Parallelization Of RSA Algorithm Using OpenMP

[1]NISHA M PATIL, [2]AISHWARYA S BINGERI, [3]AKSHATA SAJJAN,[4]MUSKAN KHAZI

[1]Nisha M Patil,Student,Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India
[2]Aishwarya S Bingeri,Student,Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India
[3]Akshata Sajjan,Student,Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India
[4]Muskan Khazi,Student,Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India

Abstract

In the field of Cryptography, public key algorithms satisfy all the four requirements of security: Confidentiality, Integrity, Non -repudiation, Authentication as compared to the conventional Crypto system. . RSA and RSA-based digital signature algorithms are some of the most popularly used public-key infrastructure cryptographic algorithms with their roots in the modular arithmetic. Public key Cryptography is widely known to be slower than symmetric key alternatives for massive computations of modular arithmetic. The modular arithmetic and modular exponentiation makes RSA computationally expensive when compared to symmetric algorithms. Therefore, to make a more efficient, secure and faster implementation of RSA Algorithms is publicly concerned. With the development of the OpenMP , more computing problems are solved by using the CPU(Central Processing Unit) based implementation.

Index Terms – Cryptography,Encryption,Decryption,Plaintext,Ciphertext,RSA,OpenMP,Parallel Computing,Modular.

1. INTRODUCTION

The concept of public key cryptography (PKC) was invented and introduced by Whitfield Diffie and Martin Hellman, and independently by Ralph Merkle. Their contribution to cryptography was that the keys could come in pairs i.e. an encryption key and a decryption key and the decryption key cannot (practically) be derived from the encryption key. Public key methods are important because they can be used for transmitting encryption keys or other data securely even when the parties have no opportunity to agree on a secret key in private. The encryption key is also called the public key and the decryption key the private key. The security provided by these ciphers is based on keeping the private key secret. Public key encryption and decryption is compute-intensive because a lot of modular multiplications with very large numbers are needed to perform these tasks. Therefore public key algorithm is known to be much slower than symmetric key algorithms. Recently the use of OpenMP on the GCC infrastructure for general purpose computing has been gaining widespread usage for parallelizing algorithms. Many computational problems have gained a significant performance increase by using the highly parallel properties of the Open MP. GCC infrastructure is a

framework which makes these kinds of implementations available to the general programmers. The OpenMP approach makes it simpler to implement parallel programs.

The target in this work is to develop a secure, faster and efficient RSA using OpenMP. This implementation focuses on parallelizing the exponentiation operation of the algorithm. To provide a robust analysis, the study makes use of a High Performance Computing environment to illustrate results for different scenarios in terms of parallel processing units. Through experimental analysis, the implementation is shown to have greatly improved execution times when compared against serial implementation.

1.1 RSA ALGORITHM

The RSA algorithm was introduced in 1977 and is one of the most important algorithms used for encryption and authentication on Internet. It was the first algorithm suitable for both digital signature and data encryption applications. It is widely used in the protocols supporting the e-commerce today. Mathematically, it is based on factorization of large integers, which is computationally very difficult to carry out. It provides strong security with sufficiently long keys. For example, if the key length of 1024 bits is used then it is nearly impractical to break up the security of RSA encryption even when working with high performance computers. The RSA algorithm is divided into three parts – Key Generation, Encryption, and Decryption.

1.1.1 Key Generation

The key generation part of RSA algorithm is multi-step process which is given below –

1. Choose two very large random prime integers having bit size 512: p and q.

2. Compute m = p*q, which is used a modulus.

3. Compute $\varphi(n) = (p-1)(q-1)$.

4. Choose an integer e, $1 < e < \varphi(n)$ such that: $GCD(e, \varphi(n)) = 1$(GCD is greatest common denominator) .

5. Compute d, $1 < d < \varphi(n)$ such that: $ed \equiv 1 \pmod{\varphi(n)}$ Since in the above procedure e is the public or encryption exponent and d is the private or decryption exponent, thus Publish e and n as the public key and keep d and n as the secret key.

