing parallel application and port it on multi processors. It is recommended to first write MPJ Express application for multicore and then use the same code for cluster configuration without modification. This configuration is suitable for teaching purposes. Each MPI process is a single thread. The multicore communication device uses effective inter thread mechanism.

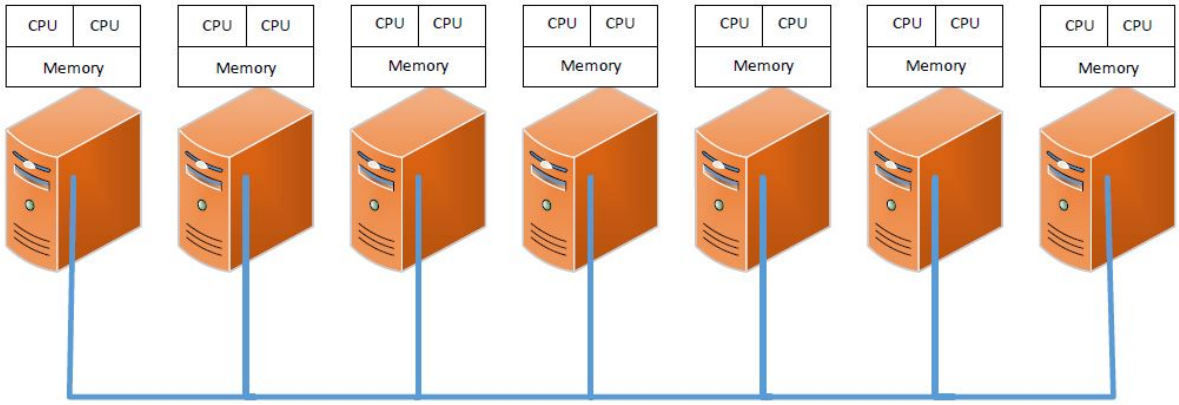

Figure 4.7 : MPJ Express Hybrid (Multicore + Cluster) Configuration [2]
Cluster consists of standalone multicore systems. All cores or nodes communicate through their respective MPJ Express process.

Cluster Configuration Cluster configuration is meant for distributed memory environment. In Figure 4.8 , there are 6 compute nodes interconnected via private network. MPJ Express will start one MPJ Express process on each node. They will communicate with each other using message passing. Each node of modern HPC clusters is mostly equipped with multicore processors. Hybrid device transparently uses both multicore and cluster configuration for intra and node communication as shown in Figure 4.7 .

4.4 Motivation

- AES is an ideal candidate for parallel implementation [21].

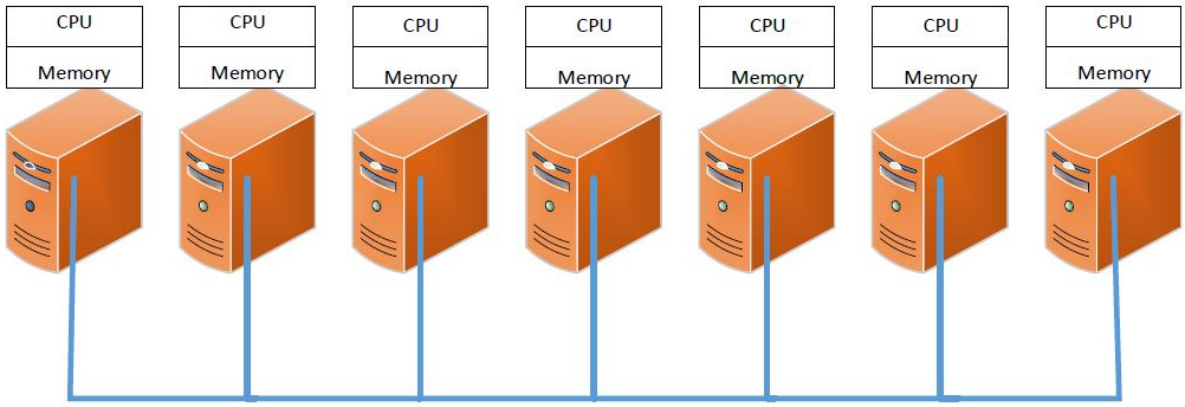

Figure 4.8 : MPJ Express Cluster Configuration [2]
Cluster system consists of multiple nodes or standalone systems. All nodes communicate through their respective MPJ Express process.

- Despite the fact that term *Standard* refers to applications of United States government only, AES is also required in numerous industrial standards and is utilized in several commercial applications and systems. Commercial standards that incorporate AES include Transport Layer Security (TLS), Skype, Internet Protocol Security (IPsec), SSH (Secure Shell), IEEE 802.11i the Wi-Fi encryption standard and many other security products around the world to date. [61]

- AES is computationally intensive and open source. Its decoding is still a challenge for the world. Only possible solution is brute force which is highly infeasible.

- Reason for preferring MPJ Express on other Java messaging passing libraries is due to the fact that MPJ Express performs considerably well in contrast to other Java MPIs including MPJ/Ibis and mpiJava [68].

- MPJ Express performed well in cosmological simulations that are massively parallel codes [11].

- Parallel implementation of AES using MPJ Express is of very preliminary nature in literature [25].

4.5 Methodology

Each MPJ Express process that runs on a single node (computer system) initialize its own instance of program and create its own threads consequently. Each process reads data from a file and divides it among its threads. Threads encrypt data using Java Cryptographic library and return encrypted data written in a file. The Java Cryptography Extension (JCE) provides security features in Java. It is an application program interface and offers a uniform framework. Figure 4.9 is high level view of proposed methodology.

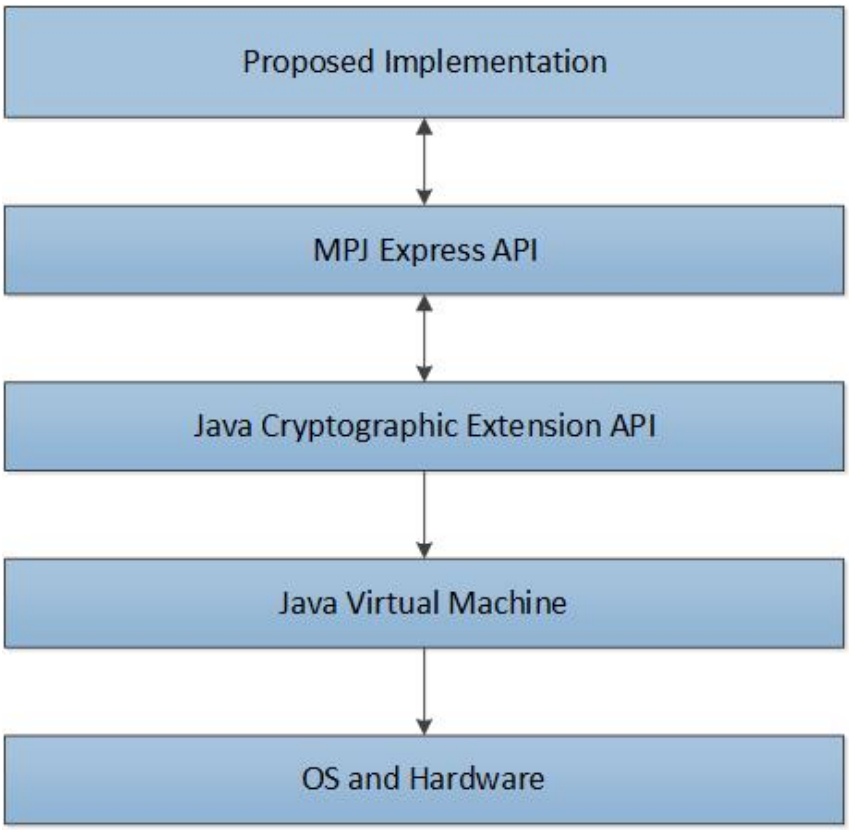

Figure 4.9 : Proposed Methodology
High level view of proposed methodology that shows the working of algorithm on computer system.

4.5.1 Algorithm

The AES algorithm has been parallelized on HPC platform using MPJ Express, to cut down the encryption time. Figure 4.10 explains proposed parallel implementation of AES. Paralleliza-

tion is done using High Level data parallelism. High Level data parallelism refers to dividing data and then applying same operations on divided data in parallel.

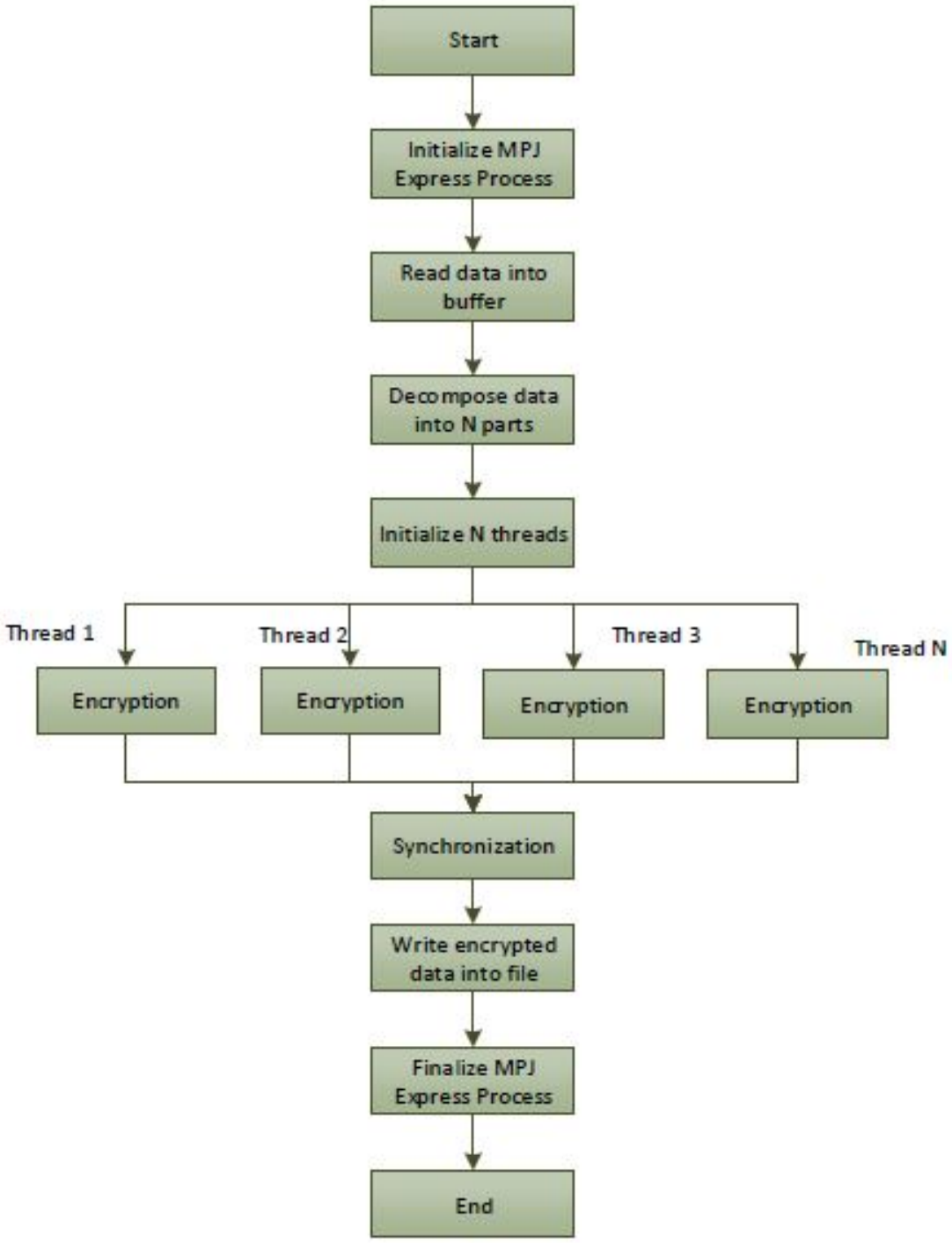

Figure 4.10 : Working of Algorithm
High level block diagram of algorithm that shows working of algorithm.

Initialize MPJ Express process(es).

1. User provides number of process(es) (np). This number remains constant throughout the program. Each process has unique ID and reads corresponding data (*task*) from the input file (input file is divided into np blocks). File size must not be very small. Encrypting a few bytes of file with large number of thread and processes adds thread overhead to total encryption time. Encrypting that file serially would be more economical.

2. For each process, data is further divided into n blocks (*fine grained*), where n is the number of threads you want to create in a process for encryption. In short, each block from np is further divided into n blocks.

3. Assign each block to each thread (*scheduling*) and start encryption.

4. Each thread encrypts its corresponding data and waits (*synchronization*) for other threads to complete encryption before returning it to master thread.

5. When encryption is complete by one process it waits for other processes to complete encryption (*synchronization*) before returning to master process.

6. When encryption is complete, each process writes encrypted data into a file one by one.

7. When file is written, MPJ Express processes return resources to operating system.

8. Number of total created processes and threads must not be greater than (#processors in the system * 2), if hyper threading is enabled.

$$np \leq \#processors * 2$$

Each physical/logical core can handle one thread at a time (one clock cycle). Operating system handles the thread and process assignment to cores. Overhead is added to total execution time, if number of threads become greater than 2 x number of cores.

4.5.1.1 Pseudo Code

```
MPI.Initialize();/* initialize MPJ Express Process */
Size=MPI_Total_Processes();/* Total number of processes created */
Rank=MPI_Current_Process();/* Current process running the program */
Buffer=Read(input_file);/* Read input file in a buffer */
if(Rank==0){/* Rank 0 is reponsible for sending data to other processes */
int bytes=file_size/Size;/* Number of bytes to be assigned to each process for encryption */
Buffer_Process[Rank]=Copy(Buffer,(Rank*bytes):((Rank*bytes)+bytes);  /* Copy bytes form "Buffer" and send to each process */
for(i=1;i<Size;i++){
Send(i,Buffer_Process[i]);}
MPI.COMM_WORLD.Barrier();/* Wait until all data is send */
write(encrypt(Buffer_Process[0]),Total_Threads,output_file));/* After sending, Rank 0 encrypts its data and write to output file */
}
else{
Recv(Rank,Buffer_Process[Rank]);/* Processes other than Rank 0 receive their data */
MPI.COMM_WORLD.Barrier();/* Wait untill all processes receive their data */
write(encrpyt(Buffer_Process[Rank]),Total_Threads,output_file);/* Encrypt data and write to output file */}
MPI.Finalize();/* return resources to OS */
```

4.6 Summary

In this chapter, proposed methodology is discussed along with the key concepts of parallel computing, multicore and cluster system. MPJ Express is the programming model used for parallelizing AES. Results are discussed in next chapter.

