An Efficient Parallel Algorithm for Secured Data Communications Using RSA Public Key Cryptography Method

Sapna Saxena
Chitkara University
Himachal Pradesh, India
sapna.saxena@chitkarauniversity.edu.in

Bhanu Kapoor
Chitkara University
Himachal Pradesh, India
bhanu.kapoor@chitkarauniversity.edu.in

Abstract—**Public-key infrastructure based cryptographic algorithms are usually considered as slower than their corresponding symmetric key based algorithms due to their root in modular arithmetic. In the RSA public-key security algorithm, the encryption and decryption is entirely based on modular exponentiation and modular reduction which are performed on very large integers, typically 1024 bits. Due to this reason the sequential implementation of RSA becomes compute-intensive and takes lot of time and energy to execute. Moreover, it is very difficult to perform intense modular computations on very large integers because of the limitation in size of basic data types available with GCC infrastructure. In this paper, we are looking into the possibility of improving the performance of proposed parallel RSA algorithm by using two different techniques simultaneously, first implementing modular calculations on larger integers using GMP library and second by parallelizing it using OpenMP on the GCC infrastructure. We have also analyzed the performance gained by comparing the sequential version with the parallel versions of RSA running on the GCC infrastructure.**

Keywords-***Public key algorithm; parallel computation; RSA; GMP Library; Open MP; GCC infrastructure;***

I. INTRODUCTION

Securing data in communication is a major challenge in the modern era of Internet. Various techniques have been introduced by different computer scientists to communicate data securely across the open communication channels, for example secret key cryptography, hashing for data integrity checks etc. One of the most important techniques is public-key cryptography (PKC) or asymmetric cryptography which was invented by Whitfield Diffie, Martin Hellman [2] and Ralph Merkle [3]. They proposed that to provide more secured data communication instead of single secret key, pair of key can be used i.e. an encryption key and a decryption key and the decryption key cannot (practically) be derived from the encryption key [1, 2]. An encryption key, called the public key, is revealed and the decryption key, called the private key, is kept private. The PKC methods are important for scenarios where the communicators cannot share a secret key due to the presence of open communication channels.

Public-key encryption and decryption is complicated and compute-intensive nature of modular multiplications with very large numbers which are needed to perform these tasks. In PKC-based algorithms, for example the RSA algorithm, the keys used have a very large size of usually 1024 bits or more. Moreover, the RSA encryption and decryption is based on repeated modular exponentiation and modular reduction of very large numbers which make it compute-intensive and energy-intensive as well. Therefore, the public-key algorithms are much slower than the symmetric key algorithms but do provide stronger security.

RSA [4] is one of the most important PKC-based algorithms which is widely used for the cryptographic purposes. RSA is based on the factorization technique that given a very large number, for example a number which is composed of 1024 bits or more, it is almost impractical to find the two prime numbers whose product will be the given number. But it is quite impossible to work on such large numbers on GCC infrastructure directly. Therefore, it is vital to have some third party software that may assist in manipulating large numbers on GCC infrastructure. GNU provides an excellent library called GNU's Multi Precision Library – GMP [5] to handle very large number up to 2048 bits having arbitrary precision.

Recently, the use of OpenMP [6] 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 OpenMP parallel API. 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.

II. RSA ALGORITHM

The RSA algorithm [4, 7, 8] was invented in 1977 by the group of three cryptographers and it is named after their names Rivest, Shamir, and Adleman. It is one of the most important algorithms used for both encrypting and authenticating the data while transmitting it on the Internet. It was introduced as the first trustworthy algorithm suitable for both digital signature and data encryption applications. It is widely used in the protocols supporting the e-commerce today.

RSA is based on the factorization of very large numbers, i.e. given a very large number it is not practical to find the two prime numbers whose product is given number because it will take a very long time even with the best known

978-1-4799-2572-8/14/$31.00

algorithms for factorization. It provides strong security subject to if provides with sufficiently long keys. For example, if the key length of 1024 bits or more is used then it is nearly impractical to break up the security of RSA encryption even when working with high performance computers.

