Parallelization of Symmetric and Asymmetric security Algorithms for MultiCore Architectures

T. Balamurugan[1], T. Hemalatha[2]

[1]PG Scholar, PSNA College of Engineering & Technology, Dindigul, Tamilnadu-624622, India,

[2]Associate Professor, PSNA College of Engineering & Technology, Dindigul, Tamilnadu-624622, India,

Abstract: ***Symmetric and Asymmetric algorithms are complex that requires large number of mathematical computations to be done. The sequential execution of the algorithms would require a considerable amount of execution time. This may not feasible for most of the applications that require faster rate of encryption and decryption to match the required data flow. The objective of this work is to optimize the execution of Symmetric and Asymmetric algorithms at code level that is to be implemented in a MultiCore environment. In this work, code level parallelism is preferred since it can work on any architecture without any modifications. Hence, our objective is to parallelize the existing security algorithms, so that the proposed algorithm can enhance the power of MultiCore architectures thereby it leads to less run time complexity. In this work the time complexity of the most popular algorithms for parallelization since it is predominantly used in all Network Protocols and Applications. The work focuses on enhancing the performance of these algorithms by parallelizing code so that it can utilize more than one processor in MultiCore architecture efficiently for its execution.***

Keywords: Cryptography, Parallel Computation, AES, DES

1. Introduction

Earlier computers had only uniprocessor in which Single central processing unit is used to execute all computer tasks sequentially one at a time. All the operations are executed sequentially. Uniprocessor is used to distinguish the class of computers where all processing tasks are executed in a single CPU. Recent desktop computers are having multiprocessing architectures. Recent computers have MultiCore processors which have more than one CPU. MultiCore computers have two or more Central Processing Units (CPUs) within a single computer system. Multiprocessing refers to the execution of multiple concurrent processes in a system as opposed to a single process at any one instant.

1.1 Parallel Computing

Parallel computing is the computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. Parallelism can be achieved in data level, task level and code level.

1.1.1 Code level parallelism

Code level parallelism is measure of number of lines of codes that are executed in a computer system can be performed simultaneously. The potential overlap among code is called Code level parallelism. Code level parallelism can be implemented by using Message Passing Interface (MPI) and OpenMP. Here OpenMP tool is used for parallel computing. OpenMP supports Linux platform. We can speedup by multiple threads simultaneously. OpenMP is directly supported by the compiler. Additional libraries unlike MPI need not be installed.

Open MultiProcessing is the tool used for parallel programming interface which is supported by MultiCore architectures to provide multithreaded shared memory parallelism. The master thread executes sequentially until the first parallel region construct is encountered and then it creates a team of parallel threads, within the parallel region construct are then executed in parallel among the various threads. [6]

1.1.2 Techniques used in OpenMP

a) Fork/join threads
- Synchronization
- Mutual exclusive

b) Assign/distribute work to threads
- Work share
- Task queue

c) Run time control
- Query/request available resources
- Interaction with OS

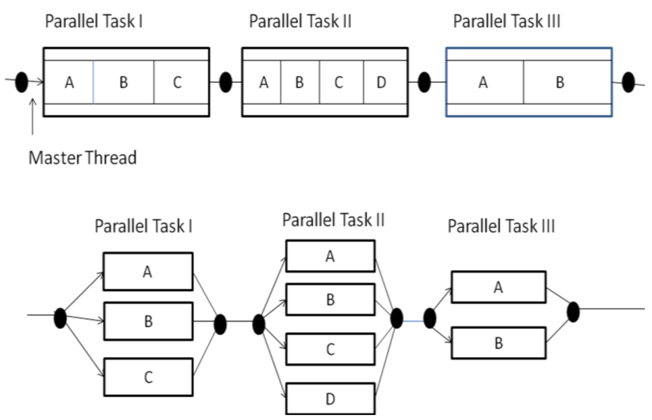

Figure 1: OpenMP Execution Model

1.1.3 MPI

MPI is a communication protocol used to program parallel computers. MPI is a message-passing application programmer interface. mpicc is a program which helps the programmer to use a standard C programming language compiler together with the Message Passing Interface (MPI) libraries Most MPI implementations consist of a specific set of routines. MPI's goals are high performance, scalability, and portability.