Chapter 5

Results

5.1 Introduction

AES is implemented using proposed methodology. Specifications of the system that is used for implementation is mentioned in section 5.2 and Performance Parameters in Section 5.3. Remaining chapter is divided into 3 parts. Section 5.4 is Parallel AES in Java using MPJ Express, Section 5.5 Parallel AES in C using OpenMP and MPICH, Section 5.6 Parallel AES in C using CUDA. We compared and discussed the results in this chapter.

5.2 System Specification

Cluster consists of 31 nodes and each node is equipped with 2x2.27 GHz 64bit Intel 4-core Xeon E5520 processors with total of eight physical cores (sixteen logical cores with hyper threading) having 24 Giga Bytes(GB) DDR3 RAM and Linux platform along with NVidia Tesla S1070 GPU having

- 4 x 1 Tesla T10 Processors
- CUDA Driver Version/ Runtime Version: 6.5

- CUDA Capability Version: 1.3
- 4 x 240 CUDA Cores
- GPU Clock Rate of 1.44 GHz

5.3 Performance Parameters

Performance parameters are:

- Execution Time $\longrightarrow$ Total encryption time. It does not include communication, decomposition or synchronization time.
- Throughput $\longrightarrow$ Number of bytes encrypted per second

$$Throughtput = T = \frac{Data\ Size}{Execution\ time\ using\ a\ multiprocessor\ with\ p\ processors} \quad (1)$$

- Speed up $\longrightarrow$ Ratio of serial encryption time to parallel encryption time

$$Speed\ up\ = S = \frac{Execution\ time\ using\ one\ processor\ (sequential\ algorithm)}{Execution\ time\ using\ a\ multiprocessor\ with\ p\ processors}$$
$$S = \frac{t_s}{t_p} \quad (2)$$

- Efficiency $\longrightarrow$ Ratio of speed up to total number of processors

$$Efficiency = E = \frac{Speed\ up\ factor}{No.\ of\ processors} = \frac{S}{p} = \frac{t_s}{p * t_p} \quad (3)$$

Note: p=16 as we have maximum 16 logical cores at one node.

Performance parameters are calculated by using formula (1), (2) and (3) for multicore platform while formula (1) and (2) for cluster platform .

5.4 Parallel AES in Java using MPJ Express

AES is implemented with Java cryptographic framework using Java inbuilt threads and MPJ Express.

5.4.1 Multicore Platform

Multicore platform is one node of the cluster, where we have 16 logical cores. We have created variable threads (1,2,4,8,16) and encrypted each file (size in Mega Bytes (MBs)) with these threads. As mentioned in Chapter 4, $np \leq \#processors * 2$. So, maximum 16 threads are created to get maximum system utilization. Results of these experiments are given in Table 5.1 and 5.2 .

Table 5.1: Execution Time (milli seconds (ms))of MPJ Express based AES on Multicore Platform for Different File Sizes and Threads

No. of Threads	**File Size**			
	107MBs	**207MBs**	**654MBs**	**1080MBs**
1	1835ms	3670ms	10987ms	18615ms
2	927ms	1853ms	5497ms	9099ms
4	500ms	912ms	2761ms	4557ms
8	260ms	489ms	1395ms	2294ms
16	136ms	255ms	729ms	1165ms

Table 5.2: Speed up and % Efficiency of MPJ Express based AES on Multicore Platform for Different File Sizes

File Size	**107MBs**	**207MBs**	**654MBs**	**1080MBs**
Speed up	13.49	14.39	15.07	15.97
% Efficiency	84.32	89.95	94.19	99.86

It is clear from the Table 5.1, by increasing number of threads execution time decreases. After a certain limit when number of threads become equal to number of cores, further increasing threads adds up time to total execution time. Execution time depends on file size as well. Very smaller file size takes more execution time in parallel encryption process. As we have discussed in Chapter 4, if granularity of the task is extremely fine-grained, the cost of mapping tasks to processors and their scheduling is enormous and yields a substantial total execution time. Increasing number of threads and decreasing data size for constant number of threads both increase granularity of the task. From Table 5.2, Speed up and % efficiency is less for small file and increases by increasing file size. But there is a limit to which they can be increased. It depends on the configuration and load on the system at which encryption is being performed. Figure 5.1 and 5.2 are illustrating execution time and throughput respectively for all file sizes and threads.

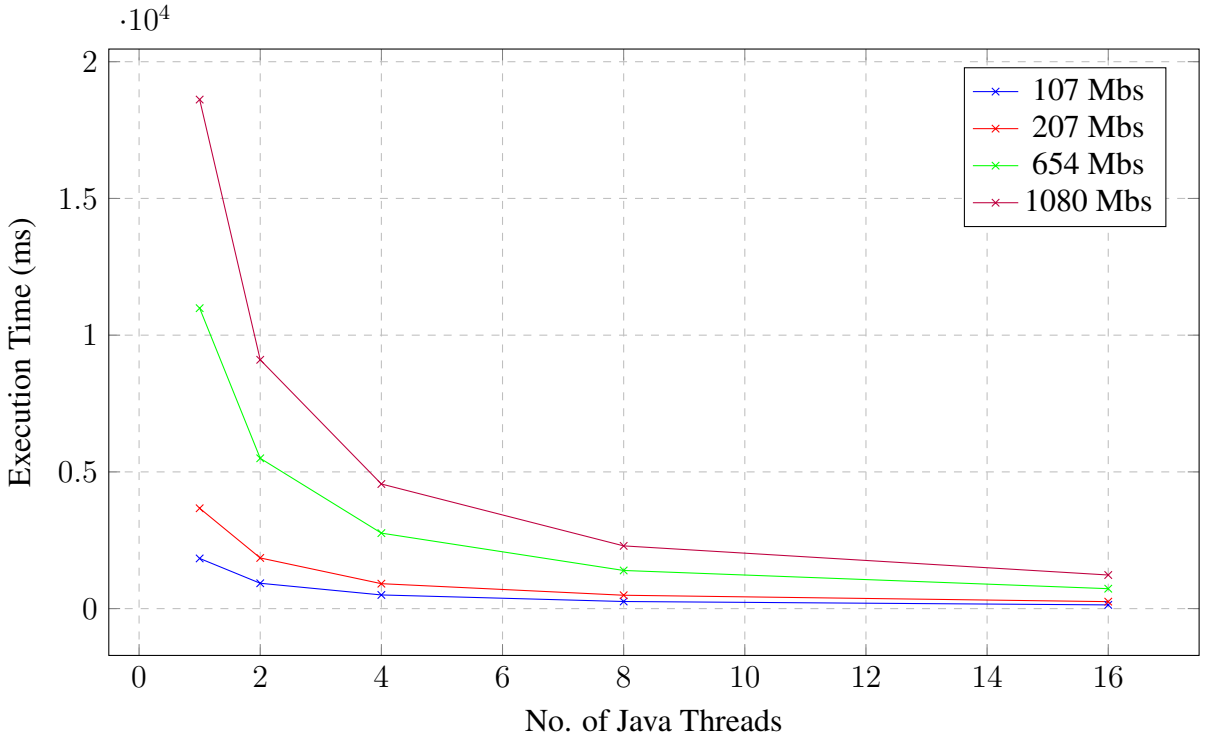

Figure 5.1 : Graph showing Execution Time of AES Encryption for Multiple File Sizes on Multicore Platform

From Figure 5.1 , all file sizes are following the same trend. We have achieved scalability. Execution time taken by 2 threads is almost half of 1 thread. Time of 4 threads is almost half

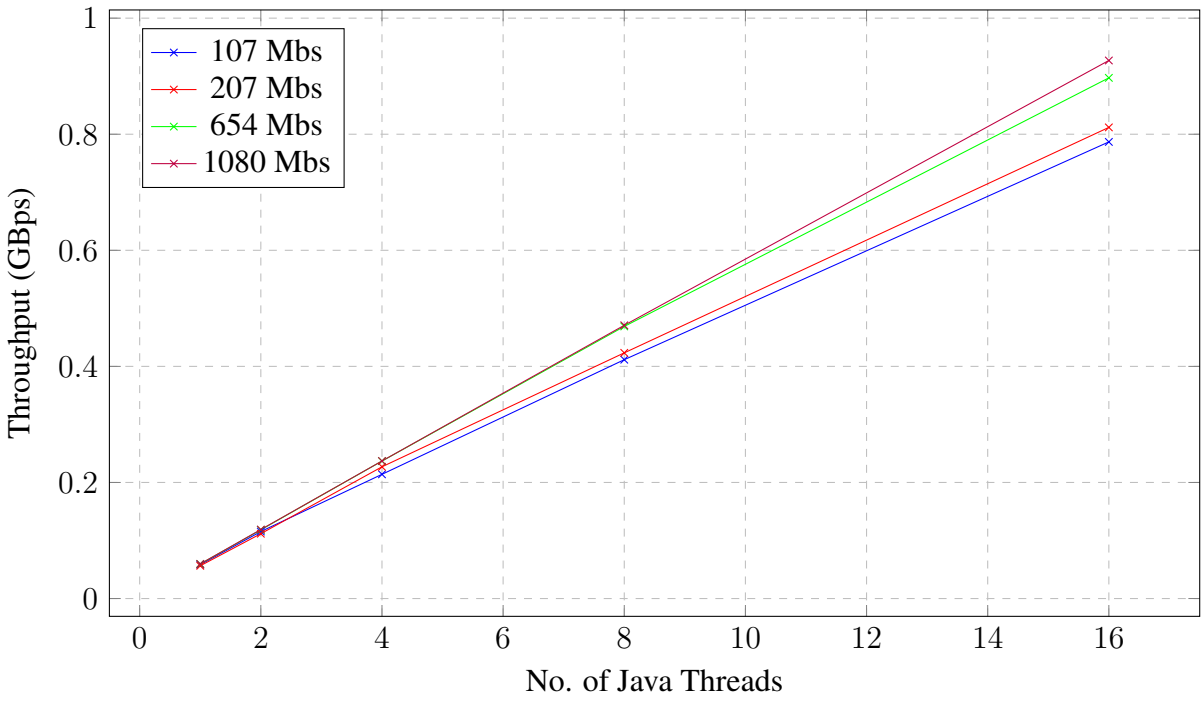

Figure 5.2 : Graph showing Throughput of AES Encryption for Different File Sizes on Multicore Platform

of 2 threads and so on. Execution time of 107 MBs and 207 MBs files are very close to each other because their file size difference is very small. Execution time difference between file sizes is large in the beginning but it decreases as number of threads increase, it is because of the fact that small file sizes face threads overhead but as file size increases the overheads become minimal thus execution time decreases at a good rate. From Table 5.1, we can see that scalability for small file size is less and it increases as file size increases. Increasing number of threads increases the throughput as shown in Figure 5.2 . Biggest file has the highest throughput. Throughput difference of file sizes for small number of threads is less but this difference increases since number of threads are increasing as more number of bytes are being encrypted in less time.

5.4.2 Cluster Platform

Our cluster platform comprises of 31 nodes while each node has 16 logical processors. The experiments are performed on various set of nodes. The results are stated in Table 5.3 and 5.4.

Table 5.3: Execution time for MPJ Express based AES Encryption of Different File Sizes for Cluster Platform

No. of Processes	**File Size**			
	107MBs	**207MBs**	**654MBs**	**1080MBs**
1	138ms	254ms	731ms	1200ms
4	52ms	98ms	197ms	315ms
8	34ms	49ms	109ms	165ms
12	29ms	44ms	85ms	119ms
16	23ms	34ms	79ms	110ms
20	18ms	32ms	75ms	100ms
24	17ms	29ms	70ms	92ms
28	16ms	25ms	64ms	88ms
31	15ms	23ms	59ms	82ms

Table 5.4: Speed up of MPJ Express based AES on Cluster Platform for Different File Sizes

File Size	**107MBs**	**207MBs**	**654MBs**	**1080MBs**
Speed up	122.33	159.56	186.22	227.01

Results of execution time and throughput are shown in Figure 5.3 and 5.4 respectively. From Table 5.3, result for 1 process is almost same as multicore 16 threads. Execution time decreases by increasing number of processes. Trend of scalability is same but execution time is not cutting down at good rate that is why throughput graph is not a straight line graph. We are creating large number of threads (Number of processes * 16) and file size is same which is causing overheads (extremely fine grained). 16 threads are created on each node, as each node has 16 logical processors to get maximum system utilization. Speed up increases by increasing file size.

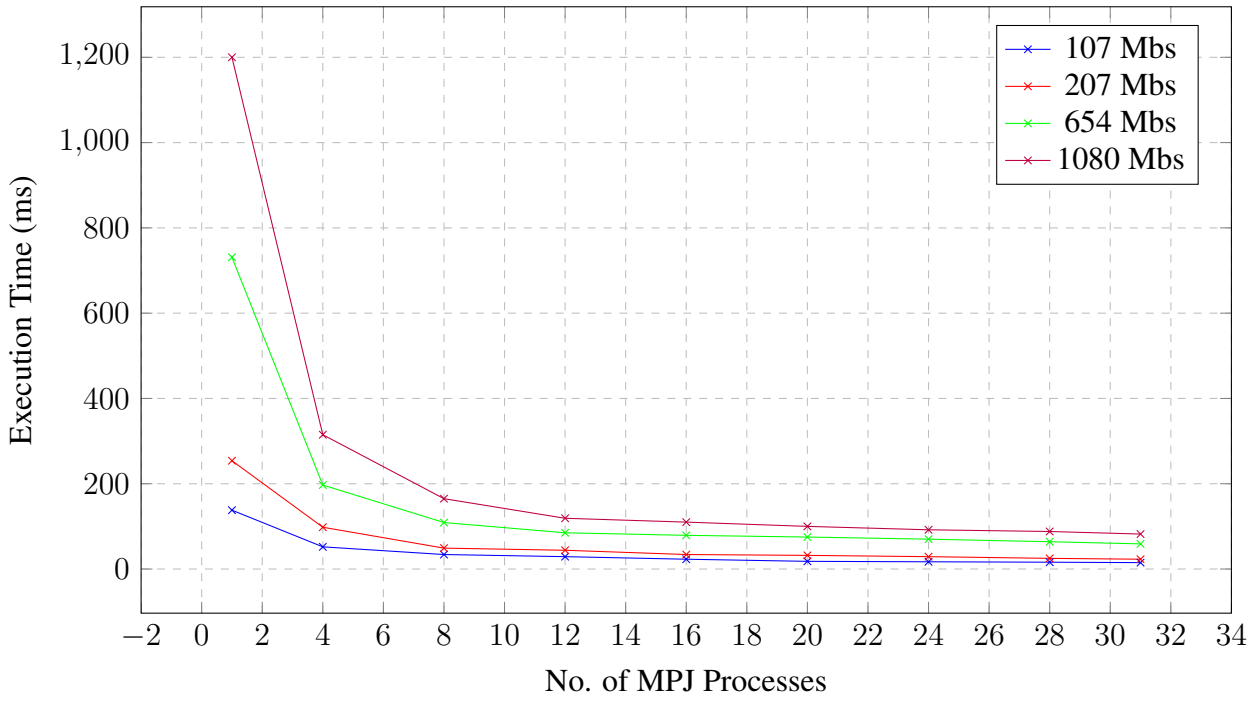

Figure 5.3 : Graph showing Execution Time of AES Encryption for Different File Sizes on Cluster Platform