Generally, the RSA algorithm is divided into three parts – Key Generation, Encryption, and Decryption.

A. Key Generation

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

1) Select two very large random prime integers p and q having bit size at least 512
2) Compute m = p*q, which is used a modulus
3) Compute $\varphi(n) = (p-1)(q-1)$
4) Select 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 d and n are kept as the private key.

B. 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.

$$C = Me \% m$$

C. 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.

$$M = Ce \% m$$

III. Parallelization of RSA

The main focus of this paper is to design new parallel algorithm that provide efficient parallel implementation 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 parallel algorithm for modular exponentiation and reduction routines using two facilities –

1) *GNU's Multi Precision (GNU MP) Library* - Using the functions offered by this library to implement the key generation, encryption and decryption routines.
2) *OpenMP API* – OpenMP APIs are used to parallelize the algorithm to obtain higher performance. OpenMP code is compiled through GNU's GCC C++ compiler which leads to automatic creation and management of parallel threads.

A. Fast Modular Computations using GNU MP Library

Any RSA implementation involves intensive modular exponentiation and modular reduction computations on very large numbers which poses lot of complications while performing. One solution for this problem is to design a whole new library to work with large integers. But the PKC technique which we are proposing through this paper is targeted to implement diversified architectures present on Internet. Therefore the solution should be capable of handling all the complications arise during the cross-platform computations over large integers. One Library provided by GNU is already present which is capable of handling all such complications.

GNU MP Library is an excellent portable library provided by GNU to be used with GCC infrastructure. GNU MP Library is comprised of multiple portable functions written in C language for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It was developed to provide the fastest possible arithmetic for all those applications that require computations on vary large numbers having higher precision. Because it is written in C language it is directly supported by the basic C types.

GNU MP Library is designed in such a manner that it can provide good performance both in the case of small precision numbers (few hundred bits) as well as large precision numbers which use thousands of bits. The speed of GMP is achieved by using "fullword" as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs such as Intel, AMD, etc., and by a general emphasis on speed [5]. We have used the facilities offered by the GMP library heavily throughout our application. The key generation, encryption and decryption routines all use the integer handling functions offered by this library.

B. Fast Parallelization using OpenMP

To implement new parallel algorithm on GCC infrastructure, the OpenMP API is used for the experiments. The OpenMP API is chosen for the experiments because it is an excellent potable parallel programming model that can be implemented on shared memory architecture to achieve high parallel capabilities. 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.

IV. Methodology

The method that is used to implement Parallel RSA on multi-core machines is the repeated-square and multiply method [9, 10]. The performance gained by the Parallel RSA is analyzed in terms of the time conserved during modular exponentiation and modular reduction routines. The repeated square-and-multiply modular exponentiation [11, 12]

algorithm is based on the simple observation that for an even e,

$g^e \bmod m = (g^{e/2} * g^{e/2}) \bmod m$.

The recursive definition for the repeated square-and-multiply method is described in Figure 1.

ModExp(g,e,m) =	**1,**	**if e = 0**
	(g X ModExp(g, (e-1), m)) mod m,	**if e is odd**
	ModExp(g, e/2, m)^2 mod m,	**if e is even**

Figure 1. repeated-square and multiply method

1) Result=1
2) N=Power/Number_Of_Cores
3) Divide the whole modular exponentiation and reduction into N parts
4) For each part execute step 5 PARALLELY
5) IF Power mod Number_of_Cores == 0
   FOR I = 1 to N
   Result = Result * Base;
   END FOR
   ELSE
   FOR I = 1 to N
   Result = Result * Base;
   END FOR
   Result = Result * Base;
   END IF
6) Cipher = Result MOD Modulus

Figure 2. Proposed Parallel RSA Algorithm using repeated-square and multiply method

This method can be used to improve the performance of the algorithm for the larger e such as 1024 bits or higher. The method used in designing and implementing new parallel RSA algorithm makes use of data decomposition which is discussed in this paper. We have parallelized the modular exponentiation part of RSA algorithm in the following fashion:

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

The proposed algorithm for parallel RSA is given in Figure 2.

