Enhancing Parallel Implementation of RSA Algorithm using OpenMP

Rim Ali Salah Al-Ardhi (1,*)
Dr. Mohammed Fadhl Abdullah (2)

Received: 20/3/2024
Revised: 14/4/2024
Accepted: 17/11/2024

[1] Faculty of Engineering, Aden University, Aden, Yemen.
[2] Faculty of Eng.& Computing, UST University, Aden, Yemen. Email: m.albadwi@ust.edu
* Corresponding Author Designation, Email: reemalardi@ust.edu

Enhancing Parallel Implementation of RSA Algorithm using OpenMP

Rim Ali Salah Al-Ardhi
Faculty of Engineering, Aden University
Aden, Yemen
reemalardi@ust.edu

Mohammed Fadhl Abdullah
Faculty of Eng.& Computing, UST University
Aden, Yemen
m.albadwi@ust.edu

Abstract—**The Rivest-Shamir-Adleman (RSA) method is commonly used to ensure secure data transmission. However, RSA's encryption and decoding methods need significant processing resources, which can strain computing capacity. To address this issue, we offer a parallel implementation of the RSA algorithm based on OpenMP, a popular shared-memory parallelism API. A series of trials show that our parallel implementation outperforms its sequential equivalent in terms of execution times. Our investigation focuses primarily on the achieved speedup on a multi-core CPU, with an examination of how thread count affects the performance of our parallel solution. This study demonstrates the power of parallel computing in optimizing the RSA method for speedier data transmission.**

Keyword___ RSA algorithm, OpenMP, parallel implementation, multi-core processor, Speedup.

I. Introduction

Encrypting and decrypting data at high speeds is crucial in various applications that utilize cryptographic techniques. The goal of cryptographic algorithms is to securely transmit data through open communication channels while preserving confidentiality, integrity, and availability (the CIA triad) of information resources [1].

Symmetric encryption and asymmetric encryption are two fundamental cryptographic techniques used to secure data and communications. They differ in terms of the keys used and the operations performed during encryption and decryption. The RSA algorithm is a classic example of asymmetric encryption, ensuring private communication even over insecure channels [7].

To address the need for faster RSA implementations, our study explores parallelization of the RSA algorithm in a high-performance computing environment. The paper is organized as follows: Section 2 offers a comprehensive review of related work. Section 3 provides an overview of the RSA algorithm. Section 4 explains The methodology used and our experimental methods for sequential and parallel implementations. Discussion of the finding and result is given in section 5. The final conclusion is given in section 5 followed by the references.

II. RELATED WORK

A parallel implementation of the RSA algorithm was developed to target the exponentiation operations. By partitioning the exponentiation operations into individual processing elements, the encryption and decryption mathematical operations were computed faster, resulting in improved program execution time compared to the serial implementation [1]. However, This study does not address parallelizing or optimizing the modular operations. Additionally, the underlying protocol of the RSA algorithm for key sharing is not considered.

Singh [2] presented an OpenMP-based implementation of the RSA algorithm. The study discussed the traditional RSA cryptosystem and identified that the large size of the text and the modular power function act as bottlenecks for performance. A multi-core architecture was used for the parallel implementation of RSA. The paper compared the execution of RSA in CPU-only mode and in OpenMP mode with different byte sizes (10-byte, 100-byte, 200-byte, 300-byte, 400-byte, and 500-byte). The results showed that higher degrees of parallelism resulted in better performance. The comparison revealed an overhead between sequential and parallel execution.

Patil [3] focused on the development of parallel algorithms to speed up the processes of encryption, decryption, and digital signing in the RSA algorithm. The goal was to reduce both the time and energy involved in these processes.

III. RSA ALGORITHM

The RSA algorithm, developed in 1977 by Rivest, Shamir, and Adleman, is widely used for data authentication and encryption in secure transmission over open networks. It is employed in various online applications, including e-commerce, credit card processing, key exchanges, and digital signatures [6]. RSA relies on modular exponentiations and is slower compared to symmetric-key algorithms. To speed up RSA cryptology process especially for large files size, various solutions have been proposed, including parallel computing.