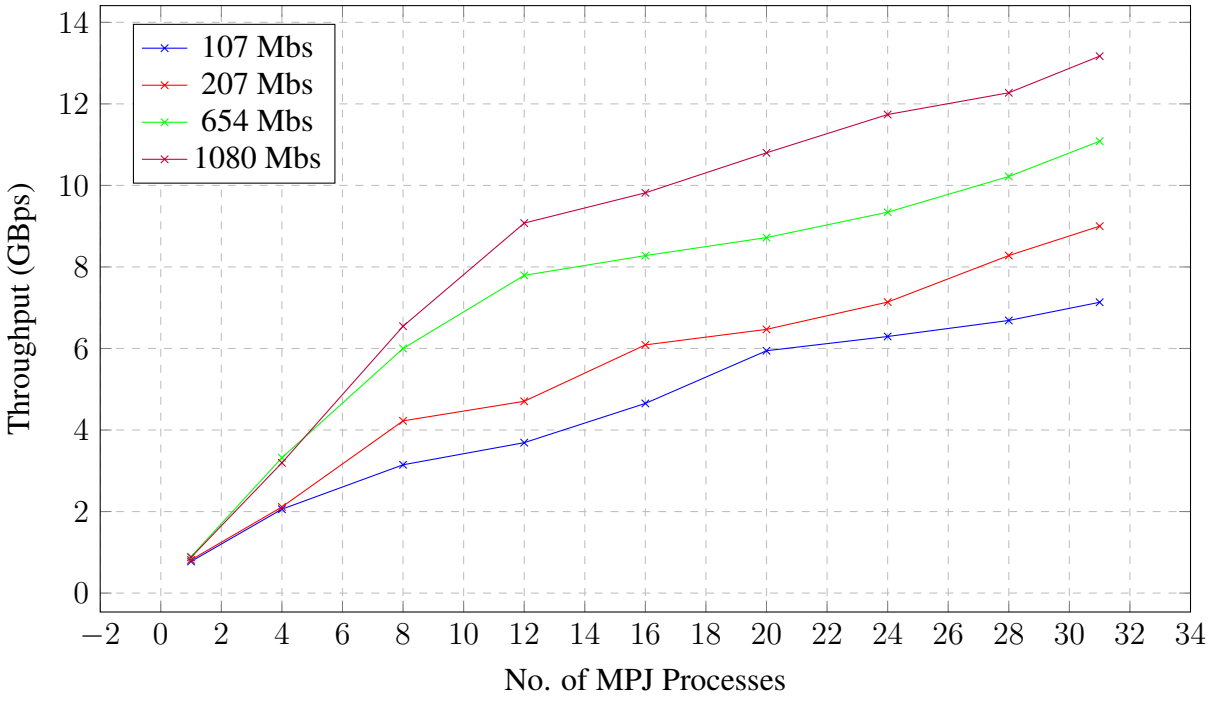

Figure 5.4 : Graph showing Throughput of AES Encryption for Multiple File Sizes on Cluster Platform

5.5 Parallel AES in C

The implementation results informed in this section evaluates serial and parallel implementation of algorithm. AES is implemented with OpenSSL cryptographic library using OpenMP threads

and MPICH as a message passing interface. Results are evaluated on multicore and cluster platform.

- MPICH is a broadly portable and high performance MPI standard (MPI-1, 2, 3). It proficiently supports different platforms including high speed networked clusters and proprietary high performance computer systems.
- OpenSSL is a general purpose cryptographic library
- OpenMP is an application program interface (API) that supports shared memory multi process programming on multi platform. It is portable and scalable that allows development of parallel applications from desktop to super computer.

5.5.1 Multicore Platform

We have 16 logical cores and created variable threads (1/2/4/8/16) for each file size. Multicore implementation of AES in C using OpenMP yielded better results as compared to multicore Java implementation using MPJ Express. Results are presented in Table 5.5 and 5.6.

Table 5.5: Execution time for OpenMP based AES Encryption of Different File Sizes for Multicore Platform

No. of Threads	**File Size**			
	107MBs	**207MBs**	**654MBs**	**1080MBs**
1	698ms	1330ms	4200ms	6880ms
2	380ms	708ms	2116ms	3370ms
4	190ms	359ms	1070ms	1750ms
8	140ms	268ms	800ms	902ms
16	117ms	218ms	660ms	798ms

Table 5.6: Speed up and % Efficiency of OpenMP based AES on Multicore Platform for Different File Sizes

File Size	107MBs	207MBs	654MBs	1080MBs
Speed up	5.96	6.10	6.36	8.62
% Efficiency	37.28	38.13	39.77	53.88

Execution time is far more less and throughput is far greater as compared to MPJ Express. Trend of both implementations is different. Execution time is less scalable as compared to MPJ Express implementation that resulted in less speed up and efficiency. Execution time for 107 MBs and 207 MBs files are very close as file size difference is small. Results are demonstrated graphically in Figure 5.5 and 5.6 .

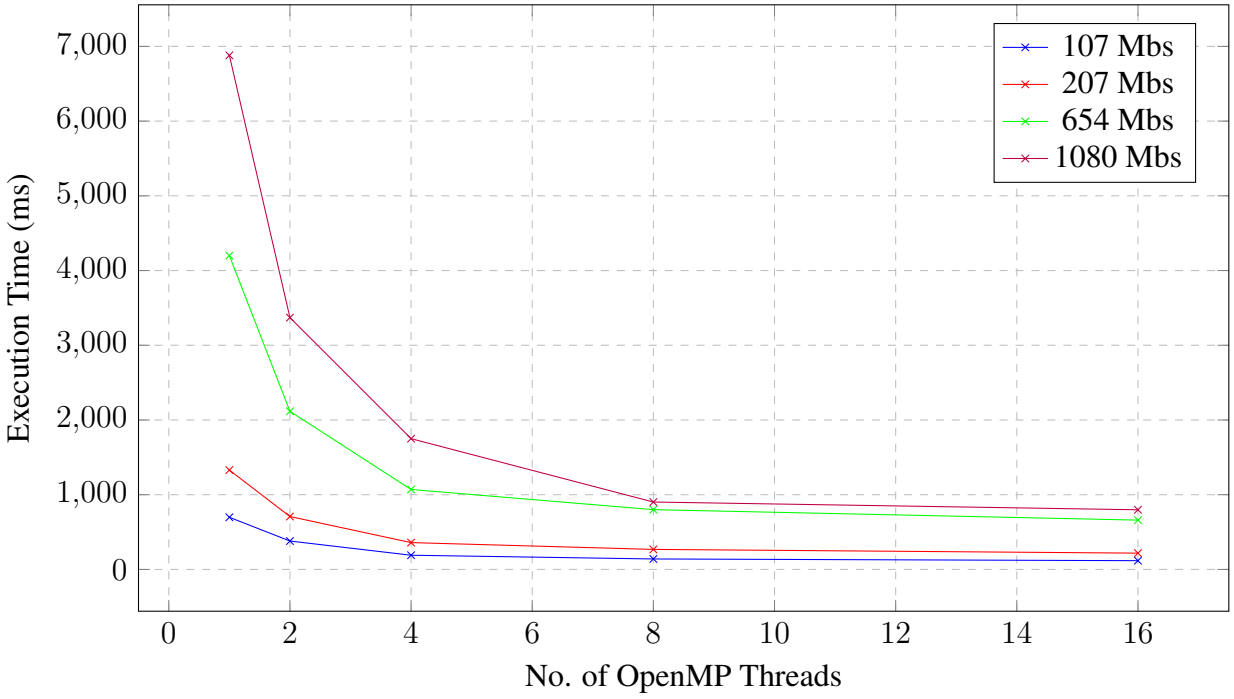

Figure 5.5 : Graph showing Execution Time of AES Encryption for Different File Sizes on Multicore Platform

Throughput is number of bytes encrypted per second; so scalability of execution time is directly affecting it which explains the behavior of throughput graph. Throughput is not a straight line graph; slope of 1080 MBs file is changing after 8 threads while for others slope is changing after 4 threads. This is because throughput is following the scalability trend of execution time.

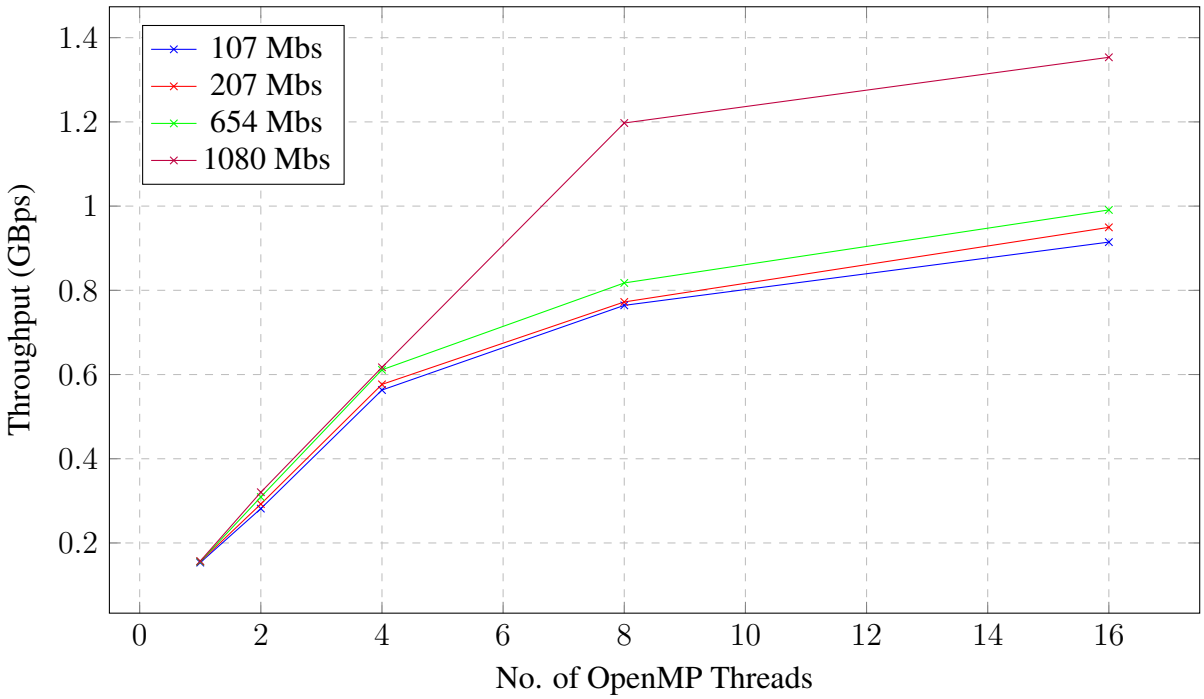

Figure 5.6 : Graph showing Throughput of AES Encryption for Different File Sizes on Multicore Platform

5.5.2 Cluster Platform

Cluster platform has 31 nodes and each node has 16 logical processors. Each file is encrypted for variable sets of nodes. Results are presented in Table 5.7 and 5.8. Execution Time is not showing scalability because thread overhead is increasing. Speed up increases as file size increases. Results are displayed in a graph format in Figure 5.7 and 5.8 . Throughput is not a straight line graph for the same reason. Slope is changing at every node.

5.6 Parallel AES in C using CUDA

The implementation results informed in this section evaluates the implemented algorithm. AES is implemented with OpenSSL cryptographic library using CUDA programming model.

- CUDA is a parallel programming model and computing platform developed by NVIDIA. It allows intense increase in performance of computer systems by utilizing the power of GPUs.

Table 5.7: Execution time for MPICH based AES Encryption of Different File Sizes for Cluster Platform

No. of Processes	File Size			
	107MBs	**207MBs**	**654MBs**	**1080MBs**
1	111.93ms	161.25ms	500.48ms	800.30ms
4	35.95ms	60.62ms	175.80ms	264.82ms
8	29.24ms	45.01ms	85.68ms	132.70ms
12	27.36ms	35.48ms	64.56ms	97.21ms
16	24.68ms	29.80ms	57.65ms	79.48ms
20	20.67ms	26.28ms	51.86ms	71.82ms
24	19.07ms	23.55ms	47.57ms	62.84ms
28	17.35ms	21.30ms	44.44ms	56.54ms
31	16.65ms	19.37ms	40.90ms	52.02ms

Table 5.8: Speed up for MPICH based AES Encryption of Different File Sizes for Cluster Platform

File Size	**107MBs**	**207MBs**	**654MBs**	**1080MBs**
Speed up	41.90	68.64	102.67	132.23

- Tesla architecture is created on a scalable processor array. NVidia Tesla S1070 GPU has 4 x 1 Tesla T10 Processors. Each T10 processor is composed of 10 independent texture processor clusters (TPCs) that contain 30 streaming multiprocessors (SMs) which are made up with 240 streaming processor (SP) Figure 5.9 , 5.10 . We will not discuss details of architecture.