The parallel RSA algorithm, which is used in the experiments, is divided into three parts-Key Generation, Encryption, and Decryption. The parallel RSA algorithm is based on repeated square-and-multiply method to implement modular exponentiation and modular reduction. And the significant improvements are recorded after performing the experiments. 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.

V. PERFORMANCE ANALYSIS

The experiments are performed on dual quad core computer using OpenMP [12] on the GCC infrastructure in the presence of GNU's MP Library on Linux environment.

The experiments are performed on following platform:

Processor: AMD FX(tm)-8120 Eight-Core Processor 3.10 GHz
RAM: 4.00 GB
O.S.: Ubuntu Linux 11.04
Platform: GCC Infrastructure

The experiments performed in order to improve the performance of RSA on multi-core machines shows some encouraging results. We have used OpenMP in combination with GCC infrastructure and GNU's MP Library to implement parallel RSA that decreases the execution time and improves the performance of the algorithm. The time taken during the execution of different test cases is measured using time utility of Linux.

In order to find the improvement in the performance of encryption and decryption performed by Parallel RSA over the sequential RSA, we have executed it on dual core, quad core, 6 cores and dual quad core computers. Each experiment is performed repeatedly for exactly 25 times and average of the time is taken as the final value for that test case. The time is measured in terms of three distinct categories – encryption time, decryption time and overall time. Encryption time is the time measured during the encryption process, decryption time is the time taken during decrption and overall time includes key generation time, encrryption time as well as decryption time.

For the expriments we have used two different set of test cases to test and prove the strength of our version of parallel RSA which are as follows –

1) *Test Case Set 1* – Under first set of test cases range of experiments are performed by taking different key sizes varying from 128 bits to 2048 bits as shown in Table 1. For each key size same set of message with 5000 characters is taken for encryption and decryption to find the exact difference between the execution time of serial RSA and parallel RSA. The results obtained from test case set 1 are given in the table1.

2) *Test Case Set 2* – Under second set of test cases experiments are performed by taking fixed key size of 1024 bits and message of variable sizes from 1000 characters to 10,000 characters as shown in Table 2. Each test case is again performed on 2 cores, 4 cores, 6 cores and 8 cores machine repeatedly for 25 times and the average of all readings is taken as the final time. The results obtained from test case set 2 are given in table2.

The Table 1 and Table 2 shows the improvements and performance comparison in parallel runtimes using 2 cores, 4 cores, 6 cores and 8 cores versus that of a sequential implementation using a single core for different set of test cases. It is shown in the tables that the speedup increased when we increase the number of cores to execute the parallel RSA. Similarly Parallel RSA gives encouraging results for the test cases when the key size is taken sufficiently large

such as 2048 bits and the message size is increasd from 5,000 to 10,000 characters. In such cases we get approximately 5X speedup when Parallel RSA executed on 8 core machine as compared to the serial version of the RSA executed on single core.

VI. Conclusion and Futurescope

The experimental results show that the parallel RSA gives the improved results using OpenMP in combination with GCC infrastructure and GNU's MP Library. The parallel RSA are more efficient than that of the sequential version of it in terms of time and energy.

The programs used in the experiments are executed in dual quad core environment which are based on repeated square-and-multiply method. They could be performed with other modular exponentiation methods 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] Diffie, W. & Hellman, M. Nov 1976. New directions in cryptography. Information Theory, IEEE Transactions on, 22(6), 644–654.

[3] 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.