1.1.4 Parallel Algorithm Models

An algorithm model is the representation of a parallel algorithm by selecting a strategy for dividing the data and processing technique and applying the appropriate method. The various models available are

- The Data Parallel model
- The Task Graph model
- The Work Pool model
- The Master Slave model
- The Pipeline or Producer Consumer model
- Hybrid models

1.1.4.1 The Data-Parallel Model

In this model, the tasks are statically or semi-statically attached onto processes and each task performs identical operations on a variety of data. Tasks are executed in different phases that are uniformly partitioned. Data-Parallel algorithms can be implemented in both shared address space and message passing paradigms. Shared address space is easily programming. Dense matrix multiplication is an example for this model.

1.1.4.2 The Task Graph Model

The computations in any parallel algorithm can be viewed as a task graph. The type of parallelism that is expressed by the task graph is called task parallelism. This model is applied to huge relative to the amount of data associated with them. Parallel quick sort, sparse matrix factorization are the example for this model.

1.1.4.3 The Work Pool Model

The work pool or the task pool model is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be executed by any process. There is no desired pre-mapping of tasks onto processes. Parallel tree search is an example for thi]]s model.

1.1.4.4 The Master-Slave Model

In the master-slave model, one or more master processes generate work and allocate it to slave processes. The manager can estimate the size of the tasks or random mapping can do an adequate job of load balancing. Workers are assigned smaller pieces of work. This model is suitable to shared address space or message passing paradigms.

1.1.4.5 The Pipeline or Producer-Consumer Model

In the pipeline model, a stream of data is passed on through a succession of processes. This simultaneous execution of different programs on a data stream is called stream parallelism.

1.1.4.6 Hybrid model

In this hybrid model more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. This model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. Parallel quick sort is an example for this model. [7]

1.2 Security

Security is the most important requirement for exchanging information from one place to another. The Cryptography mechanism can provide the most secure way of transferring information between sender and receiver. The objective is to make the secret information to non readable format to all other except specified receiver. So Advanced Encryption Algorithm (AES) and Data Encryption Standard algorithm (DES) provides confidentiality. AES and DES algorithm requires more memory and execution time. So that we have to speed up the AES and DES algorithm by parallelizing the implementations. So that it can enhances the computing power if all cores. This enhances the performance by reducing the running time and speeds up the encryption and decryption powers. The existing implementations of such algorithm are designed for executing in single core architecture.

Security algorithms are categorized into two types. They are Symmetric algorithm and Asymmetric algorithm.

1.2.1 Symmetric algorithm

Symmetric algorithm is also called shared key algorithm in which both encryption and decryption uses the same key. The speed of this algorithm is at least 1000 times faster when compared to asymmetric algorithm. The weakness of this algorithm is distribution of the key and does not provide non repudiation of data. [8]

Example: AES, Blowfish, DES, Serpent, Twofish.

1.2.2 Asymmetric algorithm

Asymmetric algorithm is also called public key algorithm, public key is used for encryption and private key is used for decryption. Public key is available to all and private need to be secured.

Example: Diffie–Hellman key exchange, Digital Signature Algorithm, ElGamal, RSA, Merkle–Hellman.

Table 1: list of Security Services & its Algorithms

Security service	*Most widely used algorithms*
Confidentiality	SSL_RSA_WITH_RC4_128_MD5
Authentication	Rivest Shamir Adleman (RSA) algorithm and Diffie-Hellman (DH) algorithm
Integrity	SHA-1, MD5
Non reputation	Digital signature

2. Related Work

Jyothi and Omar suggested the parallelization of AES algorithm in their work by proposing a hardware model. The hardware models of advanced encryption standard algorithm protects from malicious attacks. The Rinjindael AES algorithm with key length 128 bits is implemented in hardware. The two hardware models based on HDL and IP core are used to evaluate the performance of the algorithm. As single custom processors are used in specific applications, they outperform general purpose processors used in a SoPC model. The encryption time and the performance metrics such as speed and memory utilization are evaluated. Since the proposed system uses a hardware model and the fixed length of key size 128 bit alone is supported. Other key lengths are not supported. The proposed system is hardware specific thereby the parallelized implementation is not portable. Hence the hardware implementation is not efficient than the software implementation. [1]

Dazhong Wang and Xiaoni suggested the Parallelization of AES algorithm in their work by improved method to increase AES system speed which is used to protect electronic data. The system consists of two main components Cipher and Key-Expansion. Furthermore, there are four small parts in process of cipher conversion. The AES algorithm is capable of using cryptographic keys of 28,192 and 256 bits to encrypt and decrypt information. In the design, after receiving the key matrix, every part of the key expansion is continuously working state. The S-box size is used to improve the encrypting performance. Combining the AES algorithm and the RSA algorithm that could increase the speed of encryption and the decryption. Many devices like memory units and time delay are used to increase the speed. There are various methods used in this work, not focused on MultiCore environment. [2]