1.1.2 RSA Encryption

In order to encrypt, the plain text data is raised to the power of encryption key and then divided by the product of the prime numbers to calculate the remainder. The remainder is sent as cipher text.

C = M^e % m

1.1.3 RSA Decryption

In order to decrypt, the cipher text data is raised to the power of decryption key and then divided by the product of the prime numbers to calculate the remainder. The remainder is the original plain text.

M = C^e % m

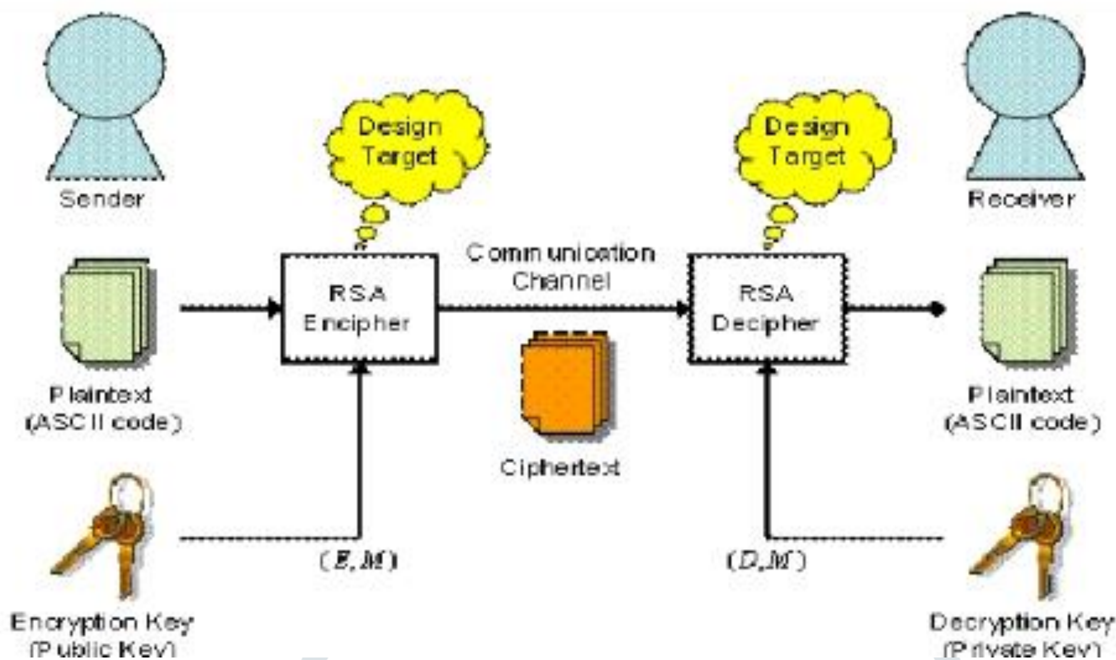


Figure 1.1 Structure Of RSA Algorithm

2. PROBLEM STATEMENT

Due to increased data movement and information exchange over internet and web, preserving data confidentiality and security has emerged as a prime concern for the end users. From bank transactions to document verification portals, from government official websites to social media, all these platforms share data remotely over web which contains certain confidential information and thus uses security mechanisms based on cryptographic algorithm to preserve the confidentiality of data preventing any information leakage or breach. Computing prime factors of these large numbers is compute intensive task where the serial programming makes RSA algorithm to slow down.

As per the literature survey, public-key cryptography algorithms are computationally expensive .Modular exponential and modular multiplications requires massive computations. RSA, one of the known public key algorithm despite of the mathematically soundness is not efficient and secure. RSA algorithm has a trade-off between security and efficiency. To have secure RSA, one require large key size and large key size means more computations, making RSA slow. If efficient RSA with small key size is used, it will make RSA algorithm vulnerable to many security attacks.