The RSA algorithm involves the following steps:

- Select two large prime numbers, p and q.
- Calculate the modulus, n, by multiplying p and q: n = p * q.
- Calculate Euler's totient function, phi (φ), as φ = (p - 1) * (q - 1).
- Choose an encryption exponent, e, such that 1 < e < φ and gcd(e, φ) = 1 (e and φ are coprime).
- Calculate the decryption exponent, d, which is the modular multiplicative inverse of e modulo φ, such that (d * e) mod φ = 1.
- The public key consists of the modulus, n, and the encryption exponent, e.
- The private key consists of the modulus, n, and the decryption exponent, d.

Encryption: Convert the plaintext message into a numeric representation. Apply modular exponentiation to the numeric representation using the recipient's public key (n, e) to obtain the ciphertext. E (M) = $M^e \bmod n$, where M is the plain text

Decryption: Apply modular exponentiation to the ciphertext using the recipient's private key (n, d) to obtain the original plaintext. ***Cd mod n*** , where C is the cipher text

IV. METHODOLOGY

A. RSA parallelization

The chosen tool for parallelization is OpenMP, which is a portable API for creating shared memory parallel programs. OpenMP provides a user-friendly and efficient approach to utilize multiple cores in a multicore computer. It offers compiler directives and library routines that allow programmers to write code capable of using all available cores.

Here are the steps for parallel RSA encryption and decryption: (i) Creating a Parallel Region using the #pragma directive provided by OpenMP, (ii) Loop Parallelization: The #pragma omp for directive is used to specify the loop iterations that should be executed in parallel, (iii) Divide the message and the ciphertext into multiple chunks or segments: Depending on the number of available threads for parallel processing, we work on evaluating the message in parallel, (iv) Schedule clause is used to evenly distribute the loop iterations among the threads, and (v) Merge the encrypted/ decrypted chunks from all threads to obtain the final encrypted/ decrypted message.

B. Performance Evaluation: Evaluation of RSA algorithm performance, in its serial (SRSA) and parallel (PRSA) forms, is implemented using OpenMP API

C. Speedup Analysis: The speedup analysis focuses on comparing the execution time of the serial RSA (SRSA) and parallel RSA (PRSA) implementations. This analysis provides insights into the efficiency gained through parallelization.

$$\text{Speed Up:} \quad \boldsymbol{S} = \frac{T_S}{T_P}$$

where **Ts** is the execution time of the sequence RSA implementation and **Tp** is the execution time of the parallel RSA implementation.

D. Efficiency Analysis: This subsection analyzes the efficiency of the parallelized RSA algorithm. It provides insights into the effectiveness of parallelization and resource utilization**.**

$$\textbf{Efficiency:} \quad \mathrm{E} = \frac{S}{p}$$

where **S** is the speedup obtained and **P** is the number of threads utilized**.**

V. EXPERIMENT RESULT AND PERFORMANCE EVALUATION

A. Encryption and Decryption Time Evaluation:

In this section, we present experimental data acquired from encrypted messages saved in an external file as string-based plaintext input. And specific computations are performed using our recognized approach. We effectively parallelize the implementation of the RSA algorithm across the encryption and decryption processes by using the OpenMP library. To assess performance, we will run tests that include both successful encryption and decryption of the input. These executions will occur on a single physical CPU machine or node with varying thread counts (2, 4, 6, 8).

Table 1 shows the encryption time for parallel RSA and demonstrate the improvement in execution time as the number of threads increases from 2 to 6 but there is draw back when it is 8 threads.

Table 1: Encryption Time for Parallel

No. of Threads	Parallel Encryption Time
2	0.001087
4	0.001003
6	0.000989
8	0.00126

Table 2 shows the decryption time for parallel RSA and demonstrate the improvement in execution time as the number of threads increases from 2 to 6 but there is draw back when it is 8.