Navalgund suggested the parallelization of AES algorithm in their work by Parallelization of AES algorithm using OpenMP. The parallelization of AES algorithm is optimized at data level and control level. According to the parallel computation paradigms, the independent parts of the algorithms must be identified and then prepared to work in separate threads. The AES algorithm is divided into parallelizable and unparallelizable parts. The parallelized portion and unparallelized portion are joined using fork-join model. This work is focused on parallelization at data level and control level. AES and DES have number of iteration for encryption and decryption. So it is not more efficient than using code level. The system is specially designed for AES algorithm, not focused on DES algorithm. [3]

Bin Liu and Bevan M. Baas suggested the parallelization of AES algorithm in their work by improving the performance of AES algorithm at task level and data level. The system is designed for fine-grained many core systems. The design of this system requires minimum six cores for offline key expansion and eight cores for online key expansion. The largest requires 107 and 137 cores respectively. The fastest design achieves a throughput, when the processors are running at a specified frequency. This system allocates the workload of each processor, which reduces the number of cores. This type of parallelization is suitable only for 6 cores but not less than that i.e. 2/3/4 cores. Parallelization is done only for AES under symmetric cryptosystem. [4]

Nagendra and Chandra Sekhar suggested the Parallelization of AES algorithm in their work by improving the performance of Advanced Encryption algorithm using parallel computation that uses divide and conquer strategy. Such strategy is used in parallel computation to solve algorithms in parallel by partitioning and allocating number of given subtask to available processing units. The text file is given as input. The text file is decomposed into number of blocks. Each block is executed in a single core. The implementation of cryptography algorithm is done on dual core processor. OpenMP API is used to reduce the execution time. The system is specially designed for dual core processors but recent processors have many cores, thereby performance of the system is not improved much. The system is specially designed for AES algorithm, under Symmetric cryptosystems. But most of the applications in internet use only asymmetric cryptography to achieve confidentiality, authentication and integrity. The blocks of text file are executed in dual core processor, so that the CPU resources are not fully utilized in looping statements. The text file is executed in parallel, but the source code is not parallelized. [5]

3. Techniques

The objective of the proposed system is attained in two phases. In phase I, the sequential code that satisfies the property of SIMD is selected. In this phase, the sequential code and parallelized codes are executed in Linux platform. gcc compiler is used execute the sequential code. mpicc complier is used execute the parallelized code. The result of execution time is calculated.

Message Passing Interface is a powerful model used in high performance computing. It allows users to create programs that can run efficiently on most parallel architectures. Communicator is a group of processes that are used to communicate to each other. Rank is to distinguish each process of the communicator, an ID assigned to each of termed as rank of the process. One processor communicates explicitly to another process using this ID. Size is the total number of processes belonging to a communicator.

MPI runs on virtually any hardware platform. They are
- Distributed Memory
- Shared Memory
- Hybrid

The group of routines is used for interrogating and setting the MPI execution environment. They are

<u>MPI_Init</u>

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
MPI_Init (&argc,&argv)
MPI_INIT (ierr)
```

<u>MPI_Comm_size</u>

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
```

MPI_Comm_rank

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID.

```
MPI_Comm_rank (comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
```

MPI_Abort

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates all processes regardless of the communicator specified.

```
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

MPI_Get_processor_name

This function returns the name of the processor. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name
(&name,&resultlength)
MPI_GET_PROCESSOR_NAME
(name,resultlength,ierr)
```

MPI_Get_version

It returns the version and subversion of the MPI standard that's implemented by the library.

```
MPI_Get_version (&version,&subversion)
MPI_GET_VERSION (version,subversion,ierr)
```

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized (&flag)
MPI_INITIALIZED (flag,ierr)
```

MPI_Wtime

Returns an elapsed wall clock time in seconds on the calling processor.

```
MPI_Wtime ()
MPI_WTIME ()
```

MPI_Wtick

Returns the resolution in seconds of MPI_Wtime.