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

[5] The GNU Multiple Precision Arithmetic Library, Edition 5.1.0, 18 December 2012

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

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

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

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

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

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

[12] 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

TABLE I. Test Case – Set 1: Comaparative results obtained using fixed message size but varied key sizes

S. No	Key Size (in Bits)	Program Segment Type	Time (in seconds) taken w.r.t. the serial and parallel execution of the same code on varied number of cores					
			Serial Code (Core 1)	*2 cores*	*4 cores*	*6 cores*	*8 cores*	*Speedup on 8 cores w.r.t. to sequential code*
1	128	Encryption Time	0.01025	0.00637	0.00457	0.00486	0.00423	2.43x
		Decryption Time	0.06553	0.0478	0.02566	0.0137	0.01399	4.69x
		Overall Time	0.07928	0.05769	0.03371	0.0221	0.02252	3.53x
2	256	Encryption Time	0.02001	0.01558	0.00889	0.00583	0.00987	2.03x
		Decryption Time	0.29416	0.19298	0.09827	0.07153	0.06138	4.8x
		Overall Time	0.31847	0.21277	0.11134	0.08158	0.0751	4.25x
3	512	Encryption Time	0.04934	0.04435	0.02476	0.01474	0.01247	3.96x
		Decryption Time	1.37108	1.0708	0.51514	0.36241	0.30385	4.52x
		Overall Time	1.4267	1.12223	0.54684	0.38409	0.32289	4.42x
4	768	Encryption Time	0.1126	0.07945	0.03794	0.0296	0.02361	4.77x
		Decryption Time	4.81935	3.22759	1.5884	1.0948	1.19792	4.03x
		Overall Time	4.95449	3.32597	1.64481	1.14298	1.23753	4.01x
5	1024	Encryption Time	0.18261	0.01194	0.06308	0.04888	0.03982	4.59x
		Decryption Time	10.988	7.27724	3.59786	2.47073	2.10255	5.23x
		Overall Time	11.23602	7.4667	3.72418	2.5825	2.19647	5.12x
6	1280	Encryption Time	0.2665	0.18026	0.09486	0.06462	0.05506	4.85x
		Decryption Time	20.12456	13.2836	6.56407	4.55322	3.82693	5.26x
		Overall Time	20.47295	13.5461	6.74019	4.69844	3.96789	5.16x

TABLE II. TEST CASE – SET 2: COMAPARATIVE RESULTS OBTAINED USING FIXED KEY SIZE BUT VARIED MESSAGE SIZES

S. No	Message size (In characters)	Program Segment Type	Time (in seconds) taken taken w.r.t. the serial and parallel execution of the same code on varied number of cores					
			Sequential Code (Core 1)	*2 cores*	*4 cores*	*6 cores*	*8 cores*	*Speedup on 8 cores w.r.t. to sequential code*
1	1000	Encryption Time	0.03016	0.02689	0.01627	0.01080	0.01039	2.91x
		Decryption Time	1.87211	1.44590	0.72740	0.50758	0.41880	4.48x
		Overall Time	1.95527	1.53493	0.80602	0.58074	0.48479	4.04x
2	2000	Encryption Time	0.06294	0.05387	0.03047	0.01805	0.01985	3.18x
		Decryption Time	3.73262	2.92550	1.43904	0.98343	0.85580	4.37x
		Overall Time	3.85117	3.04296	1.53314	1.06385	0.92902	4.15x
3	3000	Encryption Time	0.09095	0.08032	0.04187	0.02821	0.02807	3.25x
		Decryption Time	5.63559	4.34253	2.16426	1.49051	1.27379	4.43x
		Overall Time	5.18047	4.48739	2.26898	1.58123	1.35528	3.83x
4	4000	Encryption Time	0.12020	0.10032	0.05527	0.04260	0.03130	3.85x
		Decryption Time	7.50242	6.22360	2.89878	1.99339	1.67439	4.49x
		Overall Time	7.67651	6.38835	3.01791	2.09883	1.75975	4.37x
5	5000	Encryption Time	0.15060	0.15593	0.06443	0.04974	0.0405	3.72x
		Decryption Time	9.30783	7.22676	3.58106	2.48179	2.10316	4.43x
		Overall Time	9.51191	7.44727	3.71004	2.59500	2.20005	4.33x
6	6000	Encryption Time	0.19511	0.14662	0.07666	0.05487	0.05249	3.72x
		Decryption Time	11.24749	9.07411	4028718	2.98370	2.51541	4.48x
		Overall Time	11.49780	9.28917	4042824	3.10180	2.62606	4.38x