Table 2: Decryption Time

No. of Threads	Parallel Decryption Time
2	0.000611
4	0.000599
6	0.000445
8	0.00053

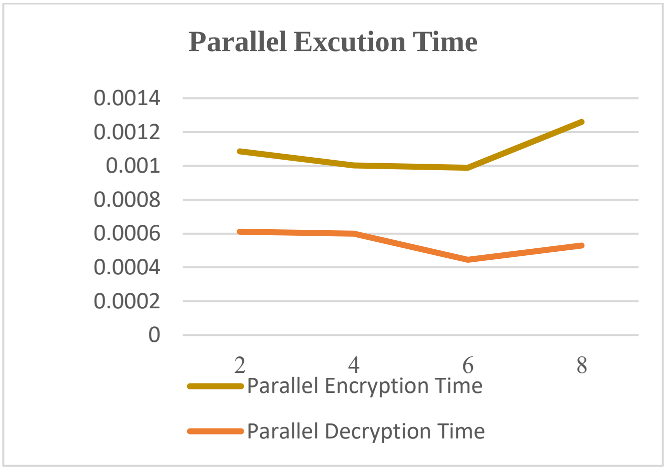

Fig.1 Encryption/Decryption Execution time for parallel in seconds with variant Threads No

B. Speedup and Efficiency Evaluation:

To demonstrate our experimental findings, we measure the speedup and efficiency during variant thread numbers, to present our findings Table 3 is provided, illustrating the runtime (in seconds), speedup, and efficiency for each thread count in encryption process.

Table 3: Driven Speedup and Efficiency in encryption process

No. of Threads	Total Serial Time	Encryption Parallel Time	Speedup	Efficiency
2	0.014469	0.001087	13.31095	6.655474
4	0.014469	0.001003	14.42572	3.606431
6	0.014469	0.000989	14.62993	2.438322
8	0.014469	0.00126	11.48333	1.435417

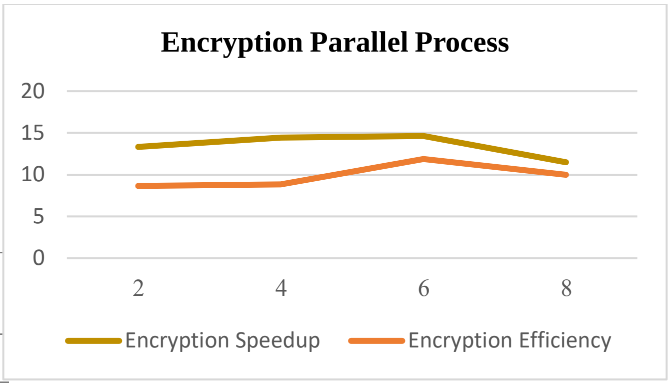

Fig.2 Speedup and Efficiency in seconds in Encryption Process with variant Threads No

Table 4 is provided, illustrating the runtime (in seconds), speedup, and efficiency for each thread count in decryption process.

Table 4: Driven Speedup and Efficiency in Decryption process

No. of Threads	Serial Time	Decryption Parallel Time	Speedup	Efficiency
2	0.005282	0.000611	8.644845	4.322422
4	0.005282	0.000599	8.81803	2.204508
6	0.005282	0.000445	11.86966	1.978277
8	0.005282	0.00053	9.966038	1.245755

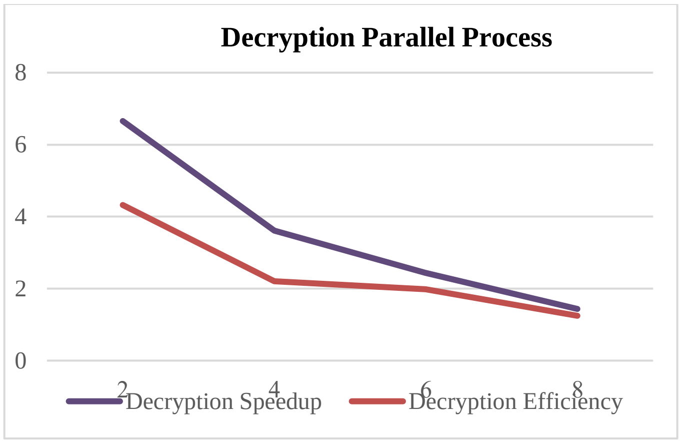

Fig.3 Speedup and Efficiency in Decryption process with variant Threads No

Importantly, the variability in input strings does not affect the speedup and efficiency of our developed tool. The only aspect that varies is the execution time.