Although GPUs are entirely programmable processors, but they are used in union with CPUs as they cannot execute stand-alone applications. CPU supervises GPU as it only does the heavy computation in most cases. CPU offloads the input data to GPU memory. GPU code that runs

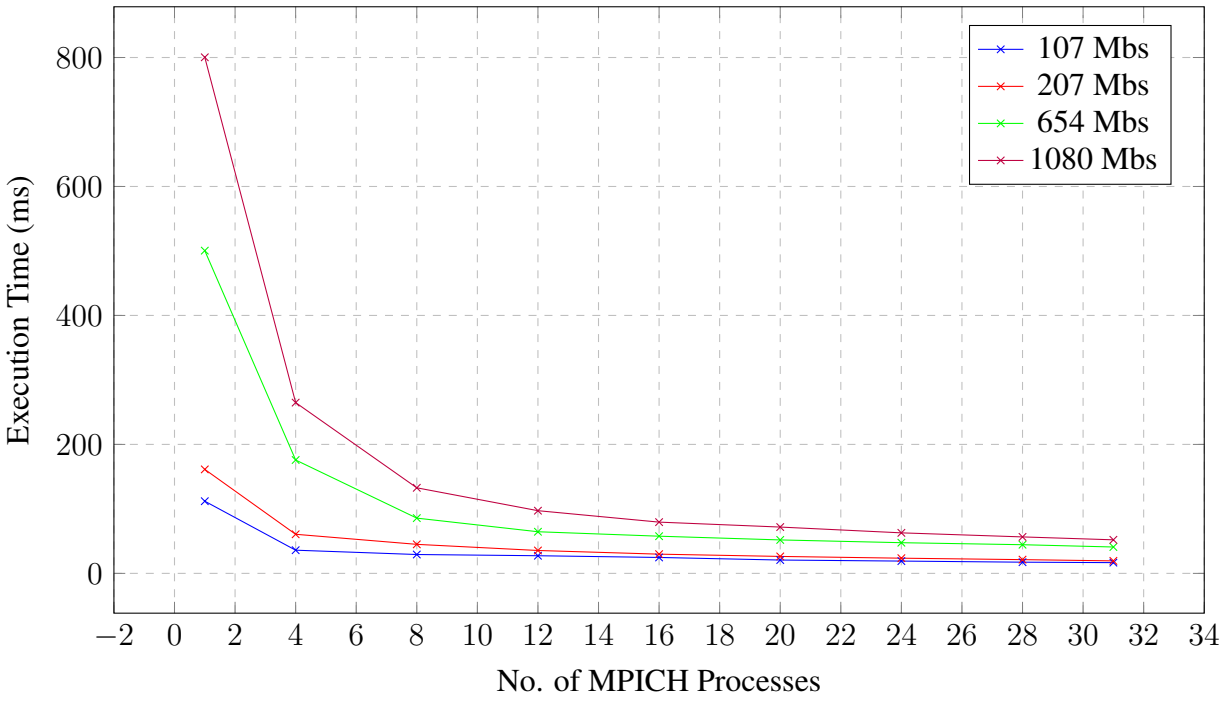

Figure 5.7 : Graph showing Execution Time of AES Encryption for Different File Sizes on Cluster Platform

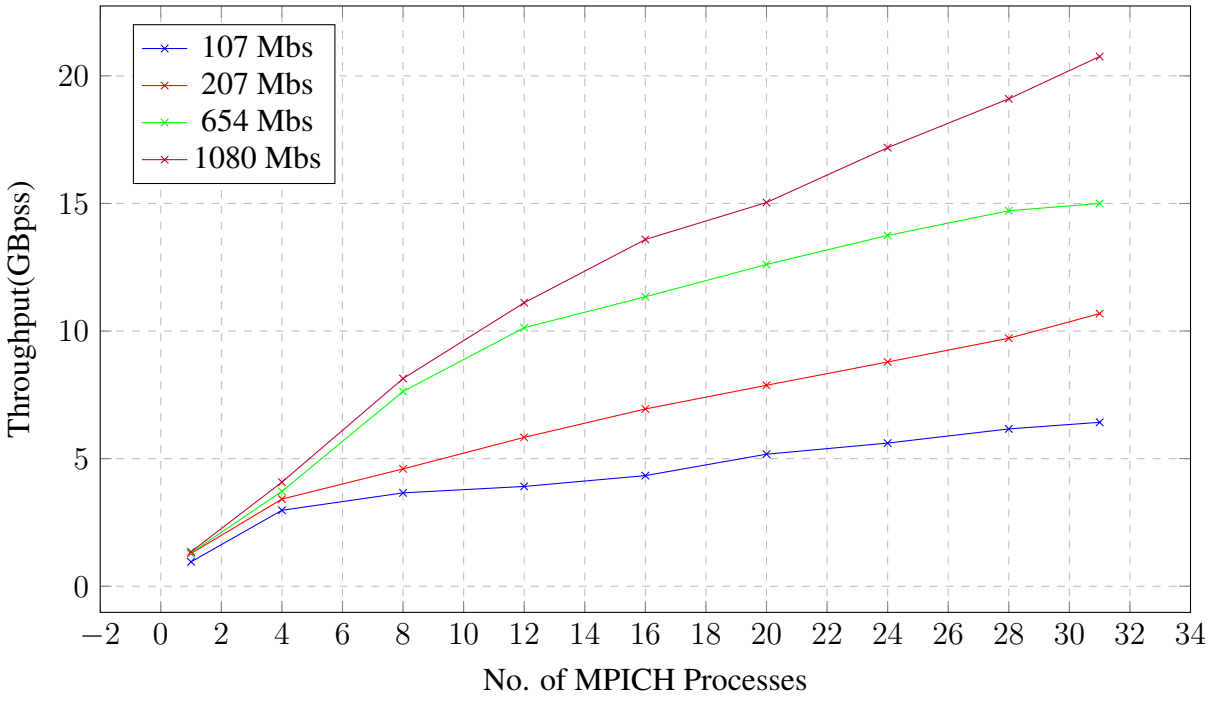

Figure 5.8 : Graph showing Throughput of AES Encryption for Multiple File Sizes on Cluster Platform

in parallel is called as kernel. Apparently kernel is a sequential code but thousands of parallel GPU threads run it. Hardware controls the in-dependency (unique identifier, control path, data) of these threads. CPU runs the main program and copies the data to GPU memory for parallel

processing that calls up GPU kernel and in return thousands of threads. A GPU kernel maintains its state that may include GPU-optimized data structures not shared with the CPU. After the computation is done, processed data is copied back to CPU memory. Each thread handles 128-bit block of data. Plaintext blocks are encrypted in parallel by numerous threads. Block size and number of threads are optimization factors that balances the compromise in replicating data to GPU memory [76]. To maintain maximum GPU utilization, latency is being hidden by launching more threads [18]. After encryption/decryption is done, data is copied back from GPU to CPU memory [70]. Flow chart of AES encryption on GPU is shown in Figure 5.11 .

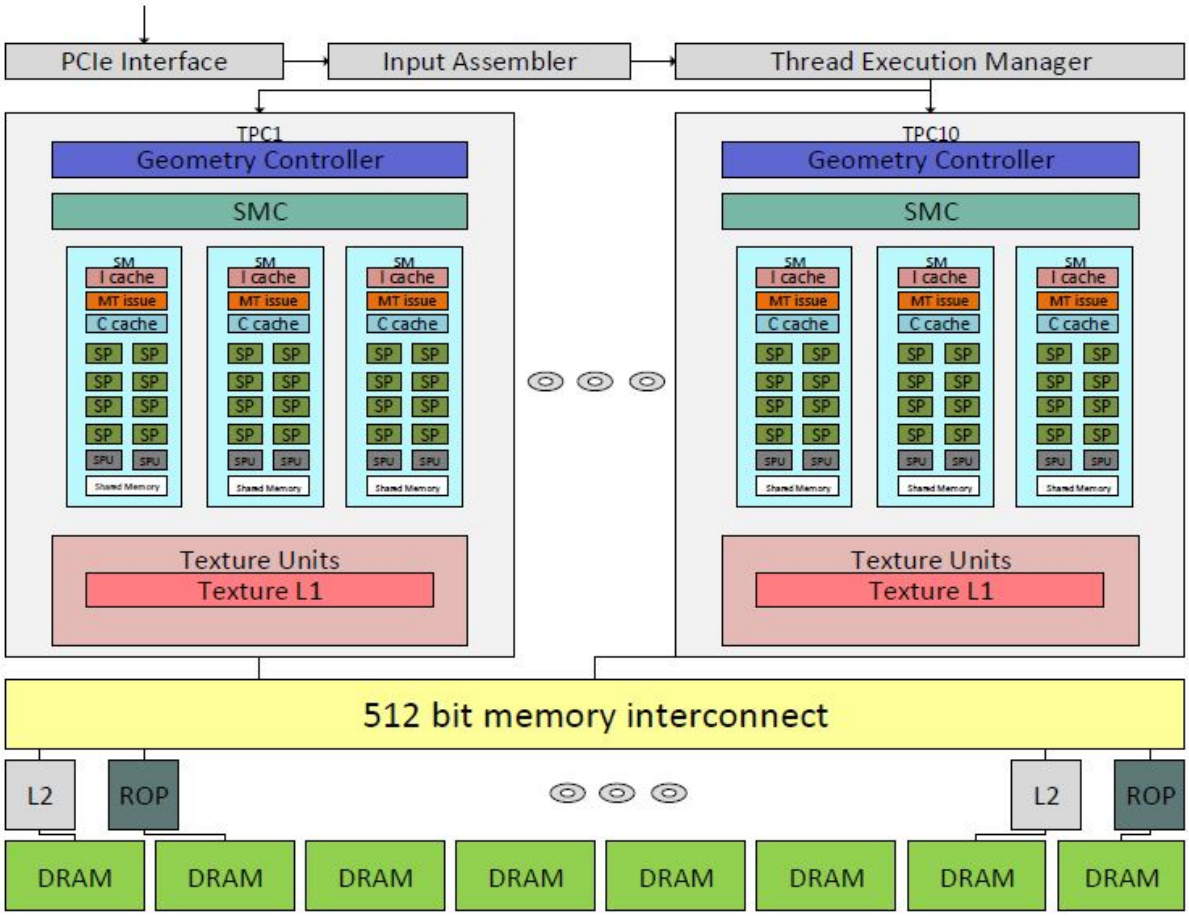

Figure 5.9 : Tesla T10 Architecture [46]
Tesla architecture is created on a scalable processor array. NVidia Tesla S1070 GPU has 4 x 1 Tesla T10 Processors. Each T10 processor is composed of 10 independent texture processor clusters (TPCs) that contain 30 streaming multiprocessors (SMs) which are made up with 240 streaming processor (SP)

There are two differences between CPU and GPU threads. [70]

1. GPU threads are lightweight and facilitate fine grained parallelism (1 operation/thread).

2. Number of cores/processors does not give the number of GPU threads that GPU can support at an instant. We can create large number of threads. Performance of GPU is best with large number of threads.

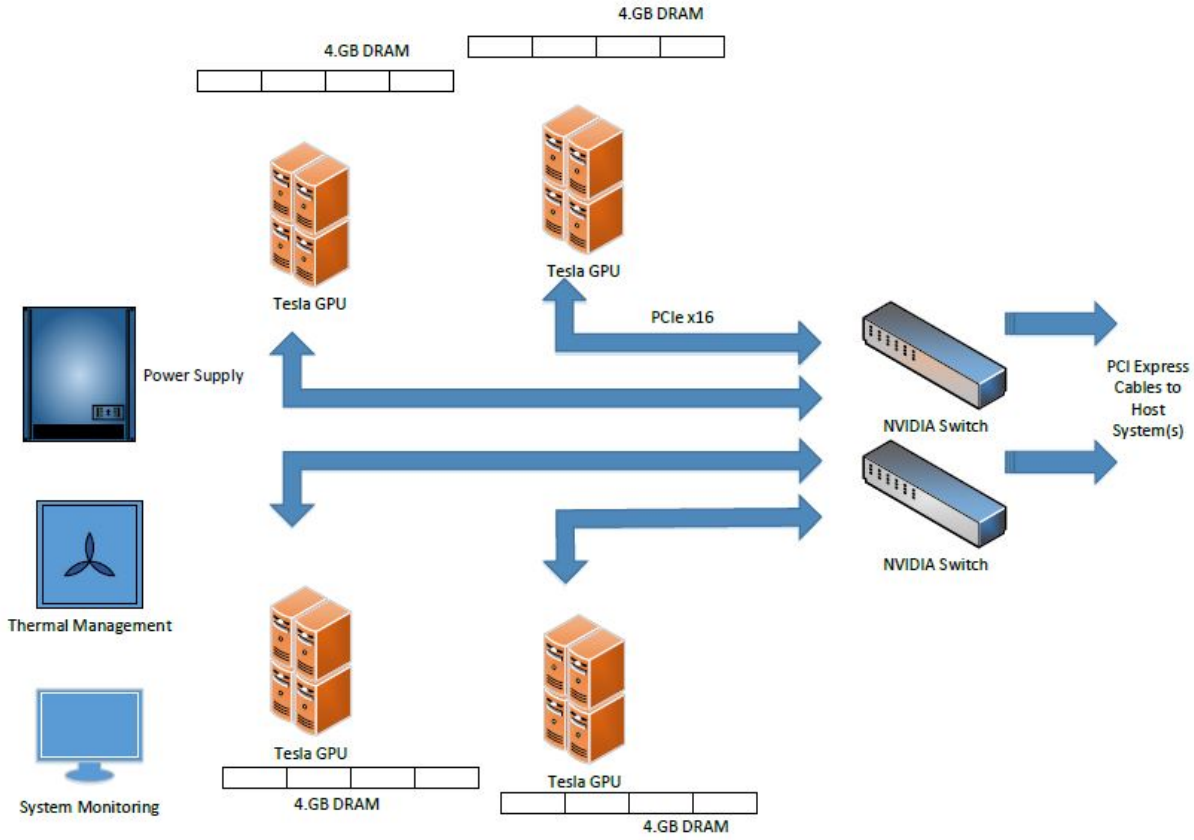

Figure 5.10 : High level view of Nvidia Tesla S1070 GPU [8]
S1070 GPU consists of four T10 Tesla processors each having 4 GB DRAM. They are connected to NVidia switch with PCIe x16 slots. GPU has its own thermal management and system monitoring unit.