3. PARALLEL COMPUTING

Parallel programming (Quinn, 2003) can be used to improve the efficiency and performance of the applications on multi-core processing machines through concurrent execution of instructions provided by parallel program. For the last 30-40 years, parallel programming has been used in the area of High Performance Computing (HPC). In this area, large complex problems are decomposed into the smaller parts, and then these parts are executed concurrently. Thereafter, the results obtained from all parts are combined to get the final result.

Therefore to achieve higher performance in the area of security, the security algorithms can be implemented in parallel (Gilles, 1974) such that they can be executed on multi core processor to increase the speed and efficiency. By parallelizing these algorithms the power consumption can also be reduced and high performance can be achieved in terms of energy as well.

3.1 OpenMP

OpenMP "Open multiprocessing" is API for application that uses shared memory. It is portable, user friendly and efficient. It was first released in 1957. It is an API for writing multithreaded applications. It is a set of compiler directives and library routines for parallel application programmers. Using OpenMP, the programmer can write code that will be able to use all cores of a multicore computer and will be able to run faster if the number of cores are increased. It is used in shared memory architecture. This is a block -structured approach introduced for concurrency.

It operates on fork and join model of parallel execution as shown in figure 3.1. All OpenMP programs started as a single process called as master thread. It executes sequentially until a parallel region encountered. At this point the master thread "forks" into a number of parallel worker threads. The instructions in the parallel region are then executed by this team of worker threads. At the end of the parallel region, the threads synchronise and join to become the single master thread again. Parallelisation with OpenMP is specified through compiler directives which are embedded in the source code.

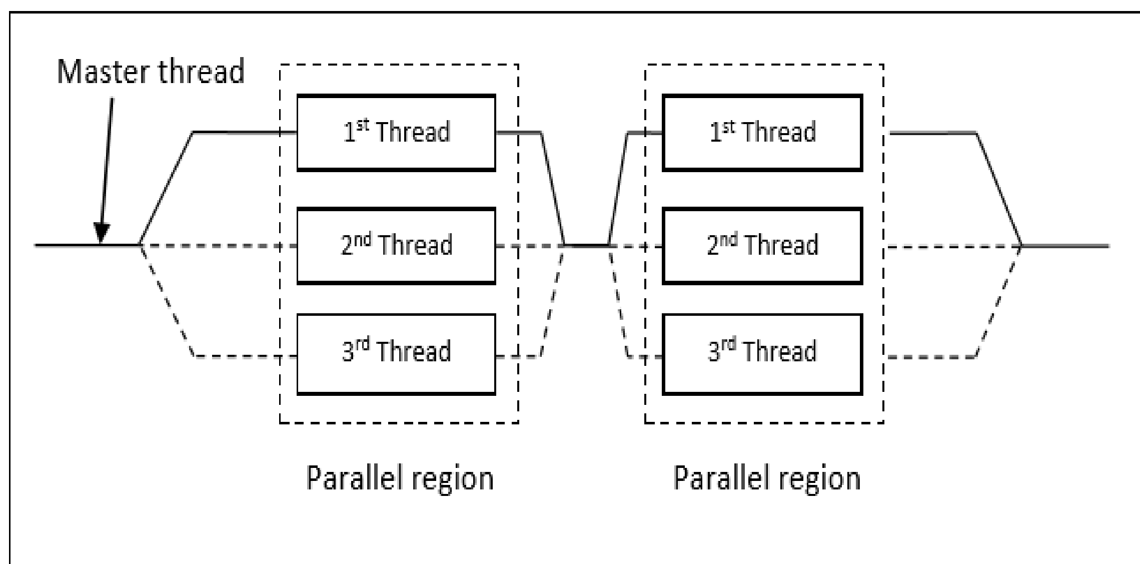


Figure 3.1.Fork and join model

4. EXPERIMENTATION METHODOLOGY

The OpenMP API defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. There are three kinds of directives: parallelism/work sharing, data environment, and synchronization. In C and C++, the directives are implemented as #pragma statements,

and in Fortran 77 and 90 they are implemented as comments. OpenMP is based on a fork-join model of parallel execution.