According to our testing, the execution time of the encryption process improved as the number of threads increased, but we discovered that at thread 8, performance start to declined due to the balancing load overhead. We too noticed something similar throughout the decryption procedure. In speedup and efficiency calculations, the values we get raise a big question about why the decryptions process does not achieve the same progress we did when we went through threads 2 to 6, and some questions are raised about whether there is a bottleneck we caused in the steps performed and library header folders we used during the implementation of parallel RSA algorithm.

VI. Conclusion

In this study we effectively parallelized the implementation of the RSA algorithm across the encryption and decryption processes using the OpenMP library. To assess performance, we run tests that include both encryption and decryption of the input on a single physical CPU machine with varying thread counts of 2, 4, 6, and 8 threads.

A series of trials show that our parallel implementation outperforms its sequential equivalent in terms of execution times. Our investigation was focused primarily on the achieved speedup on a multi-core CPU, with an examination of how thread count affects the performance of parallel solution.

This study demonstrated the power of parallel computing in optimizing the RSA method for speedier data transmission. We have found that our modified RSA algorithms give us an acceptable result in the decryption and encrypting process considering that we picked a sequence value of the Thread from 2 to 8, but we noted some issues cause this progress fall down in thread number 8 due to the balancing overheads of the parallel computations.

VII. References

[1] Md. A. Ayub, Z. A. Onik, and S. Smith, "Parallelized RSA Algorithm: An Analysis with Performance Evaluation using OpenMP Library in High Performance Computing Environment," in *2019 22nd International Conference of Computer and Information Technology (ICCIT)*, 18-20 Dec. 2019.

[2] N. Singh, "Parallel Implementation of RSA Algorithm using OpenMP and Analysis of Speedup," Conference Paper, Aug. 2022. DOI: 10.5958/2231-3915.2020.00015.2.

[3] N. M. Patil, A. S. Bingeri, A. Sajian, and M. Khazi, "Parallelization of RSA Algorithm Using OpenMP," *JETIR*, vol. 7, no. 7, July 2020.

[4] D. Hu, "Using RSA and AES for file encryption," *CodeProject*, 2014. [Online]. Available: https://www.codeproject.com/Tips/834977/Using-RSA-and-AES-for-File-Encryption. (Accessed: Apr. 27, 2019).

[5] H. M. Fadhil and M. I. Younis, "Parallelizing RSA algorithm on multicore CPU and GPU," *International Journal of Computer Applications*, vol. 87, no. 6, 2014.

[6] A. Mathur, S. Sharma, and S. Khatri, "Parallel Implementation of Cryptographic algorithm for Image Encryption," vol. 4, no. 2, pp. 424-426, 2016.

[7] S. Dhang, R. Das, and P. Chaudhury, "ACAFP: Asymmetric Key based Cryptographic Algorithm using Four Prime Numbers to Secure Message Communication," in *IEEE*, pp. 332-337, 2017.

[8] V. Vidhani and R. Kadam, "Performance Analysis of RSA Algorithm with CUDA Parallel Computing," *IRJET*, vol. 6, no. 5, pp. 6304-6307, May 2019.

[9] S. J. A. O. Djungu, "Parallel approximation of RSA encryption," *IJCSI International Journal of Computer Science Issues*, vol. 17, no. 2, Mar. 2020.

[10] D. I. George Amalarethinam and H. M. Leena, "Enhanced RSA Algorithm with varying Key Sizes for Data Security in Cloud," in *2016 World Congress on Computing and Communication Technologies*.