```
MPI_Wtick ()
MPI_WTICK ()
```

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

For MPI program in C language include the header file mpi.h using the command #include<mpi.h> will include all MPI subroutines. During the MPI subroutine call from the main program, compiler will go to MPI subroutine definition, which is available at the MPI library and executes the function definition. After execution, result is returned to the program from where it was called. All the MPI subroutines have the form MPI subroutine. In an MPI function call, the arguments that specify the address of a buffer should be specified as pointers. Return code of an MPI function call should be an integer value. Data types defined in the C semantics is in that is more individual.

4. Implementation

The second phase of the proposed work is decomposed into six modules. The high level design of the proposed system is shown in figure 2. The Request handler accepts the source code from the user and it transfers to the code level Parallelizer. The source code may be in C/C++/JAVA languages. The code level Parallelizer stores the code into the database. The source code analyzer fetch the source code by using substring functions. Each substring is converted by using XML tags and it is stored in the file system. The convertor separates the declaration statements, looping statements, etc in source code. Then the looping statements are converted into parallelized code by the Code Level Parallelizer. Code Level Parallelizer consists XML Encoder, XML parser and Parallelizer. XML Encoder encodes the code into the XML instance. XML parser is used to identify the strings in the source code. XML Parallelizer converts the parallelized parts like looping statements. The result of parallelized code sent back to the user.

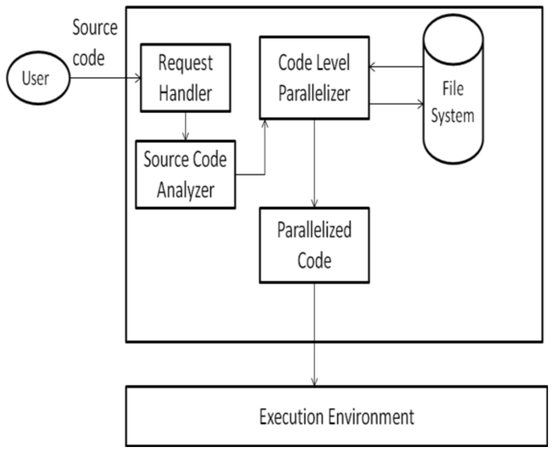

Figure 2: High level Design of the proposed system

The parallelized code can be executed in MultiCore architectures. The system checks for more than one CPU. If it has more number of cores present in the CPU. According to the number of cores the tasks are subdivided and each core executes the appropriate task. After executed of all the tasks the results are obtained from all cores and combined into one. The inputs file are divided into number of core present in the CPU and the encryption/decryption on multiple blocks of code simultaneously executed by core-0, core-1, up to core-(n-1). Else the system has only one core the code executed in the single processor. The sequential and parallel code execution time is calculated.

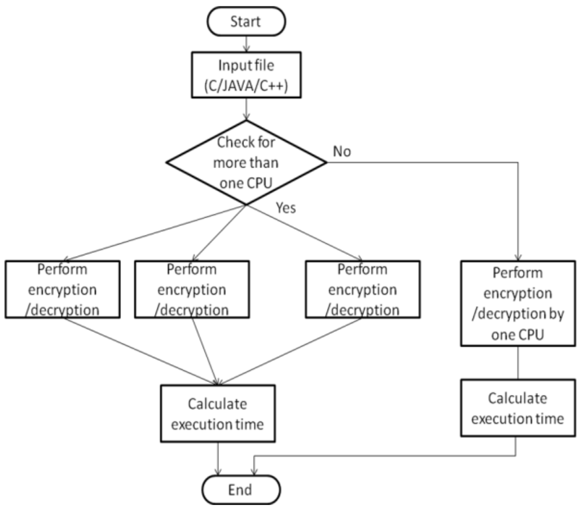
Start
Input file (C/JAVA/C++)
Check for more than one CPU
No
Yes
Perform encryption /decryption
Perform encryption /decryption
Perform encryption /decryption
Perform encryption /decryption by one CPU
Calculate execution time
Calculate execution time
End

Figure 3: Flow chart for parallel execution