The sequential code sections are executed by a single thread, called the master thread. The parallel code sections are executed by all threads, including the master thread. The fundamental directive for expressing parallelism is the parallel directive. It defines a parallel region of the program that is executed by multiple threads. All of the threads perform the same computation, unless a work sharing directive is specified within the parallel region. Work sharing directives, such as for, divide the computation among the threads. For example, the _for‘ directive specifies that the iterations of the associated loop should be divided among the threads so that each iteration is performed by a single thread. The _for‘ directive can take a schedule clause that specifies the details of the assignment of iterations to threads. Schedules can specify assignments such as round-robin or block.

OpenMP also defines shorthand forms for specifying a parallel region containing a single work sharing directive. For example, the parallel for directive is shorthand for a parallel region that contains a single for directive. The data environment directives control the sharing of program variables that are defined outside the scope of a parallel region. The data environment directives include: shared, private, firstprivate, reduction and threadprivate. Each directive is followed by a list of variables. Variables default to shared, which means shared among all the threads in a parallel region. A private variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. A firstprivate variable has the same attributes as a private variable except that the private copies are initialized to the variables at the time the parallel region is entered. The reduction directive identifies variables. Finally, OpenMP provides the threadprivate directive for named common blocks in Fortran and global variables in C and C++. Threadprivate variables are private to each thread, but they are global in the sense that OpenMP also defines shorthand forms for specifying a parallel region containing a single work sharing directive. For example, the parallel for directive is shorthand for a parallel region that contains a single for directive. The data environment directives control the sharing of program variables that are defined outside the scope of a parallel region. The data environment directives include: shared, private, firstprivate, reduction and threadprivate. Each directive is followed by a list of variables. Variables default to shared, which means shared among all the threads in a parallel region. A private variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. A firstprivate variable has the same attributes as a private variable except that the private copies are initialized to the variables at the time the parallel region is entered. The reduction directive identifies variables. Finally, OpenMP provides the threadprivate directive for named common blocks in Fortran and global variables in C and C++.

4.1.Software Tools

Tools Used for the Parallelization

Following are the tools that are be used for parallelization –

4.1.1. Linux Platform

Linux (Bovet and Cesati, 2005) is an UNIX –like open source operating system that is used throughout the world. It is system software specially designed to access and instructs various hardware devices available with the target machine.

4.1.2. GCC Infrastructure

GCC stands for GNUs Compiler collection (Bovet and Cesati, 2005) which is a compiler system introduced by GNU project. It is a collection of compilers for C, C++, Java, Ada, etc. For the experimentations of this research the GCC infrastructure is used heavily with the combination of Linux Platform to test the proposed algorithms for worldwide agreement.

S.NO	Components	Description
1.	Processor	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (8 CPUs), ~1.8GHz
2.	Frequency	1.60 GHz
3.	RAM	8192MB
4.	System Type	64 Bit Operating System
5.	Environment	Ubuntu Linux
6.	Platform	GCC Infrastructure
7.	Operating System	Ubuntu 18.04.4 LTS
8.	Compiler	GCC 7.5.0
9.	API	OpenMP 4.5

Table 4.1.1: Configuration of the Computer System used for Experiment

5.PERFORMANCE AND EXPERIMENTAL OUTCOMES

5.1 Experimental Result

5.1.1 Features of the experiment

Threads	First Execution Time	Second Execution Time	Third Execution Time	Fourth Execution Time	Average Execution Time	Speed	Efficiency
1	0.20	0.17	0.22	0.17	0.19	1.05	1.05
2	0.19	0.22	0.18	0.19	0.19	1.05	0.52
3	0.16	0.19	0.22	0.21	0.19	1.05	0.35
4	0.15	0.16	0.20	0.19	0.17	1.17	0.29
5	0.21	0.18	0.20	0.22	0.17	1.17	0.23
6	0.20	0.13	0.16	0.18	0.16	1.25	0.20