Result for GPU implementation is presented in Table 5.9 and Figure 5.12 . The order in which number of threads is increasing is not uniform. This is because Table 5.9 was becoming very big and to reduce the size of table we have taken values that are showing significant time difference. That is why graph is not showing uniform increase in speed up factor. For uniform increase in number of threads, the graph would be a smooth curve. This is evident from these results that increasing number of threads is increasing speed up factor. But there is a limit to which speed

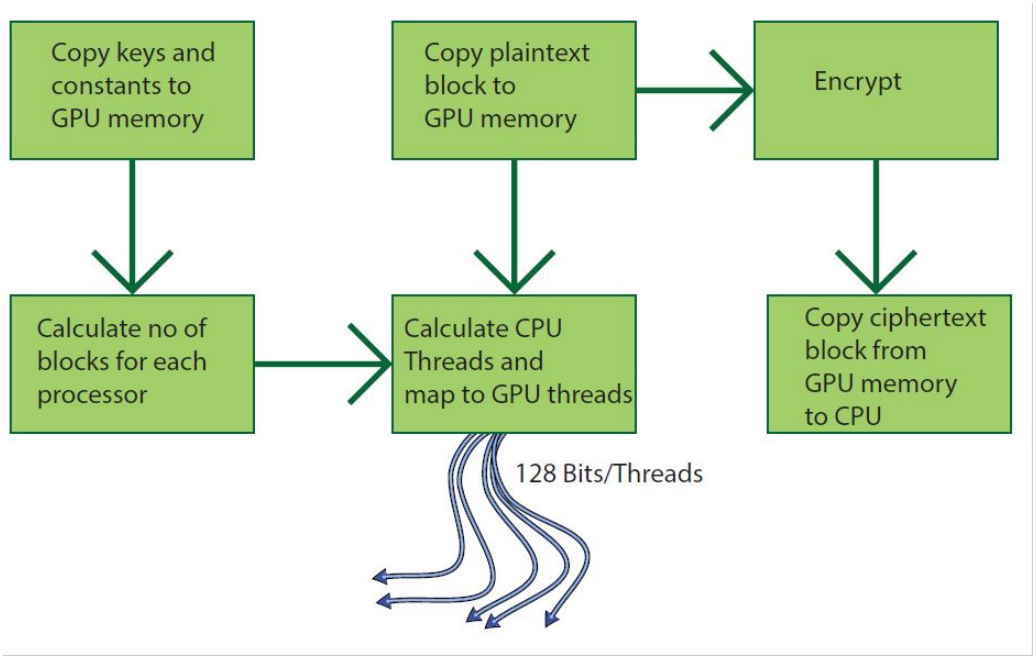

Figure 5.11 : GPU Implementation Flow Chart of AES Encryption
Keys, constants and plaintext blocks are copied to GPU memory. These blocks are divided among processors. CPU threads are created and mapped to GPU threads. Blocks are encrypted by respective processors and ciphertext blocks are copied from GPU to CPU memory.

up factor can increase after that it becomes constant. This limit depends on file size. The bigger the file size the higher is the limit and vice versa for small file size. It highly depends on the logic of the application along with configuration of the system. For example, number of active threads that a GPU can handle (occupancy) differs from one type of GPU to another (system configuration). AES is more suitable for parallel implementation than other ciphers (logic of application). The PCI Express interface used in our system is x16 that is further composed of two x8. It slows down process and thus speed up factor decreases (system configuration). The GPU implementation outperforms CPU implementations either on multicore or cluster platform. They are even better in computing cryptographic primitives, as they are able to attain better throughput specially due to coarse grain parallelism in some ciphers [76]. GPUs work better with larger data size, as increase in size refers to increase in number of threads working on large data, hence increase in computational proficiency concealing the latency encountered in copying data.

Table 5.9: Execution Time of CUDA based AES Encryption of Different File Sizes for GPU Platform

No. of GPU Threads	File Size			
	107MBs	**207MBs**	**654MBs**	**1080MBs**
1K	1505.86ms	3049.74ms	4782.92ms	11865.01ms
10K	116.69ms	212.43ms	520.06ms	826.04ms
50K	56.24ms	83.99ms	142.75ms	209.48ms
100K	49.30ms	61.75ms	89.72ms	117.37ms
500K	42.22ms	46.57ms	53.20ms	61.06ms
1000K	40.47ms	42.24ms	46.12ms	56.46ms
5000K	40.60ms	40.89ms	41.81ms	53.72ms
10000K	41.34ms	41.11ms	40.04ms	47.68ms
20000K	41.31ms	41.07ms	40.52ms	40.95ms
50000K	41.43ms	40.55ms	40.75ms	40.22ms

Table 5.10: Comparison of Speed up of AES Encryption on all Platforms

Platform	107 MB	207 MB	654 MB	1080 MB
MPJ Multicore (1 node)	13.49	14.39	15.07	15.97
MPJ Cluster (31 nodes)	122.33	159.56	286.22	277.01
MPICH Multicore (1 node)	5.96	6.10	6.36	8.62
MPICH Cluster (31 nodes)	41.90	68.64	102.67	132.33
CUDA (1 node)	17.24	103.36	178.60	178.60

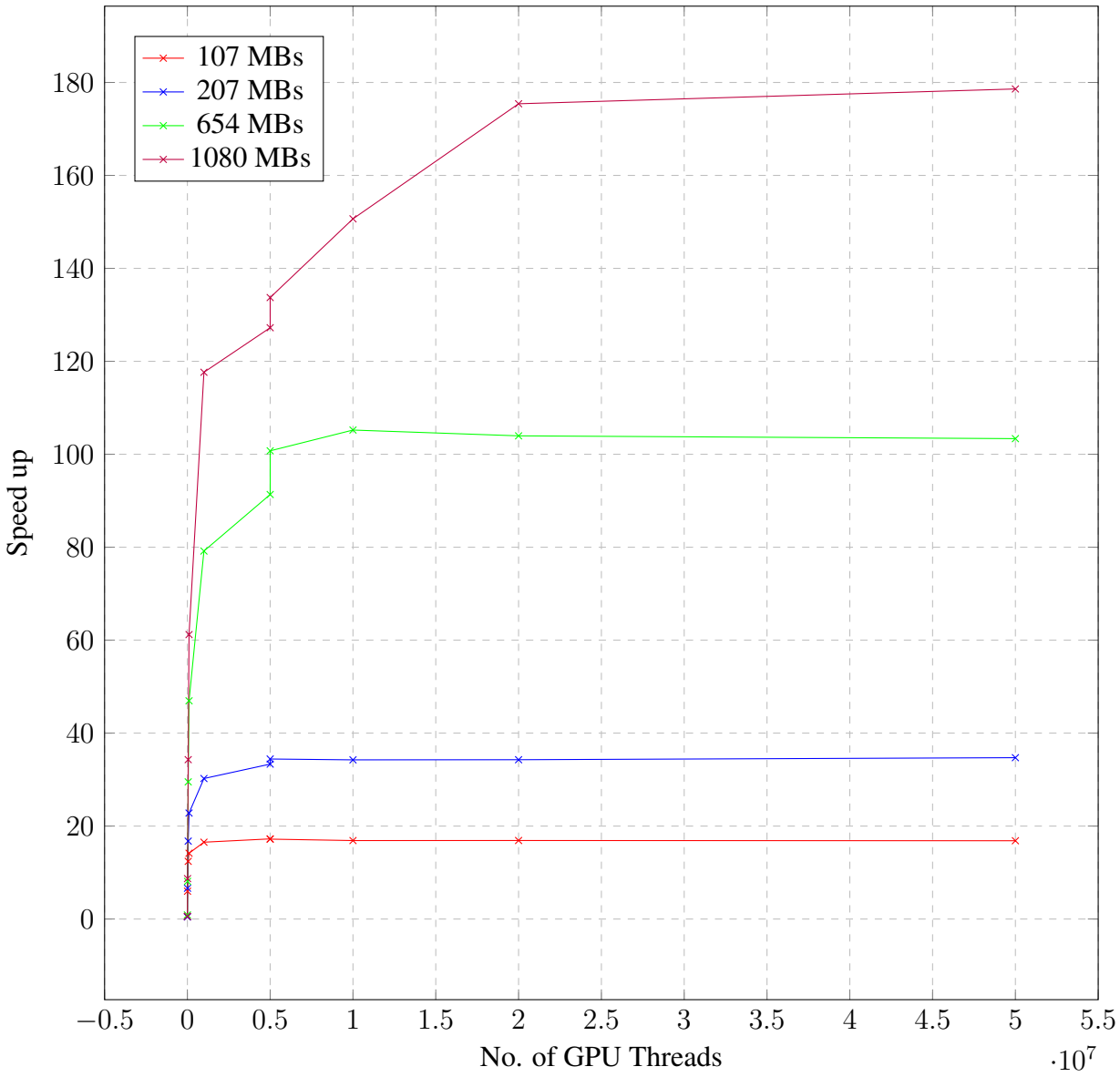

Figure 5.12 : Graph showing Speed up of CUDA based AES Encryption of Different File Sizes for GPU Platform

5.7 Summary

In this chapter implementation of proposed methodology is discussed. A comparative evaluation is presented using tables and graphs. In this work, we used MPJ-Express to accelerate Advanced Encryption Standard (AES) algorithm. We have partitioned the problem at two levels. By employing a data parallel approach, we divide the data length first among available processors and then data at each processor is further divided among processor cores. The experimental results show almost linear throughput in case of multicore platform (1 node or stand alone system) and non linear throughput for cluster platform. These experimental results are

compared with the AES algorithm accelerated by separately using other parallel programming tools in C language such as OpenMP API , MPICH and CUDA programming model. Parallel AES implementation using MPJ Express provides high speed up and efficiency for multicore and cluster platform as compared to AES accelerated in C using OpenMP on multicore platform and MPICH on cluster platform. But the speed up of GPU based implementation of AES in C using CUDA (1 node) is higher as compared to AES in MPJ Express using multicore. Overall performance of AES accelerated in C on all platforms is best as compared to AES accelerated using MPJ Express.

Chapter 6

Conclusion

The current research is done to explore the parallel implementation of AES with MPJ Express on multicore and cluster platform and then compare its results with other implementations. Figure 6.1 shows multiple implementations of AES that we have used in this research.

Table 6.1: Multiple AES implementations used in this research

Hardware	**Programming Model**	**Platform**	**Programming Language**
CPU	MPJ Express	Multicore	Java
		Cluster	
	MPICH	Multicore	C
		Cluster	
GPU	CUDA	1 node of cluster	

A series of conclusions were made which are following:

- Performance of AES in C using MPICH and OpenMP is great as compared to AES in Java using MPJ Express.
- Although throughput and execution time of AES in C using MPICH and OpenMP is

superior than MPJ Express but speed up factor and efficiency of AES in Java using MPJ Express is better as compared to them.

- AES in Java using MPJ Express provides platform independent implementation.
- AES in C using CUDA (1 node) out performed AES in MPJ Express multicore implementations.
- To obtain optimal performance, user must find a compromise between number of threads and data size (granularity).
- It all depends on requirement of application while selecting platform for implementation. If performance is a priority then AES in C using MPICH, OpenMP and CUDA is the best. Platform independent application in MPJ Express costs performance.
- MPJ Express is a good choice for parallel programming developers as it has inherent qualities of Java e.g.
  - Automatic garbage collection
  - Inbuilt security
  - Run time error checking
  - Vast variety of available libraries
  - Built in support for threads
  - Memory safe
  - Frequently used for enterprise applications
  - Ease of programming due to high level programming concepts
  - Less lines of code and takes less time for development due to vast variety of available resources

6.1 Future Work

Based on the work done in this dissertation some recommendations for future research are as follows:

- Increasing granularity by parallelizing internal operations of AES and then applying data parallelism.
- Implement it on CPU (MPJ Express , MPICH) and compare its results with current research findings
- Implement it on GPU (CUDA , Java Compute Unified Device Architecture (JCUDA)) and compare its results with current research findings

Bibliography

[1] http://m.2cto.com/os/201607/528685.html.

[2] http://mpj-express.org/docs/guides/linuxguide.pdf.

[3] https://www.biomedcentral.com/content/figures/1471-2105-11-217-1-l.jpg.

[4] https://www.cs.rit.edu/ ark/251/module05/notes.shtml.

[5] https://www.ibm.com/support/knowledgecenter/en/ssb23s_1.1.0.13/gtps7/bulkcip.html.

[6] https://www.top500.org/lists/.

[7] http://www.admin-magazine.com/hpc/articles/building-an-hpc-cluster.

[8] http://www.nvidia.com/docs/io/43395/sp-04154-001_v02.pdf.

[9] Hassan Anwar, Masoud Daneshtalab, Masoumeh Ebrahimi, Juha Plosila, Hannu Tenhunen, Sergei Dytckov, and Giovanni Beltrame. "Parameterized AES-based crypto processor for FPGAs". In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 465–472. IEEE, 2014.

[10] Mark Baker, Bryan Carpenter, and Aamir Shafi. "MPJ: A New Look at MPI for Java". In *Proc. of the UK e-Science All Hands Meeting*. Citeseer, 2005.

[11] Mark Baker, Bryan Carpenter, and Aamir Shafi. "MPJ Express meets Gadget: towards a Java code for cosmological simulations". In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 358–365. Springer, 2006.

[12] T Balamurugan and T Hemalatha. "Parallelization of Symmetric and Asymmetric security Algorithms for MultiCore Architectures".

[13] J Saira Banu, M Vanitha, J Vaideeswaran, and S Subha. "Loop parallelization and pipelining implementation of AES algorithm using OpenMP and FPGA". In *Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013 International Conference on*, pages 481–485. IEEE, 2013.

[14] Gerassimos Barlas, Ahmed Hassan, and Yasser Al Jundi. "An analytical approach to the design of parallel block cipher encryption/decryption: A CPU/GPU case study". In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 247–251. IEEE, 2011.

[15] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. "Design of a parallel AES for graphics hardware using the CUDA framework". In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8. IEEE Computer Society, 2009.

[16] Holger Blume, Jörg von Livonius, Lisa Rotenberg, Tobias G Noll, Harald Bothe, and Jörg Brakensiek. "OpenMP-based parallelization on an MPCore multiprocessor platform–A performance and power analysis". *Journal of Systems Architecture*, 54(11):1019–1029, 2008.

[17] Joppe W Bos, Dag Arne Osvik, and Deian Stefan. "Fast Implementations of AES on Various Platforms.". *IACR Cryptology ePrint Archive*, 2009:501, 2009.

[18] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. "Guided region-based GPU scheduling: utilizing multi-thread parallelism to hide memory latency". In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 441–451. IEEE, 2013.

[19] Cristian Chitu, David Chien, Charles Chien, Ingrid Verbauwhede, and Frank Chang. "A hardware implementation in FPGA of the Rijndael algorithm". In *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, volume 1, pages I–507. Ieee, 2002.

[20] Paweł Chodowiec and Kris Gaj. "Very compact FPGA implementation of the AES algorithm". In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 319–333. Springer, 2003.

[21] Joan Daemen and Vincent Rijmen. *"The design of Rijndael: AES-the advanced encryption standard"*. Springer Science & Business Media, 2013.

[22] Max Domeika. *Software development for embedded multi-core systems: a practical guide using embedded Intel architecture*. Newnes, 2011.

[23] Jack J Dongarra, L Grandinetti, J Kowalik, and GR Joubert. *"High performance computing: technology, methods and applications"*, volume 10. Elsevier, 1995.

[24] Cristina-Loredana Duta, Gicu Michiu, Silviu Stoica, and Laura Gheorghe. "Accelerating encryption algorithms using parallelism". In *Control Systems and Computer Science (CSCS), 2013 19th International Conference on*, pages 549–554. IEEE, 2013.

[25] Fadi El-Faleet. "A HIGH PERFORMANCE ENHANCED SEQUENTIAL AND PARALLEL AES, MS Thesis,Islamic University of Gaza, Deanery of Higher Studies Faculty of Engineering Computer Engineering Department", 2011.

