Comparative Analysis of Sequential and Parallel Implementations of RSA

Sapna Saxena, Neha Kishore, Disha Handa, Bhanu Kapoor

Abstract—Public key infrastructure based cryptographic algorithms are usually based on modular arithmetic. As a result, they are considered to be slower when compared to the symmetric cryptographic algorithms. In the RSA public key security algorithm, the encryption and decryption is based on modular exponentiation and modular reduction using large integers. Due to the size of integers used in the RSA, typically 1024 bits, the algorithm becomes compute-intensive. Thus the sequential implementation of RSA takes large runtimes. In this paper, we are looking into the possibility of improving the performance of RSA by parallelizing it using OpenMP on the GCC infrastructure. We have developed two versions of the parallel RSA for our experiments. We have also analyzed the performance gained by comparing the sequential version with the parallel versions of RSA running on the GCC infrastructure.

Index Terms— Cryptography, GCC infrastructure, Public Key algorithm, Parallel implementation, Serial Implementation, RSA, OpenMP

1 INTRODUCTION

THE concept of public key cryptography (PKC) was invented and introduced by Whitfield Diffie and Martin Hellman [1, 8], and independently by Ralph Merkle [7]. 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 [12] 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.

- *Sapna Saxena is currently pursuing PhD inComputer Science and Engineering inChitkara University,India, PH- 9315870921. E-mail:sapna.saxena@chitkarauniversity.edu.in*
- *Neha Kishore is currently pursuing PhD inComputer Science and Engineering inChitkara University,India, PH- 9592405665. E-mail:neha.kishore@chitkarauniversity.edu.in*
- *Disha Handa is currently pursuing PhD inComputer Science and Engineering inChitkara University,India, PH- 9417994228. E-mail:disha.handa@chitkarauniversity.edu.in*
- *Bhanu Kapoor Professor,Computer Science and Engineering inChitkara University,India, E-mail:bhanu.kapoor@chitkarauniversity.edu.in*
- *Co-Author name is currently pursuing masters degree program in electric*

2 RSA ALGORHTM

The RSA algorithm [3, 6, 9] 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.

2.1 Key Generation

The key generation part of RSA algorithm is muli-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 φ(n) = (p-1) (q-1)
4. Choose an integer e, 1 < e < φ(n) such that: GCD(e, φ(n)) = 1(GCD is greatest common denominator)
5. Compute d, 1 < d < φ(n) such that: ed ≡ 1 (mod φ(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.

2.2 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

2.3 RSA Decryption

In order to encrypt, 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 = Ce % m

3 PARALLELIZATION OF RSA ALGORITHM

The main focus of this paper is to design new parallel algorithms that provide efficient parallel implementations of RSA to be executed on multi-core machine and compare the performance gained by the parallel implementation.

The RSA algorithm is based on modular arithmetic. While executing the algorithm, most of the time is consumed during the encryption / decryption part and the majority of this time is consumed in modular exponentiation and modular reduction. We have implemented various parallel algorithms for exponentiation and reduction.

The methods that are implemented, analyzed and compared in this paper are the memory efficient methods of modular exponentiation and modular reduction. In order to parallelize the RSA and to execute it on multi-core machines, we have implemented it in two forms analyzed the performance gained in terms of time.

The two methods which have been implemented are –

1. The repeated-square and multiply method, and
2. The right-to-left binary method.

3.1 First Form: Repeated-square and multiply method

The first form of parallel implementation is based on the repeated square-and-multiply method [2, 10]. The repeated squa gorithm is ba

T ow –

$$ModExp(g, e, m) = \begin{cases} 1, & if\ e = 0 \\ (g \times ModExp(g, (e-1), m)) \bmod m, & if\ e\ is\ odd \\ ModExp(g, e/2, m)^2 \bmod m, & if\ e\ is\ even \end{cases}$$

This method improves the performance to a great extent for larger e. The principle used to implement this form is data decomposition. We have parallelized the modular exponentiation part of RSA in the following manner –

If the exponent e is even then it can be divided into two or four parts as per the availability of the cores in the system. Then each part of the exponentiation computation can be assigned to multiple threads and finally the results of each thread can multiplied to get the final result.

If the e exponent is odd, then the following can be done –

- The value of g can be stored in one variable say A.
- We then subtract 1 from the e to make it even.
- e will be divided into 2/4 parts depending upon the number of cores present in the target machine.
- Each exponentiation computation will be allocated to a separate thread and processor.
- The result obtained from each thread will be multiplied to get the combined result.
- Finally the result will be multiplied to A to get the final result.