Table 5.1.1 Execution Cases Of RSA Algorithm

5.1.2 Behaviour of the Parameters

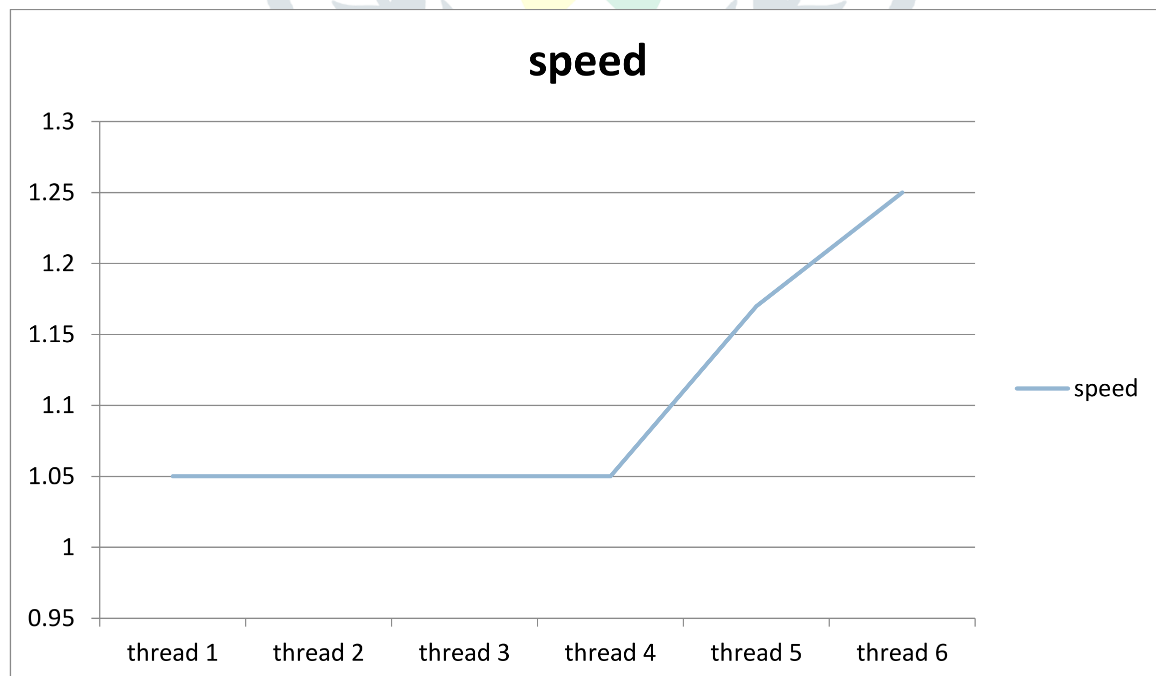


Figure 5.1.1 Speed up Graph For Parallelized RSA Algorithm

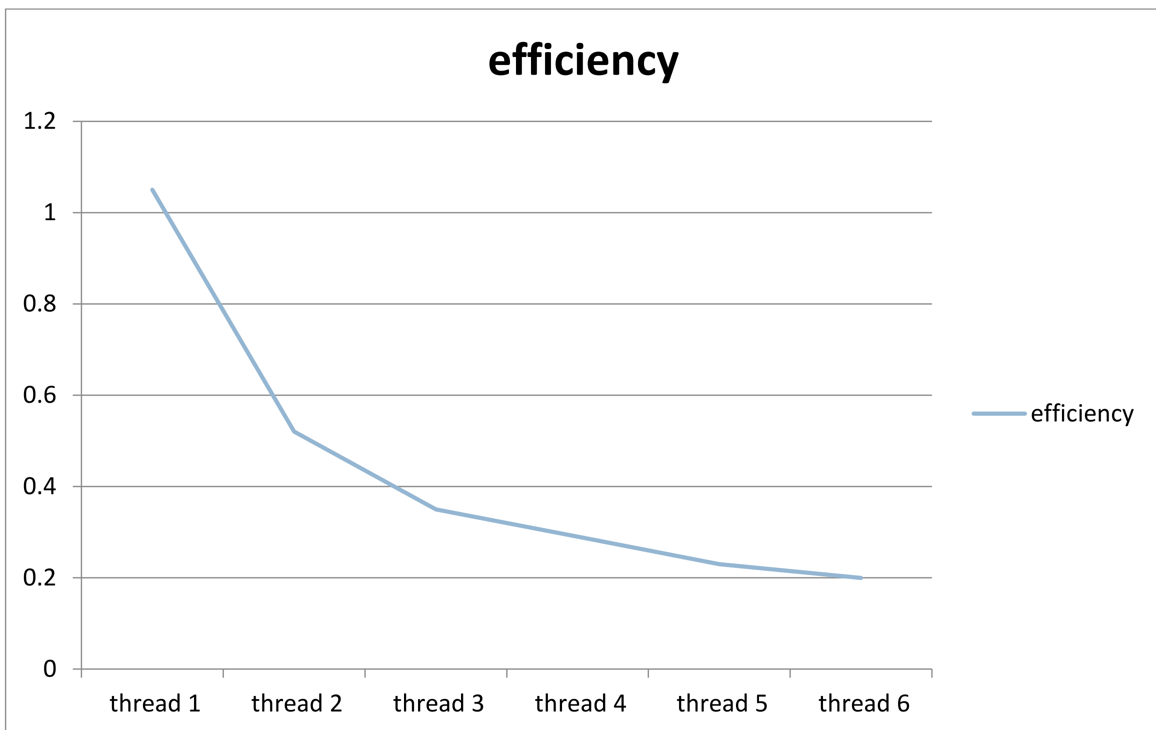


Figure 5.1.2 Efficiency Graph For Parallelized RSA Algorithm

5.1.3 Demonstration

```
real    5m16.418s
user    5m16.298s
sys     0m0.004s
user@user-HP-Laptop-15-bs1xx:~/Desktop/test2$
```

Figure 5.1.3 Run Time Of Serial Code

```
user@user-HP-Laptop-15-bs1xx:~/Desktop/test2$ g++ -ftime-report rsa1.cpp -o rsa1.exe

Execution times (seconds)
 phase setup             :   0.01 ( 3%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall    1495 kB ( 3%) ggc
 phase parsing           :   0.31 (84%) usr   0.16 (100%) sys   0.49 (89%) wall   40507 kB (87%) ggc
 phase lang. deferred    :   0.04 (11%) usr   0.00 ( 0%) sys   0.03 ( 5%) wall    3946 kB ( 8%) ggc
 phase opt and generate  :   0.01 ( 3%) usr   0.00 ( 0%) sys   0.02 ( 4%) wall     839 kB ( 2%) ggc
 |name lookup            :   0.06 (16%) usr   0.01 ( 6%) sys   0.11 (20%) wall    2590 kB ( 6%) ggc
 |overload resolution    :   0.06 (16%) usr   0.03 (19%) sys   0.07 (13%) wall    5502 kB (12%) ggc