[26] Ghada F Elkabbany, Heba K Aslan, and Mohamed N Rasslan. "A Design of a Fast Parallel-Pipelined Implementation of AES: Advanced Encryption Standard". *arXiv preprint arXiv:1501.01427*, 2015.

[27] Chih-Peng Fan and Jun-Kui Hwang. "Implementations of high throughput sequential and fully pipelined AES processors on FPGA". In *Intelligent Signal Processing and Communication Systems, 2007. ISPACS 2007. International Symposium on*, pages 353–356. IEEE, 2007.

[28] Reza Rezaeian Farashahi, Bahram Rashidi, and Sayed Masoud Sayedi. "FPGA based fast and high-throughput 2-slow retiming 128-bit AES encryption algorithm". *Microelectronics journal*, 45(8):1014–1025, 2014.

[29] Xiongwei Fei, Kenli Li, Wangdong Yang, and Keqin Li. "Practical parallel AES algorithms on cloud for massive users and their performance evaluation". *Concurrency and Computation: Practice and Experience*, 2015.

[30] Xiongwei Fei, Kenli Li, Wangdong Yang, and Keqin Li. "A secure and efficient file protecting system based on SHA3 and parallel AES". *Parallel Computing*, 52:106–132, 2016.

[31] Viktor Fischer and Miloš Drutarovskỳ. "Two methods of Rijndael implementation in reconfigurable hardware". In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 77–92. Springer, 2001.

[32] Kris Gaj and Pawel Chodowiec. "Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays". *Topics in CryptologyâĂŤCT-RSA 2001*, pages 84–99, 2001.

[33] Artur Gielata, Pawel Russek, and Kazimierz Wiatr. "AES hardware implementation in FPGA for algorithm acceleration purpose". In *Signals and Electronic Systems, 2008. ICSES'08. International Conference on*, pages 137–140. IEEE, 2008.

[34] José M Granado-Criado, Miguel A Vega-Rodríguez, Juan M Sánchez-Pérez, and Juan A Gómez-Pulido. "A new methodology to implement the AES algorithm using partial and dynamic reconfiguration". *INTEGRATION, the VLSI journal*, 43(1):72–80, 2010.

[35] Guang-liang Guo, Quan Qian, and Rui Zhang. "Different implementations of AES cryptographic algorithm". In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 1848–1853. IEEE, 2015.

[36] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. "Design and implementation of low-area and low-power AES encryption hardware core". In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 577–583. IEEE, 2006.

[37] Trang Hoang et al. "An efficient FPGA implementation of the Advanced Encryption Standard algorithm". In *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on*, pages 1–4. IEEE, 2012.

[38] Alireza Hodjat and Ingrid Verbauwhede. "A 21.54 Gbits/s fully pipelined AES processor on FPGA". In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 308–309. IEEE, 2004.

[39] Mary James and Deepa S Kumar. "An implementation of modified lightweight advanced encryption standard in FPGA". *Procedia Technology*, 25:582–589, 2016.

[40] Pradeeban Kathiravelu and Luis Veiga. "An adaptive distributed simulator for cloud and mapreduce algorithms and architectures". In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 79–88. IEEE, 2014.

[41] Louiza Khati, Nicky Mouha, and Damien Vergnaud. "Full Disk Encryption: Bridging Theory and Practice. In *Cryptographers' Track at the RSA Conference*, pages 241–257. Springer, 2017.

[42] Deen Kotturi, Seong-Moo Yoo, and John Blizzard. "AES crypto chip utilizing high-speed parallel pipelined architecture". In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4653–4656. IEEE, 2005.

[43] Deguang Le, Jinyi Chang, Xingdou Gou, Ankang Zhang, and Conglan Lu. "Parallel AES algorithm for fast data encryption on GPU". In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 6, pages V6–1. IEEE, 2010.

[44] Qinjian Li, Chengwen Zhong, Kaiyong Zhao, Xinxin Mei, and Xiaowen Chu. "Implementation and analysis of AES encryption on GPU". In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 843–848. IEEE, 2012.

[45] MC Liberatori and Juan Carlos Bonadero. "AES-128 cipher: Minimum area, low cost FPGA implementation". *Latin American applied research*, 37(1):71–77, 2007.

[46] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A unified graphics and computing architecture". *IEEE micro*, 28(2), 2008.

[47] Bin Liu and Bevan M Baas. "Parallel AES encryption engines for many-core processor arrays". *IEEE transactions on computers*, 62(3):536–547, 2013.

[48] Svetlin A Manavski. "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography". In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2007.

[49] J Quirm Michael. "Parallel Programming in C with MPI and OpenMP". *Dubuque, IA: McGraw-Hill*, 2004.

[50] Stefan Mocanu, Gabriel Munteanu, and Daniela Saru. "GPGPU optimized parallel implementation of AES using C++ AMP". *Journal of Control Engineering and Applied Informatics*, 17(2):73–81, 2015.

[51] M Nagendra and M Chandra Sekhar. "Performance improvement of Advanced Encryption Algorithm using parallel computation". *International Journal of Software Engineering and Its Applications*, 8(2):287–296, 2014.

[52] SS Navalgund, Akshay Desai, Krishna Ankalgi, and Harish Yamanur. "Parallelization of AES algorithm using OpenMP". *Lecture Notes on Information Theory Vol*, 1(4), 2013.

[53] TRAN Nhat-Phuong, LEE Myungho, HONG Sugwon, and LEE Seung-Jae. "High throughput parallelization of AES-CTR algorithm". *IEICE TRANSACTIONS on Information and Systems*, 96(8):1685–1695, 2013.

[54] Julian Ortega, Helmuth Trefftz, and Christian Trefftz. "Parallelizing AES on multicores and GPUs". In *Electro/Information Technology (EIT), 2011 IEEE International Conference on*, pages 1–5. IEEE, 2011.

[55] Soufiane Oukili and Seddik Bri. "Hardware Implementation of AES Algorithm with Logic S-box". *Journal of Circuits, Systems and Computers*, page 1750141, 2017.

[56] Vishal Pachori, Gunjan Ansari, and Neha Chaudhary. "Improved performance of advance encryption standard using parallel computing". *International Journal of Engineering Research and Applications (IJERA)*, 2(1):967–971, 2012.

[57] Dwiti Pandya, Khushboo Ram Narayan, Sneha Thakkar, Tanvi Madhekar, and BS Thakare. "Brief History of Encryption. *International Journal of Computer Applications*, 131(9):28–31, 2015.

[58] OZGUR PEKCAGLIYAN. "Parallel Implementation of AES algorithm using CUDA and MPI, MS Thesis,CANKAYA UNIVERSITY THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES COMPUTER ENGINEERING", September 2013.

[59] Vandan Pendli, Mokshitha Pathuri, Subhakar Yandrathi, and Abdul Razaque. "Improvising performance of Advanced Encryption Standard algorithm". In *Mobile and Secure Services (MobiSecServ), 2016 Second International Conference on*, pages 1–5. IEEE, 2016.

[60] Adrian Pousa, VICTORIA Sanz, and ARMANDO De Giusti. "Performance analysis of a symmetric cryptographic algorithm on multicore architectures". In *Computer Science & Technology Series-XVII Argentine Congress of Computer Science-Selected Papers. Edulp*, 2012.

[61] Bart Preneel, Christof Paar, and Jan Pelzl. *"Understanding Cryptography"*. Springer, 2014.

[62] Shanxin Qu, Guochu Shou, Yihong Hu, Zhigang Guo, and Zongjue Qian. "High throughput, pipelined implementation of AES on FPGA". In *Information Engineering and Electronic Commerce, 2009. IEEC'09. International Symposium on*, pages 542–545. IEEE, 2009.

[63] K Rahimunnisa, P Karthigaikumar, Soumiya Rasheed, J Jayakumar, and S SureshKumar. "FPGA implementation of AES algorithm for high throughput using folded parallel architecture". *Security and Communication Networks*, 7(11):2225–2236, 2014.

[64] Thomas Rauber and Gudula Rünger. *"Parallel programming: For multicore and cluster systems"*. Springer Science & Business Media, 2013.

[65] Gaël Rouvroy, F-X Standaert, J-J Quisquater, and J-D Legat. "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications". In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 583–587. IEEE, 2004.

[66] Aamir Shafi, Bryan Carpenter, and Mark Baker. "Nested parallelism for multi-core HPC systems using Java". *Journal of Parallel and Distributed Computing*, 69(6):532–545, 2009.

[67] Aamir Shafi and Jawad Manzoor. "Towards efficient shared memory communications in MPJ express". In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–7. IEEE, 2009.

[68] Aamir Shafi, Jawad Manzoor, Kamran Hameed, Bryan Carpenter, and Mark Baker. "Multicore-enabling the MPJ Express messaging library". In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 49–58. ACM, 2010.

[69] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *"Operating system concepts"*, volume 4. Addison-wesley Reading, 1998.

[70] Mark Silberstein. "GPUs: High-performance Accelerators for Parallel Applications: The multicore transformation (Ubiquity symposium)". *Ubiquity*, 2014(August):1, 2014.

[71] Mostafa I Soliman and Ghada Y Abozaid. "FPGA implementation and performance evaluation of a high throughput crypto coprocessor". *Journal of Parallel and Distributed Computing*, 71(8):1075–1084, 2011.

[72] Abolfazl Soltani and Saeed Sharifian. "An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA". *Microprocessors and Microsystems*, 39(7):480–493, 2015.

[73] Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. "Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs". In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 334–350. Springer, 2003.

[74] Nhat-Phuong Tran, Myungho Lee, Sugwon Hong, and Seung-Jae Lee. "High throughput parallelization of AES-CTR algorithm". *IEICE TRANSACTIONS on Information and Systems*, 96(8):1685–1695, 2013.

[75] Jason Van Dyken and José G Delgado-Frias. "FPGA schemes for minimizing the power-throughput trade-off in executing the Advanced Encryption Standard algorithm". *Journal of Systems Architecture*, 56(2):116–123, 2010.

[76] Vivek Venugopal and Devu Manikantan Shila. "High throughput implementations of cryptography algorithms on GPU and FPGA". In *Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, pages 723–727. IEEE, 2013.

[77] S-M Yoo, Deen Kotturi, DW Pan, and John Blizzard. "An AES crypto chip using a high-speed parallel pipelined architecture". *Microprocessors and Microsystems*, 29(7):317–326, 2005.

[78] Yuan Yu, Dennis Fetterly, Michael Isard, Ulfar Erlingsson, and Mihai Budiu. "General purpose distributed data parallel computing using a high level language", August 18 2015. US Patent 9,110,706.

[79] Shanghong Zhang, Zhongxi Xia, Rui Yuan, and Xiaoming Jiang. "Parallel computation of a dam-break flow model using OpenMP on a multi-core computer". *Journal of hydrology*, 512:126–133, 2014.

[80] Xinmiao Zhang and Keshab K Parhi. "High-speed VLSI architectures for the AES algorithm". *IEEE transactions on very large scale integration (VLSI) systems*, 12(9):957–967, 2004.

Appendix A

AES Encryption with Java using MPJ Express

Multicore and Cluster Platform

```
1  import java.io.*;
2  import java.nio.ByteBuffer;
3  import java.nio.channels.FileChannel;
4  import java.nio.file.Paths;
5  import java.security.Key;
6  import java.util.Arrays;
7  import java.util.concurrent.CyclicBarrier;
8  import javax.crypto.*;
9  import mpi.*;
10
11
12 public class  cluster_aes {
13 static int num;
```

```
14 static int rank;
15 static int check=0;
16 File out = new File("output.txt");
17 FileWriter fw = null;
18 static CyclicBarrier barrier;
19 static long startTime=0;
20 static AES aes=new AES();
21
22 private Runnable enc (byte[] input,Cipher cipher, int n) {
23 return new Runnable() {
24 public void run() {
25
26 try{
27
28 fw = new FileWriter(f,true);
29 // encryption function from class AES and writing in output file
30 fw.write(aes.encrypt(input,cipher));
31 barrier.await();
32 } // end try
33 catch (Exception e) {
34 // TODO Auto-generated catch block
35 e.printStackTrace();
36 } // end catch
37 } // end run
38 }; // Runnable
39 }  // end enc()
40
41 public static void main(String[] args) throws Exception {
42
43 MPI.Init(args);
```

```
44 rank=  MPI.COMM_WORLD.Rank();       // process number
45 int size=  MPI.COMM_WORLD.Size();  // no of total processes created
46 String data = System.getProperty("user.dir");
47 data=data.replace("\\", "/");
48 data=data.concat("/"+args[3]);     // Input file
49 // no of threads created in each process
50 num=Integer.parseInt(args[4]);
51 cluster_aes cluster=new cluster_aes();  // creating class object
52 byte[] sub_buf;
53 Key key = null;
54 File f = new File(data);
55 FileInputStream fin = new FileInputStream(f);
56
57 // Cyclic barrier action.
58 // When all threads reach barrier following action will be performed
59 barrier = new CyclicBarrier(num,
60 new Runnable() {
61 public void run() {
62 if(rank==0 && check==0){
63 float time=(float)(System.currentTimeMillis()-startTime);
64 check++;
65 System.out.print("Execution Time="+time+"ms");
66 }
67 }
68 }); // end barrier
69
70
71 try {
72 key = aes.password("12345678qwertyuipzxcvbnmamhetcbh");
73 }  // end try
```

```
74 catch (Exception e) {
75
76 e.printStackTrace();
77 } // end catch
78
79 int count=(int)f.length();  // No of bytes in input file
80 float temp=(float)count/(float)size;  // no of bytes to be assigned to ↩
      each file
81 int sub=(int)Math.floor(temp);
82  // If count is not divisible by no of total processes,
83                                 // we will get a fraction.
84 int diff=(int) ((temp*size)-(sub*size));
85 //  This statement will cater for those extra bytes.
86
87 if(rank==size-1)
88 {
89 sub_buf=new byte[sub+diff];  // assigning those
90                               //extra bytes to last process
91 fin.skip(rank*sub);
92 fin.read(sub_buf);          // each process will read designated bytes ↩
      from buffer
93 }
94 else
95 {
96 sub_buf=new byte[sub];
97 fin.skip(rank*sub);
98 fin.read(sub_buf);
99 }
100 // repeating same method of bytes
101 //division among threads in each process
```

```
102 temp=((float)sub_buf.length/(float)num);
103 int g0=(int)Math.floor(temp);
104 int t0=(int)(((temp)*num)-((g0)*num));
105 Cipher c=aes.initialization(key);          // initializing AES cipher
106 startTime= System.currentTimeMillis();     // strating timer
107  // creating input number of threads
108 for (int i = 0; i <(num); i++ )
109 {
110 if(i==num-1)
111 {
112 new Thread(cluster.enc(Arrays.copyOfRange
113                (sub_buf,i*(g0), ((i*(g0))+(g0+t0))),c,i)).start();
114 }  // end if
115 else
116 {
117 new Thread(cluster.enc( Arrays.copyOfRange
118                 (sub_buf,i*(g0), ((i*(g0))+(g0))),c,i)).start();
119 } // end else
120 }  // end for
121 MPI.Finalize();
122
123 } // end main
124 } // end class
```