The required objective is automated by the proposed system which is implemented using C#. The high level design of the proposed system is decomposed into six modules.
1. Request Handler
2. Source Code Analyzer
3. Code Level Parallelizer (CLP)
4. XML Encoder
5. XML Parser & Parallelizer
6. Performance Analyzer

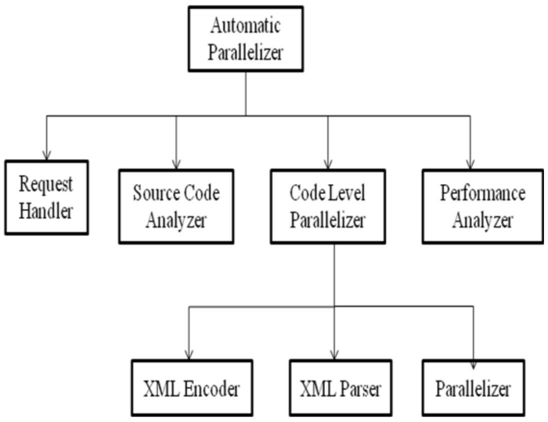

Figure 4: Module level Design of the proposed system

4.1 Request Handler

The proposed system is implemented as a web application through which the user can submit the source code in sequential form. The sequential source code should meet the requirements of proposed systems that it has to be in any one of the high level languages that are supported by the proposed system i.e. C/C++/JAVA. In this request handler, the end user sends code to parallelize, by performing authentication. Only authorized users can use this.

4.2 Source Code Analyzer

The Request Handler dispatches the source code submitted by the end user to this module. The functionality of Source Code Analyzer is to analyze the source code and ensure whether it is supported by the proposed system or not.

Source code analyzer categorizes each statement of the source code. The set of XML tags are stored in the file system that is used to analyze the source code. Using user defined XML tags the source code is categorized into declaration statement, control statement, variables and etc. From top to bottom code is analyzed.

Each and every Word of the source code has to be identified in order to find which area of code to parallelize. The system automatically decides the code which is to be parallelized. XML supports user defined tags and facility to store XML files in the file system.

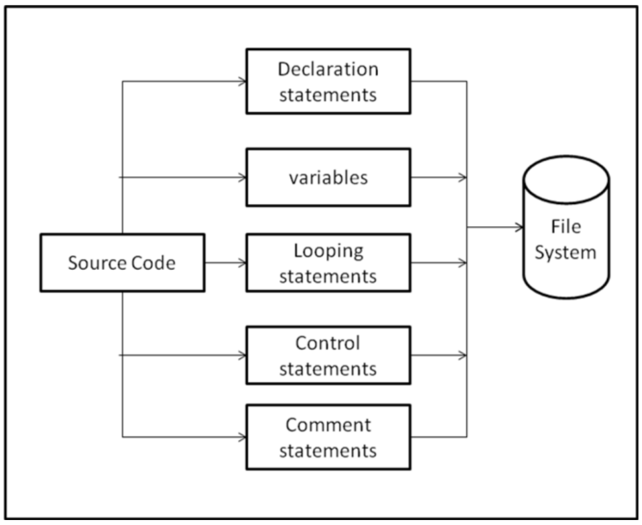

Figure 5: Source Code Analyzer

4.3 Code Level Parallelizer (CLP)

The CLP converts the sequential code into parallelized code. It does the main objective of parallelization. The looping statements are taken for the conversion. This searches each looping statements from the top to bottom of the source code. The design of CLP is shown in the figure 4.5. It consists of three modules

- XML Encoder
- XML Parser
- Parallelizer

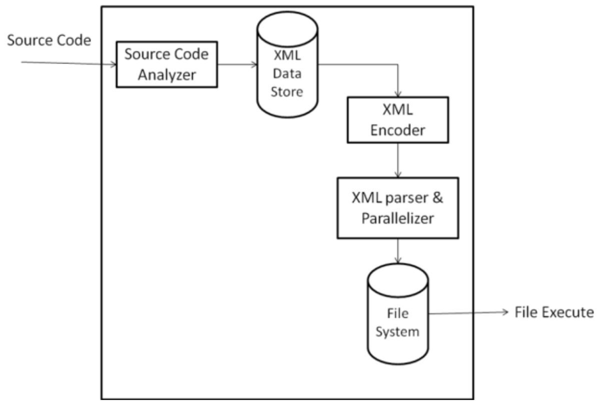

Figure 6: Code Level Parallelizer

4.4 XML Convertor

The source code written in C/C++/JAVA serves as the input to this module. The role of this process is to convert the source code into a form that can be parsed for parallelization. The process of conversion involves following steps.

- Identify each and every section & keyword in the input file.
- Insert the appropriate XML Tags as per the section.
- The Tags are user defined and it is inserted appropriately in the source code so that the task of parallelization can be made simple by the use of XML parser.
- The sample source code, the major user defined tags and parallelized code are listing below.

Example source code:

```
#include <stdio.h>
void main()
{
 int sum=0,i, n=15;
 for(i=0;i<=n;i++)
{
 sum=sum+i; }
printf("Sum= %d",sum);
}
```

XML encoder reads the above program from top to bottom and categorizes the string by the following XML tags.

```
<header_file>#include<stdio.h></header_file>
<data_type>int</data_type>
<variable>sum=0,n=15,i</variable>
<loop>for</loop>
```

Using the above XML parser the above codes are categorized as header files, declaration type, variable, loop, etc. If looping statement is found then the code can be parallelized. Check until the source program reaches the end of file. The Parallelizer translates the looping statements that are executed in parallel. The following program shows the parallelized code for the above example.

The statement for(i=0;i<=n;i++) is identified by the XML Parser. This statement is parallelized as

```
#pragma omp parallel for
 for(i=0;i<=n;i++)
```

Finally the unparallelized codes and parallelized codes are combined into single code as follows.

Parallelized code:

```
#include <stdio.h>
#include "mpi.h"
void main()
{
 int sum=0,i, n=15;
#pragma omp parallel for
 for(i=0;i<=n;i++)
{
 sum=sum+i; }
printf("Sum= %d",sum);
}
```

4.5 XML Parser & Parallelizer

XML Parameterized source code is parsed by the XML parser. XML parser is administered by the rules. Rules are defined for each of the high level language like C, C++, and Java. These rules are fetched by the XML parser from the rule repository and it is applied on the source code. Whenever there is a match against a rule in the source code, the Parallelizer is involved to insert the corresponding parallel construct in the source code. This process is repeated until it reaches End of file. The output of these two modules is parallelized code.

5. Results and Analysis

The system accepts the sequential code from the end user and returns the parallelized code to the end user. The parallelized code is stored in the file system which is further pushed to the end user by two means. They are

- If the client is synchronous, the Parallelized code is sent to the client using push technology.
- If the client is asynchronous, the parallelized code is sent to the client's mail box using SMTP/POP3 protocol.

The execution times of sequential and parallelized code for different sizes are calculated by the performance analyzer in the environment of OpenMP, Linux, Dual core CPU and 2 GB RAM. The execution time required by different size text input files for encryption and decryption process. The execution time of sequential code is greater than the parallelized code in all cases. The AES and DES algorithm has been successfully parallelized by using OpenMP.

6. Conclusion

In this work, the parallelization of sequential algorithms are done by using MPI. The execution times of both sequential and parallel algorithm are measured. The parallel implementation takes very less time than the sequential implementation. Hence, multi-core processors can execute in less time compared to sequential execution of AES and DES cryptography algorithm. The multi-core processing environment can be utilized effectively by performing code level parallelism, which provides an efficient and reliable way to execute AES and DES cryptography algorithm with less execution time thereby the overall performance can be improved.

References

[1] Jyothi Yenuguvanilanka and Omar Elkeelany, "Performance Evaluation of Hardware Models of Advanced Encryption Standard (AES) Algorithm", IEEE, pp. 222-225, 2008.
[2] Dazhong Wang and Xiaoni Li, "Improved Methods to Increase AES system Seed", The Ninth Conference on Electronic Measurement & Instruments, pp. 49-52, 2009.
[3] Navalgund. S. S, Akshay Desai, Krishna Ankalgi and Harish Yamanur, "Parallelization of AES Algorithm Using OpenMP", Lecture Notes on Information Theory Vol. 1 No. 4, pp. 144-147, 2013
[4] Bin Liu, "Parallel AES Encryption Engines for many core Processor Arrays, IEEE, pp.536-547.
[5] Nagendra. M and Chandra Sekhar. M, "Performance Improvement of Advanced Encryption Algorithm using Parallel Computation", International Journal of Software Engineering and Its Applications, pp.287-296, 2014
[6] Shao-Ching Huang ,"Parallel Computing and OpenMP Tutorial", 2013.
[7] http://nptel.ac.in/courses/106106112/2
[8] www.cryptoforge.com/what-is-encryption.htm.
[9] William Stallings, "Cryptography and Network Security", 4th Edition, 2006.

Author Profile

T. Balamurugan received the D.C.T from Christian Polytechnic College, Oddanchatram in 2003 and B.E. degree in Computer Science and Engineering from R.V.S College of Engineering and Technology, Dindigul in 2007. During 2013-2015, he is doing M.E Computer Science and Engineering, from PSNA College of Engineering and Technology, Dindigul.

T. Hemalatha received the Bachelors degree in Computer Science & engineering from Thiagarajar College of Engineering in 1994 and Masters Degree in Computer Science & Engineering from College of Engineering - Guindy, Anna University in 2004. She is a research scholar of Anna University. Currently, she is an Associate Professor at P.S.N.A. College of Engineering and Technology, Tamilnadu, India. Her interests are in Network Security, Grid Computing, Distributed System and Computer Networks.