 callgraph construction  :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall     239 kB ( 1%) ggc
 preprocessing           :   0.06 (16%) usr   0.04 (25%) sys   0.12 (22%) wall    3557 kB ( 8%) ggc
 parser (global)         :   0.06 (16%) usr   0.07 (44%) sys   0.16 (29%) wall   14340 kB (31%) ggc
 parser struct body      :   0.03 ( 8%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall    5728 kB (12%) ggc
 parser function body    :   0.03 ( 8%) usr   0.03 (19%) sys   0.04 ( 7%) wall    2370 kB ( 5%) ggc
 parser inl. func. body  :   0.02 ( 5%) usr   0.00 ( 0%) sys   0.04 ( 7%) wall    1180 kB ( 3%) ggc
 parser inl. meth. body  :   0.06 (16%) usr   0.01 ( 6%) sys   0.04 ( 7%) wall    2697 kB ( 6%) ggc
 template instantiation  :   0.09 (24%) usr   0.01 ( 6%) sys   0.11 (20%) wall   14507 kB (31%) ggc
 dominance computation   :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall       0 kB ( 0%) ggc
 initialize rtl          :   0.01 ( 3%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      12 kB ( 0%) ggc
 TOTAL                 :   0.37             0.16             0.55             46798 kB
user@user-HP-Laptop-15-bs1xx:~/Desktop/test2$
```