Appendix B

AES Encryption with C

AES Encryption with C using OpenMP

```
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>


void handleErrors(void)
{
    ERR_print_errors_fp(stderr);
    abort();
}

```

```
17 typedef struct  {
18    unsigned char *plaintext;
19    int plaintext_len;
20    int ciphertext_len;
21    unsigned char key[32];
22    unsigned char iv[17];
23    unsigned char *ciphertext;
24    int flag;
25 } args_struct;
26
27
28
29 int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *↩
       key,
30 unsigned char *iv, unsigned char *ciphertext)
31 {
32    EVP_CIPHER_CTX *ctx;
33    int len;
34    int ciphertext_len;
35    float time_spent ;
36    float begin;
37    float end ;
38
39    // initialize cipher
40    if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors();
41    if(1 != EVP_EncryptInit_ex
42                  (ctx, EVP_aes_256_cbc(), NULL, key, iv))
43    handleErrors();
44    if(omp_get_thread_num()==0)
45    {
```

```
46      begin=omp_get_wtime();
47    }
48    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len,
49        plaintext, plaintext_len)) // start encrption
50    handleErrors();
51    ciphertext_len = len;
52    if(omp_get_thread_num()==0){
53       end=omp_get_wtime();
54
55    }
56    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
57                    handleErrors();  // finalize encryption
58    ciphertext_len += len;
59    EVP_CIPHER_CTX_free(ctx); // free memory
60    return ciphertext_len;
61 }
62
63 int main (void)
64 {
65    // No of thread for encrypting input file
66    const long total_threads=1/2/4/8/;
67    unsigned char *plaintext=NULL;
68    unsigned char **cipherbuf =NULL;
69    unsigned char **pBuffers =NULL;
70    long size;
71    int i,j;
72    FILE *fp = NULL;
73    FILE *f;
74    args_struct *a[total_threads];
75    long sub_size;
```

```
76  //256 bit encryption key
77  unsigned char key[32] = ↩
        {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1};
78  //136 bit initialization vector
79  unsigned char iv [17] = {0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7};
80  ERR_load_crypto_strings();
81  OpenSSL_add_all_algorithms();
82  OPENSSL_config(NULL);
83   // input file for encryption
84  fp = fopen("input_file.txt", "r");
85  if(fp==NULL)
86  {
87     printf("unable to open");
88  }
89  f = fopen("output.txt", "w+");
90  // Size of file
91  fseek (fp, 0, SEEK_END);
92  size = ftell (fp);
93  fseek (fp, 0, SEEK_SET);
94
95  pBuffers = (unsigned char**)calloc (total_threads,
96                                      sizeof(unsigned char *));
97  cipherbuf = (unsigned char**)calloc (total_threads,
98                                       sizeof(unsigned char *));
99  // no of bytes to be assigned to each thread
100 sub_size=floor((float)size/(float)total_threads);
101
102 fseek (fp, 0, SEEK_CUR);
103
104 for ( i = 0; i < total_threads; i++) {
```

```
105
106     if((i==(total_threads-1))&&
107                     (total_threads*sub_size)<size)
108     //If "size" is not divisible by total_threads,
109     we will get a fraction.
110     //  This statement will cater for those extra bytes.
111     {
112         sub_size=(size-(total_threads*sub_size))+sub_size;
113     }
114     pBuffers[i] = (unsigned char*)calloc
115                     (sub_size, sizeof(unsigned char*));
116     cipherbuf[i] = (unsigned char*)calloc
117                     (sub_size, sizeof(unsigned char*));
118     fread(pBuffers[i],sizeof
119                         (unsigned char) ,sub_size, fp);
120             // reading specified bytes from input file
121 }
122
123 for( j=0;j<total_threads;j++)
124 { // copying data in "struct a" fields for encryption
125
126     a[j]=(args_struct*)calloc(1,sizeof (args_struct));
127     if((j==(total_threads-1))&&
128           (total_threads*sub_size)<size)
129     {
130       sub_size=(size-(total_threads*sub_size))+sub_size;
131     }
132     a[j]->plaintext=(unsigned char*)
133                 calloc(sub_size,sizeof(unsigned char*));
134     a[j]->ciphertext=(unsigned char*)
```

```
135                 calloc(sub_size,sizeof(unsigned char*));
136       a[j]->plaintext_len=sub_size;
137       a[j]->flag=0;
138       memcpy(a[j]->iv,iv,17);
139       memcpy(a[j]->key,key,32);
140       memcpy(a[j]->plaintext,pBuffers[j],sub_size);
141    }
142    omp_set_num_threads(total_threads);  // initializing threads
143    #pragma omp parallel  // starting threads in parallel
144    {
145       a[omp_get_thread_num()]->ciphertext_len=
146       encrypt(a[omp_get_thread_num()]->plaintext,
147       a[omp_get_thread_num()]->plaintext_len,
148       a[omp_get_thread_num()]-> key,
149       a[omp_get_thread_num()]->iv,
150        a[omp_get_thread_num()]->ciphertext);
151       a[omp_get_thread_num()]->flag=1;
152       memcpy(cipherbuf[omp_get_thread_num()],
153       a[omp_get_thread_num()]->ciphertext,
154       a[omp_get_thread_num()]->ciphertext_len
155       fputs(cipherbuf[omp_get_thread_num()], f);
156       #pragma omp barrier
157    }
158    fclose(f);
159    EVP_cleanup();   // clean up process
160    ERR_free_strings();
161    time_spent = (end-begin)*1000;
162    printf("Total time=%f ms\n",time_spent)
163    return 0;
164 }
```

AES Encryption with C using MPICH

```
#include <stdio.h>
#include <mpi.h>
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

int rank;
int total_pro,z,k;

void handleErrors(void)
{
ERR_print_errors_fp(stderr);
abort();
}

typedef struct  {
unsigned char *plaintext;
int plaintext_len;
int ciphertext_len;
unsigned char key[32];
unsigned char iv[17];
```

```
28 unsigned char *ciphertext;
29 int flag;
30 } args_struct;
31
32
33 // Encryption Function
34 int encrypt(unsigned char *plaintext, int plaintext_len,
35 unsigned char *key,unsigned char *iv, unsigned char *ciphertext)
36 {
37
38 EVP_CIPHER_CTX *ctx;
39
40 int len;
41 float begin;
42 float time_spent ;
43 float end ;
44 int ciphertext_len;
45
46 if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors(); // initialize cipher
47 if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv))
48 handleErrors();
49 if(rank==0 && omp_get_thread_num()==0)
50 {
51 begin=omp_get_wtime();
52 }
53 if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext,
54                           plaintext_len)) // start encrption
55 handleErrors();
56 if(rank==0 && omp_get_thread_num()==0){
57 end=omp_get_wtime();
```

```
58
59 }
60
61 ciphertext_len = len;
62 if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
63 handleErrors(); // finalize encryption
64 ciphertext_len += len;
65 EVP_CIPHER_CTX_free(ctx); // free memory
66
67 return ciphertext_len;
68 }
69
70
71
72 int  main(int argc, char *argv[])
73 {
74
75 unsigned char **cipherbuf =NULL;
76 unsigned char *pBuffers =NULL;
77
78 FILE *fp = NULL;
79 FILE *f;
80 long size;
81 int i,j,y,m;
82 long sub_size;
83 long temp;
84 long subsub_size;
85 long total_threads;
86 args_struct **a;
87 int x[1];
```

```
88 //256 bit encryption key
89 unsigned char key[32] = ↩
       {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1}
90 //136 bit initialization vector
91 unsigned char iv [17] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7};
92 sum=0;
93 ERR_load_crypto_strings();
94 OpenSSL_add_all_algorithms();
95 OPENSSL_config(NULL);
96 fp = fopen("input_file.txt", "r");// input file for encryption
97 if(fp==NULL)
98 {
99 printf("unable to open");
100 }
101 f = fopen("output.txt", "w+");
102 MPI_Init(&argc, &argv); // initialize MPI process
103 MPI_Comm_rank(MPI_COMM_WORLD,&rank);  // process number
104 // no of total processes created
105 MPI_Comm_size(MPI_COMM_WORLD,&total_pro);
106 // Size of file
107 fseek (fp, 0, SEEK_END);
108 size = ftell (fp);
109 fseek (fp, 0, SEEK_SET);
110 // no of bytes to be assigned to each process
111 sub_size=floor((float)size/(float)total_pro);
112 // no of threads created in each process
113 total_threads = strtol(argv[1], NULL, 0);
114 a = (args_struct**)calloc(total_threads, sizeof(args_struct*));
115 if(rank==0)
116 {
```

```
117 // no of bytes to be assigned to each thread
118 sub_size=floor((float)size/(float)total_pro);
119 fseek (fp, 0, SEEK_CUR);
120 for ( i = (total_pro-1); i > -1 ; i--)
121 {
122 if((i==(0))&&(total_pro*sub_size)<size)
123 //If "size" is not divisible by total_pro, we will get a fraction.
124 //  This statement will cater for those extra bytes.
125 {
126 sub_size=(size-((total_pro-1)*sub_size));
127 }
128 x[0]=sub_size;
129 pBuffers = (unsigned char*)calloc(sub_size, sizeof(unsigned char*));
130 // reading specified bytes from input file
131 fread(pBuffers,sizeof (unsigned char) ,sub_size, fp);
132 if(i!=0) // sending those bytes to other processes
133 {
134 MPI_Send(&x,sizeof(x)/sizeof(x[0]),MPI_INT,i,i+100,MPI_COMM_WORLD);
135 MPI_Send(pBuffers,sub_size,MPI_CHAR,i,i,MPI_COMM_WORLD);
136
137 }
138 else
139 {
140 // when sending data to other processes is done.
141 //Process 0 starts encrypting its own data
142 // no of bytes to be assigned to each thread
143 subsub_size=floor((float)sub_size/(float)total_threads);
144 cipherbuf = (unsigned char**)calloc
145        (total_threads, sizeof(unsigned char*));
146 for(y=0;y<total_threads;y++)
```

```
147 {
148 a[y]=(args_struct*)calloc(1,sizeof (args_struct));
149 if((y==(total_threads-1))&&(total_threads*subsub_size)<sub_size)
150 {
151 temp=subsub_size;
152 subsub_size=(sub_size-(y*subsub_size));
153 a[y]->plaintext=(unsigned char*)calloc
154          (subsub_size,sizeof(unsigned char*));
155 a[y]->ciphertext=(unsigned char*)calloc
156         (subsub_size,sizeof(unsigned char*));
157 a[y]->plaintext_len=subsub_size;
158 a[y]->flag=0;
159 memcpy(a[y]->plaintext,pBuffers+(y*temp),subsub_size);
160 memcpy(a[y]->iv,iv,17);
161 memcpy(a[y]->key,key,32);
162 }
163
164 else
165 {
166 a[y]->plaintext=(unsigned char*)
167           calloc(subsub_size,sizeof(unsigned char*));
168 a[y]->ciphertext=(unsigned char*)
169           calloc(subsub_size,sizeof(unsigned char*));
170 a[y]->plaintext_len=subsub_size;
171 a[y]->flag=0;
172 memcpy(a[y]->iv,iv,17);
173 memcpy(a[y]->key,key,32);
174 memcpy(a[y]->plaintext,pBuffers+(y*subsub_size),subsub_size
175 }
176 }
```

```
177 omp_set_num_threads(total_threads); // creating threads
178 #pragma omp parallel     // threads executing in parallel
179 {
180 a[omp_get_thread_num()]->ciphertext_len=encrypt
181 (a[omp_get_thread_num()]->plaintext,a[omp_get_thread_num()]->
182 plaintext_len,a[omp_get_thread_num()]-> key,a[omp_get_thread_num()]
183 ->iv, a[omp_get_thread_num()]->ciphertext);
184 a[omp_get_thread_num()]->flag=1;
185 #pragma omp barrier
186 }
187 }
188 }
189 }
190
191 else if(rank>0)
192 {
193 MPI_Recv(&x,sizeof(x),MPI_INT,MPI_ANY_SOURCE,rank+100,
194 MPI_COMM_WORLD,MPI_STATUSES_IGNORE); // processes receives their data
195 pBuffers = (unsigned char*)calloc(x[0], sizeof(unsigned char*));
196 MPI_Recv(pBuffers,x[0],MPI_CHAR,MPI_ANY_SOURCE,rank,
197 MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
198 subsub_size=floor((float)x[0]/(float)total_threads);
199 cipherbuf = (unsigned char**)calloc(total_threads,
200 sizeof(unsigned char*));
201 for(y=0;y<total_threads;y++)
202 {
203 a[y]=(args_struct*)calloc(1,sizeof (args_struct));
204 if((y==(total_threads-1))&&(total_threads*subsub_size)<sub_size)
205 {
206 temp=subsub_size;
```

```
207 subsub_size=(sub_size-(y*subsub_size));
208 a[y]->plaintext=(unsigned char*)calloc
209 (subsub_size,sizeof(unsigned char*));
210 a[y]->ciphertext=(unsigned char*)calloc
211 (subsub_size,sizeof(unsigned char*));
212 a[y]->plaintext_len=subsub_size;
213 a[y]->flag=0;
214 memcpy(a[y]->iv,iv,17);
215 memcpy(a[y]->key,key,32);
216 memcpy(a[y]->plaintext,pBuffers+(y*temp),subsub_size);
217 }
218 else
219 {
220 a[y]->plaintext=(unsigned char*)calloc
221 (subsub_size,sizeof(unsigned char*));
222 a[y]->ciphertext=(unsigned char*)calloc
223 (subsub_size,sizeof(unsigned char*));
224 a[y]->plaintext_len=subsub_size;
225 a[y]->flag=0;
226 memcpy(a[y]->iv,iv,17);
227 memcpy(a[y]->key,key,32);
228 memcpy(a[y]->plaintext,pBuffers+(y*subsub_size),subsub_size);
229 }
230
231 }
232 omp_set_num_threads(total_threads);
233 #pragma omp parallel
234 {
235 a[omp_get_thread_num()]->ciphertext_len=
236 encrypt(a[omp_get_thread_num()]->plaintext,
```

```
237 a[omp_get_thread_num()]->plaintext_len,
238 a[omp_get_thread_num()]-> key,a[omp_get_thread_num()]->iv,
239 a[omp_get_thread_num()]->ciphertext);
240 a[omp_get_thread_num()]->flag=1;
241 memcpy(cipherbuf[omp_get_thread_num()],a[omp_get_thread_num()]
242 ->ciphertext,a[omp_get_thread_num()]->ciphertext_len);
243 fputs(cipherbuf[omp_get_thread_num()], f);
244 #pragma omp barrier
245 }
246
247 }
248 time_spent = (end-begin)*1000;
249 printf("Total time=%f ms\n",time_spent);
250 EVP_cleanup();
251 ERR_free_strings();
252 MPI_Finalize();
253 return 0;
254
255 }
```