The algorithm used for the method is given in Figure 1.

```
Result=1
N=Power/Number_of_Thread
IF Power mod Number_of_Thread == 0
    FOR I = 1 to Power
      FOR J = 1 to N
        Result = Result * Base;
      END FOR
    END FOR
ELSE
    N = N - 1
    FOR I = 1 to Power
      FOR J = 1 to N
        Result = Result * Base;
      END FOR
    END FOR
    Result = Result * Base;
END IF
Cipher = Result mod Modulus
```

Fig 1: Algorithm for repeated-square and multiply

3.2 Second Form: Right-to-Left Binary method

The second form of parallel implementation is based on right-to-left binary method which is based on the principle of exponentiation by squaring or binary exponentiation. The method is called right-to-left binary method because the binary representation of the exponent is computed from right to left. Firstly the exponent is converted into its binary representation and the right most bit is considered first. Thereafter the algorithm given in Figure 2 is used to calculate the value of expression Base exponent mod modulus.

```
Function power_binary(base, BinaryNumber, length, modulus)
Result =1
  FOR i = length to 0
    IF binaryNumber[i] % 2 == 1
       Result=(result * base)%modulus;
    END IF
    base = (base * base) % modulus;
  END FOR
return Result
```

Fig 2: Algorithm for right-to-left binary method

The loop used in the algorithm executed for the number of times equal to the number of bits present in the binary notation of the exponent. The calculation performed on the following principle –

$$\prod_{i=0}^{n-1} \left(b^{2^i}\right)^{a_i} \pmod m$$

For example base = 4, exponent = 13, and modulus = 497. The binary equivalent of exponent is 1101. Because exponent is four binary digits in length, the loop executes only four times. This is shown in Table 1 below.

TABLE 1

EXAMPLE OF CALCULATIONS

nth Iteration	Bit	Value of Result	Value of Base
1	1	(1*4) % 497 = 4	(4*4) % 497 = 16
2	0	No calculation, Value will remain same	(16*16) % 497 = 256
3	1	(4*256) % 497 = 30	(256*256) % 497 = 429
4	1	(30*429) % 497 = 445	(429*429) % 497 = 151

The loop then terminates since exponent is zero, and the result 445 is returned.

4 METHODOLOGY USED FOR EXPERIMENTS

The experiments are performed on dual core computers using OpenMP [12] on the GCC infrastructure on Linux environment. The OpenMP API is a portable and parallel programming model for shared memory multiprocessor architectures. It is an open-source programming interface that supports parallel programming using the concepts of multi-threaded programming. The OpenMP API supports C/C++ and Fortran on a wide variety of architectures. It is embedded within the programs using compiler directives. The programs are scalable and can be easily executed on any multi-core machine either quad-core or dual quad-core and so on.

For the experiments three different forms of RSA implementation are developed, first is the sequential one and other two are the parallel forms which are based on two popular memory efficient methods used for modular exponentiation and modular reduction which are repeated-square-and-multiply method and right-to-left binary method respectively. And the significant improvements are recorded after performing the experiments.

The sequential form is developed using C Language and executed on GCC infrastructure on Linux platform. To develop the parallel versions OpenMP API is used in the combination with C language on the GCC infrastructure. It requires big integer based computations [4] to implement the algorithm. The performance gained is measured and analyzed in terms of time during the experimentation. The time taken during the execution of various forms is measured using time utility of Linux. All forms are executed for exactly 25 times using the same set of message and average of the time is taken as the final value for that form.

All forms of the RSA implementation, sequential as well as the parallel, which are used in the experiments, are divided into three parts-Key Generation, Encryption, and Decryption.

To start with the process of key generation, same set of prime numbers is taken in all three forms to generate the same set of private and public keys. Also the same message is used for encryption / decryption to detect the correct differences in terms of performance gained for all the forms.

5 COMPARATIVE RESULTS

The experiments performed in order to improvise the performance of RSA on multi-core machines shows the promising results and it is find obvious after getting the initial results that RSA encryption / decryption can be implemented fast on multi-core machines if implemented parallely.

OpenMP in combination with GCC infrastructure allows the parallel implementation that decreases the execution time of RSA and improves its performance. However, it is obvious that the performance will be increased significantly in the presence of more number of processors.

We have obtained to find some improvements, as listed in the table below. The Table 2 shows the execution time and performance comparison in parallel runtimes using 2 cores versus that of a sequential implementation using a single core. In order to improvise the performance of RSA encryption we have implemented it into two ways.