Figure 5.1.4 Execution Time Of Serial Code

```
an 0 0 0 0 0 -nan 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 -nan 0 0 0 0 0 0 -nan 0 0 -nan 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -nan 0 0 0 0 0 0 0 0 0 0 0 0 -n
n 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -n
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -nan 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -nan 0 0 0 -nan 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 -nan -nan -nan 0 0 0 0 -nan 0 0 0 0 -nan 0 0 0 0 0 0 0 0 0

Total Runtime: 0.378864

real    0m0.409s
user    0m2.199s
sys     0m0.072s
user@user-HP-Laptop-15-bs1xx:~/Desktop/test7$
```

Figure 5.1.5 Run Time Of Parallel Code

```
user@user-HP-Laptop-15-bs1xx:~/Desktop/test7$ g++ -ftime-report -fopenmp rsa1parallel.cpp -o rsa1parallel.exe

Execution times (seconds)
 phase setup             :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall    1565 kB ( 3%) ggc
 phase parsing           :   0.32 (89%) usr   0.18 (95%) sys   0.49 (89%) wall   40742 kB (86%) ggc
 phase lang. deferred    :   0.02 ( 6%) usr   0.01 ( 5%) sys   0.03 ( 5%) wall    3944 kB ( 8%) ggc
 phase opt and generate  :   0.02 ( 6%) usr   0.00 ( 0%) sys   0.02 ( 4%) wall    1136 kB ( 2%) ggc
 |name lookup            :   0.07 (19%) usr   0.04 (21%) sys   0.09 (16%) wall    2618 kB ( 6%) ggc
 |overload resolution    :   0.02 ( 6%) usr   0.01 ( 5%) sys   0.06 (11%) wall    5513 kB (12%) ggc
 callgraph construction  :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall     284 kB ( 1%) ggc
 preprocessing           :   0.07 (19%) usr   0.05 (26%) sys   0.13 (24%) wall    3590 kB ( 8%) ggc
 parser (global)         :   0.09 (25%) usr   0.07 (37%) sys   0.13 (24%) wall   14882 kB (31%) ggc
 parser struct body      :   0.05 (14%) usr   0.00 ( 0%) sys   0.06 (11%) wall    5669 kB (12%) ggc
 parser enumerator list  :   0.00 ( 0%) usr   0.01 ( 5%) sys   0.00 ( 0%) wall      73 kB ( 0%) ggc
 parser function body    :   0.04 (11%) usr   0.01 ( 5%) sys   0.02 ( 4%) wall    2204 kB ( 5%) ggc
 parser inl. func. body  :   0.00 ( 0%) usr   0.01 ( 5%) sys   0.01 ( 2%) wall    1173 kB ( 2%) ggc
 parser inl. meth. body  :   0.04 (11%) usr   0.01 ( 5%) sys   0.03 ( 5%) wall    2562 kB ( 5%) ggc
 template instantiation  :   0.05 (14%) usr   0.03 (16%) sys   0.13 (24%) wall   14530 kB (31%) ggc
 tree eh                 :   0.01 ( 3%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      11 kB ( 0%) ggc
 varconst                :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall       0 kB ( 0%) ggc
 final                   :   0.01 ( 3%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      32 kB ( 0%) ggc
 initialize rtl          :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 2%) wall      12 kB ( 0%) ggc
 TOTAL                 :   0.36             0.19             0.55             47399 kB
user@user-HP-Laptop-15-bs1xx:~/Desktop/test7$
```

Figure 5.1.6 Execution Time Of Parallel Code

6.CONCLUSION

This paper is based on the conclusions drawn by overall objectives of this research work which includes the development of time efficient and energy efficient parallel PKC based algorithms.This research has been started with the need to design parallel algorithms for the existing public-key infrastructure based security algorithms. The need arose because the public-key based algorithms are compute intensive and take a lot of time and energy to execute. Parallel programming may be one of the solutions to overcome such complications that can be used to make the PKC based algorithms more applicable to mobile devices such as smart phones, tablets, etc.