Appendix C

AES Encryption with CUDA

AES Encryption with C using CUDA

```
#include <stdio.h>
#include <omp.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <cuda_runtime.h>

const int Nb_h = 4;
const int Nr_h = 10;
const int Nk_h = 4;
const uint8_t key[16]={0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6};
double begin;
float diff ;
double end ;
```

```
17
18 const uint8_t s_h[256]=
19 {
20       0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
21        0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
22       0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
23        0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
24       0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
25       0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
26       0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A,
27       0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
28       0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
29       0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
30       0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
31       0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
32       0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
33       0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
34       0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
35        0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
36       0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17,
37        0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
38       0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88,
39        0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
40       0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
41        0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
42       0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9,
43        0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
44       0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
45        0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
46       0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
```

```
47        0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
48       0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94,
49        0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
50       0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68,
51        0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
52 };
53
54 uint8_t Rcon_h[256] = {
55       0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
56        0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
57       0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
58        0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
59       0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
60        0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
61       0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
62        0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
63       0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
64        0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
65       0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
66        0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
67       0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
68        0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
69       0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e,
70        0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
71       0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
72        0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
73       0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8,
74       0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
75       0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
76        0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
```

```
77        0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91,
78         0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
79        0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,
80        0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
81        0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
82        0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
83        0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
84         0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
85        0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,
86         0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d
87 };
88
89
90 __constant__ uint8_t s[256];
91 __constant__ uint8_t Rcon[256];
92 __constant__ int Nb;
93 __constant__ int Nr;
94 __constant__ int Nk;
95 __constant__ uint32_t ek[44];
96
97 #define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
98 inline void cudaDevAssist(cudaError_t code, int line, bool abort=true)
99 {
100    if (code != cudaSuccess)
101    {
102        fprintf(stderr,"cudaDevAssistant: %s %d\n",
103         cudaGetErrorString(code), line);
104        if (abort) exit(code);
105    }
106 }
```

```
107
108 __device__ uint32_t sw(uint32_t word)
109 {
110    union {
111       uint32_t word;
112       uint8_t bytes[4];
113    } subWord  __attribute__ ((aligned));
114    subWord.word = word;
115
116    subWord.bytes[3] = s[subWord.bytes[3]];
117    subWord.bytes[2] = s[subWord.bytes[2]];
118    subWord.bytes[1] = s[subWord.bytes[1]];
119    subWord.bytes[0] = s[subWord.bytes[0]];
120
121    return subWord.word;
122 }
123
124 __device__ void sb(uint8_t* in)
125 {
126    for (int i = 0; i < 16; i++) { in[i] = s[in[i]]; }
127 }
128
129 __device__ void mc(uint8_t* arr)
130 {
131    for (int i = 0; i < 4; i++)
132    {
133       uint8_t a[4];
134       uint8_t b[4];
135       uint8_t c;
136       uint8_t h;
```

```
137         for(c=0;c<4;c++) {
138             a[c] = arr[(4*c+i)];
139             h = (uint8_t)((signed char)arr[(4*c+i)] >> 7);
140             b[c] = arr[(4*c+i)] << 1;
141             b[c] ^= 0x1B & h;
142         }
143         arr[(i)] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1];
144         arr[(4+i)] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2];
145         arr[(8+i)] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3];
146         arr[(12+i)] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0];
147     }
148 
149 }
150 
151 __device__ void sr(uint8_t* arr)
152 {
153     uint8_t out[16];
154     //On per-row basis (+1 shift ea row)
155     //Row 1
156     out[0] = arr[0];
157     out[1] = arr[1];
158     out[2] = arr[2];
159     out[3] = arr[3];
160     //Row 2
161     out[4] = arr[5];
162     out[5] = arr[6];
163     out[6] = arr[7];
164     out[7] = arr[4];
165     //Row 3
166     out[8] = arr[10];
```

```
167     out[9] = arr[11];
168     out[10] = arr[8];
169     out[11] = arr[9];
170     //Row 4
171     out[12] = arr[15];
172     out[13] = arr[12];
173     out[14] = arr[13];
174     out[15] = arr[14];
175
176     for (int i = 0; i < 16; i++)
177     {
178        arr[i] = out[i];
179     }
180 }
181
182 __device__ uint32_t rw(uint32_t word)
183 {
184     union {
185        uint8_t bytes[4];
186        uint32_t word;
187     } subWord  __attribute__ ((aligned));
188     subWord.word = word;
189
190     uint8_t B0 = subWord.bytes[3], B1 = subWord.bytes[2], B2 =
191         subWord.bytes[1], B3 = subWord.bytes[0];
192     subWord.bytes[3] = B1; //0
193     subWord.bytes[2] = B2; //1
194     subWord.bytes[1] = B3; //2
195     subWord.bytes[0] = B0; //3
196
```

```
197     return subWord.word;
198 }
199 
200 __global__ void K_Exp(uint8_t* pk, uint32_t* out)
201 {
202     int i = 0;
203     union {
204         uint8_t bytes[4];
205         uint32_t word;
206     } temp __attribute__ ((aligned));
207     union {
208         uint8_t bytes[4];
209         uint32_t word;
210     } univar[44] __attribute__ ((aligned));
211 
212     for (i = 0; i < Nk; i++)
213     {
214         univar[i].bytes[3] = pk[i*4];
215         univar[i].bytes[2] = pk[i*4+1];
216         univar[i].bytes[1] = pk[i*4+2];
217         univar[i].bytes[0] = pk[i*4+3];
218     }
219 
220     for (i = Nk; i < Nb*(Nr+1); i++)
221     {
222         temp.word = univar[i-1].word;
223         if (i % Nk == 0)
224         {
225             temp.word = (sw(rw(temp.word)));
226             temp.bytes[3] = temp.bytes[3] ^ (Rcon[i/Nk]);
```

```
227         }
228         else if (Nk > 6 && i % Nk == 4)
229         {
230            temp.word = sw(temp.word);
231         }
232         if (i-4 % Nk == 0)
233         {
234            temp.word = sw(temp.word);
235         }
236         univar[i].word = univar[i-Nk].word ^ temp.word;
237      }
238      for (i = 0; i < 44; i++)
239      {
240         out[i] = univar[i].word;
241      }
242   }
243
244   __device__ void ark(uint8_t* state, int strD, uint32_t* eK)
245   {
246      union {
247         uint32_t word;
248         uint8_t bytes[4];
249      } kb[4] __attribute__ ((aligned));
250
251      kb[0].word = eK[strD];
252      kb[1].word = eK[strD+1];
253      kb[2].word = eK[strD+2];
254      kb[3].word = eK[strD+3];
255
256      for (int i = 0; i < 4; i++)
```

```
257    {
258        state[i] = state[i] ^ kb[i].bytes[3];
259        state[i+4] = state[i+4] ^ kb[i].bytes[2];
260        state[i+8] = state[i+8] ^ kb[i].bytes[1];
261        state[i+12] = state[i+12] ^ kb[i].bytes[0];
262    }
263 }
264
265 __global__ void cudaRunner(uint8_t *state)
266 {
267    ark(state, 0, ek);
268    for (int i = 1; i < Nr; i++)
269    {
270        sb(state);
271        sr(state);
272        mc(state);
273        ark(state, i*Nb, ek);
274    }
275
276    sb(state);
277    sr(state);
278    ark(state, Nr*Nb, ek);
279 }
280
281 void* mainCypher(uint8_t *tc,int pro)
282 {
283    uint8_t *devState = NULL;
284         cudaSetDevice(pro);
285    cudaDevAssist(cudaMalloc((void**)&devState,
286                16*sizeof(uint8_t)), 266, true);
```

```
287    cudaDevAssist(cudaMemcpy(devState, &tc, 16*sizeof(uint8_t)
288    ,cudaMemcpyHostToDevice), 267, true);
289    cudaDevAssist(cudaDeviceSynchronize(), 268, true);
290         cudaSetDevice(pro);
291    cudaRunner<<<1,1>>>(devState);
292    cudaDevAssist(cudaDeviceSynchronize(), 270, true);
293    cudaDevAssist(cudaMemcpy(&tc, devState, 16*sizeof(uint8_t)
294    , cudaMemcpyDeviceToHost), 271, true);
295    cudaDeviceSynchronize();
296    cudaFree(devState);
297    return NULL;
298 }
299 void encrypt_gpu(unsigned char *pBuffer,int len,int processor,int ↩
       gpu_threads)
300 {
301    uint32_t *dev_ek;
302    uint8_t *dev_pkey;
303    uint32_t eK[44];
304    int count=0;
305    int i10=-1;
306    int index[16]={0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15};
307    int in=-1;
308    int spawn = 0;
309    uint8_t** state;
310    int x=0;
311    int y=0;
312    int z;
313    state = (uint8_t**)calloc((int)floor(len/16)+1,
314                sizeof(uint8_t*));
315     cudaSetDevice(processor);
```

```
316  cudaDevAssist(cudaMemcpyToSymbol(Nk, &Nk_h, sizeof(int),
317   0, cudaMemcpyHostToDevice), 535, true);
318  cudaDevAssist(cudaMemcpyToSymbol(Nr, &Nr_h, sizeof(int),
319   0, cudaMemcpyHostToDevice), 543, true);
320  cudaDevAssist(cudaMemcpyToSymbol(Nb, &Nb_h, sizeof(int),
321  0, cudaMemcpyHostToDevice), 903, true);
322  cudaDevAssist(cudaMemcpyToSymbol(Rcon, &Rcon_h,
323  256*sizeof(uint8_t),0, cudaMemcpyHostToDevice), 534, true);
324  cudaDevAssist(cudaMemcpyToSymbol(s, &s_h, 256*sizeof(uint8_t),
325   0, cudaMemcpyHostToDevice), 920, true);
326  cudaDevAssist(cudaMalloc((void**)&dev_pkey,
327   16*sizeof(uint8_t)), 317, true);
328  cudaDevAssist(cudaMalloc((void **)&dev_ek,
329   44*sizeof(uint32_t)), 932, true);
330  cudaDevAssist(cudaMemcpy(dev_pkey, &key,
331  16*sizeof(uint8_t), cudaMemcpyHostToDevice), 318, true);
332  cudaThreadSynchronize();
333       cudaSetDevice(processor);
334  K_Exp<<<1, 1>>>(dev_pkey, dev_ek);
335  cudaThreadSynchronize();
336  cudaDevAssist(cudaMemcpy(&eK, dev_ek, 44*sizeof(uint32_t),
337   cudaMemcpyDeviceToHost), 368, true);
338  cudaThreadSynchronize();
339  cudaDevAssist(cudaMemcpyToSymbol(ek, &eK, 44*sizeof(uint32_t),
340   0, cudaMemcpyHostToDevice), 823, true);
341  cudaThreadSynchronize();
342  cudaFree(dev_ek);
343  cudaFree(dev_pkey);
344  cudaDeviceReset();
345
```

```
346   while (len>count)
347   {
348            i10++;
349            state[i10] = (uint8_t*)calloc(16, sizeof(uint8_t*));
350            in=0;
351         spawn++;
352         while(in<16)
353         {
354            state[i10][index[in]] = pBuffer[count];
355
356                                  count++;
357                                  in++;
358                  if ((len==count) && (in > 0))
359                                 {
360                                   while(in<16)
361                                     {
362                                     state[i10][index[in]] = 0x00; in++;
363                                     }
364                                  }
365         }
366      }
367   if(gpu_threads>spawn)
368   {gpu_threads=spawn;}
369   x=floor(spawn/gpu_threads);
370   if(processor==0 && omp_get_thread_num()==0)
371   {begin=omp_get_wtime();}
372   for(z=0;z<x;z++)
373      {
374         if(z==(x-1) && ((z*gpu_threads)+gpu_threads)
375                                                <spawn)
```

```
376             {
377                             y=spawn-((z*gpu_threads)+gpu_threads);
378                             omp_set_num_threads(gpu_threads+y);
379             }
380           else
381             {
382                            omp_set_num_threads(gpu_threads);
383                 }
384             #pragma omp parallel
385                {
386                       cudaSetDevice(processor);
387                  mainCypher(state[(z*gpu_threads)+
388                         (omp_get_thread_num())],processor);
389                       #pragma omp barrier
390              }
391       }
392
393    if(processor==0 && omp_get_thread_num()==0){
394     end=omp_get_wtime();
395     diff = (end-begin)*1000;
396           }
397 }
398
399
400
401 int  main(int argc, char *argv[])
402 {
403
404   int total_pro;
405   unsigned char *pBuffers =NULL;
```

```
406  FILE *fp = NULL;
407  long size;
408  long sub_size;
409  fp = fopen(argv[1], "r");
410  if(fp==NULL)
411  {
412     printf("unable to open");
413  }
414
415  total_pro=4;
416  fseek (fp, 0, SEEK_END);
417  size = ftell (fp);
418  fseek (fp, 0, SEEK_SET);
419  sub_size=floor((float)size/(float)total_pro);
420  sub_size=floor((float)size/(float)total_pro);
421  fseek (fp, 0, SEEK_CUR);
422  pBuffers = (unsigned char*)calloc(size, sizeof(unsigned char*));
423  fread(pBuffers,sizeof (unsigned char) ,size, fp);
424  omp_set_num_threads(total_pro);
425  #pragma omp parallel
426   {
427   if((omp_get_thread_num()==(total_pro-1))&&
428         ((total_pro-1)*sub_size)<size)
429   {
430       sub_size=(size-((total_pro-1)*sub_size));
431   }
432
433  unsigned char *plaintext =NULL;
434  plaintext = (unsigned char*)calloc(sub_size, sizeof(unsigned char*));
435  memcpy(plaintext,pBuffers+(omp_get_thread_num()*sub_size),sub_size);
```

```
436   encrypt_gpu(plaintext,sub_size,omp_get_thread_num(),
437       strtol(argv[2], NULL, 10));
438   #pragma omp barrier
439
440  }
441   printf("Done - Time taken: %f ms\n", diff);
442  return 0;
443 }
```