First form is based on the repeated-square-and-multiply method version which itself shows the significant improvement over the sequential implementation. It represents a 22.42% improvement in the execution runtime of the sequential one.

Whereas the second form of the parallel implementation is based on right-to-left binary method represents a 26.54% improvement in the execution runtime.

TABLE 2

COMPARISON BETWEEN THE SEQUENTIAL AND PARALLEL IMPLEMENTATION OF RSA

S. No	Type of Implementation	Implementation Based on	Time w.r.t. sequential / parallel execution
1.	Sequential Implementation	Serial implementation	4.640s
2.	First Form of Parallel Implementation	Repeated square and multiply method	3.700s
3.	Second Form of Parallel Implementation	Right-to-Left Binary Method	3.409s

The Fig 3 shows the graphical representation of the performance comparison between the sequential and parallel implementation of RSA.

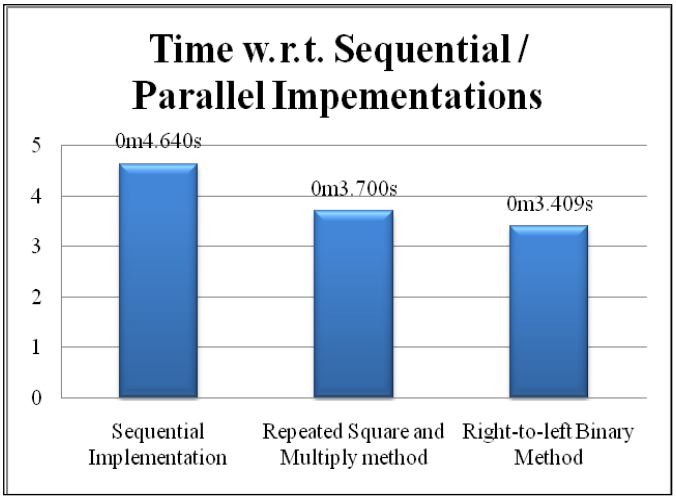

Fig 3: Comparison between the sequential and parallel implementation of RSA encryption

6 Conclusion and Futurescope

The experimental results show that the RSA can be improvised by implementing parallel using in combination with GCC infrastructure. The parallel versions of RSA are more efficient than that of the sequential version of it.

The programs used in the experiments are executed in dual core environment. They could be performed with larger number of cores and improving upon synchronization issues which will further improve the runtime. The experiments are purely based on the modular exponentiation part of the RSA. The further experiments of the research will be focused on the factorization part of the RSA key generation algorithm.

Moreover, these experiments are performed and tested in a single experimental environment. They could be performed in different environments and results can be compared in following research.

References

[1] Menezes, A. J., Vanstone, S. A., & Oorschot, P. C. V. 1996. Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA.

[2] Diego Viot, Rodolfo Aurelio, Helano Castro and Jardel Silveria, Modular Multiplication Algorithm for PKC, Universiadade Federal do Ceard, LESC

[3] Josef Pieprzyk1 and David Pointcheval, Parallel Authentication and Public key encryption, Springer-Verlag 2003

[4] Chandra, S. S. & Chandra, K. 2005. Cbigint class: an implementation of big integers in c++. J. Comput. Small Coll., 20(4), 77–83.

[5] Bewick, G. 1994. Fast multiplication algorithms and implementation.

[6] Rivest, R. L., Shamir, A., & Adleman, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 26(1), 96–99.

[7] Fu, C. & Zhu, Z.-L. Oct. 2008. An efficient implementation of rsa digital signature algorithm. In Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on, 1–4.

[8] Diffie, W. & Hellman, M. Nov 1976. New directions in cryptography. Information Theory, IEEE Transactions on, 22(6), 644–654.

[9] Barrett, P. 1986. Implementating the rivest, shamir and aldham public-key encryption algorithm on standard digital signal processor. Proceedings of CRYPTO'86, Lecture Notes in Computer Science, 311–323.

[10] Cohen, H., Frey, G. (editors): Handbook of elliptic and hyperelliptic curve cryptography. Discrete Math.Appl., Chapman & Hall/CRC (2006)

[11] Igor L. Markov, Mehdi Saeedi, "Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation", Quantum Information and Computation, Vol. 12, No. 5&6, pp. 0361-0394, 2012

[12] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel Programming in OpenMP. Morgan Kaufmann, 2000.