Therefore the main focus of this research was on the development of parallel algorithms that can be used to speed up the process of encryption and decryption as well as to speed up digital signing so that both the time and energy involved in these processes can be reduced.

7.REFERENCES

[1]. Parallel Computing using OpenMP , Ms. Ashwini M. Bhugul MMCOE, Pune-52, IJCSMC, Vol. 6, Issue. 2, February 2017, pg.90 – 94.

[2]. Design and Implementation of an Improved RSA Algorithm ,Yunfei Li School of Information Science and Engineering Yunnan University Kunming, China ,Qing Liu, Tong Li National Pilot School of Software Yunnan University Kunming, China, 2010 International Conference on E-Health Networking, Digital Ecosystems and Technologies.

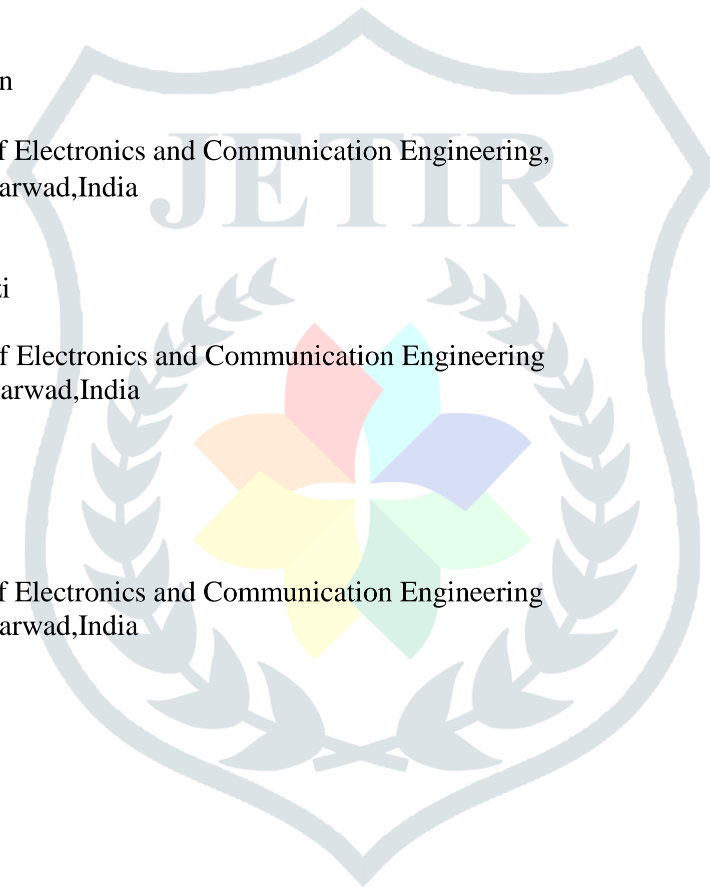
[3]. Comparative Analysis of Sequential and Parallel Implementations of RSA Sapna Saxena, Neha Kishore, Disha Handa, Bhanu Kapoor, International Journal of Scientific & Engineering Research, Volume 4, Issue 8, August-2013.

[4]. An Enhanced Parallel Version of RSA Public Key Crypto Based Algorithm Using OpenMP, Rahul Saxena,Monika Jain, Dushyant Singh, Ashutosh Kushwah, Manipal University Jaipur, India.

BIOGRAPHIES

Aishwarya S Bingeri
Student
Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India

Akshata Sajjan
Student
Department of Electronics and Communication Engineering,
SDMCET,Dharwad,India



Muskan Khazi
Student
Department of Electronics and Communication Engineering
SDMCET,Dharwad,India

Nisha M Patil
Student
Department of Electronics and Communication Engineering
SDMCET,Dharwad,India