Exploring the Tiny Encryption Algorithm: A Comparative Analysis of Parallel and Sequential Computation

Mansour Al-Hlalat

Lecturer
Department of Computer Science
University of Jordan, Amman, Jordan

***Abstract*- The Tiny Encryption Algorithm (TEA) is renowned for its strong security and impressive speed, making it highly suitable for lightweight encryption needs in diverse applications. This research paper delves into the investigation of TEA's efficiency by examining the influence of various execution parameters. The study specifically focuses on exploring the impact of factors such as data size, processing type, and the number of processing units in the execution machine on TEA's performance. Through this analysis, valuable insights can be gained to optimize TEA's usage and enhance its overall effectiveness. In addition, this paper introduces a robust model designed to efficiently execute the TEA on parallel machines with large-scale data. The proposed model utilizes a master processor for data splitting and gathering, along with multiple slave processors for executing distributed data. To assess the performance of the TEA algorithm, several experiments were conducted, evaluating factors such as efficiency, execution time, and speedup. These experiments involved varying numbers of plaintexts and key sizes, conducted on both serial and parallel machines, including different cores systems. The TEA algorithm was implemented in C/C++ language using the Message Passing Interface (MPI) library and tested on the high-performance IMAN1 super-computer. The study reveals the significant value of parallel systems in enhancing the overall efficiency of TEA (Tiny Encryption Algorithm), thereby playing a crucial role in the development of secure embedded systems within a short timeframe. The findings demonstrate that parallel processing significantly boosts the computational power of encryption algorithms by distributing computational tasks across multiple processors or cores. Remarkably, the study achieves a substantial decrease in execution time, with a record of 13.258 seconds for a 512k plaintext and 512 key size on a 128-CPU machine. Additionally, the study showcases impressive speed-up across various approaches, highlighting the impactful achievements that fuel further research in this field.**

***Index Terms*- Tiny Encryption Algorithm, Parallel Machines, Encryption, Computation, Fast Software Encryption.**

I. INTRODUCTION

Encryption is a widely used technique for transforming data from its original form, known as plaintext, into an unintelligible form called Ciphertext. This process ensures that only authorized individuals with the appropriate decryption policies and techniques can access the original data [1]. The performance of encryption is influenced by various factors, including the size of the data, the key size, the type of processing (serial or parallel), and the number of execution units employed. The Tiny Encryption Algorithm (TEA) is recognized as a simple yet highly efficient encryption algorithm. It encompasses three versions: TEA, XTEA, and XXTEA. TEA employs two 32-bit blocks and a 128-bit key, while XTEA utilizes a 64-bit block and the same 128-bit key [1, 2]. XXTEA, on the other hand, employs variable-length blocks that are multiples of 32 bits in size. This research focuses on studying the XXTEA algorithm, specifically by implementing both parallel and serial versions using a C++ program with standard and MPI libraries. The aim is to assess the impact of data size, key size, number of processors, and processing type on the algorithm's performance. To achieve this, general execution metrics such as speedup and execution time are calculated. The serial implementation is conducted on a general-purpose computer, while the parallel implementation is performed on the IMAN1 Supercomputer in Jordan. This paper follows a structured organization to present its findings and analysis. In Section II, the background of encryption is discussed, along with an overview of related works focusing on TEA. Section III introduces the proposed model, presenting both the sequential and parallel approaches. The evaluation results of these approaches are presented in Section IV. Finally, the paper concludes with a summary and concluding remarks in Section VI. This organization allows for a comprehensive understanding of the research and its contributions in the field of encryption.

II. BACKGROUND AND RELATED WORK

This section presents a background of the TEA, along with a definition of the IMAN1 Supercomputer. Additionally, an overview of related research is presented, with a specific focus on TEA.

The Tiny Encryption Algorithm (TEA)

The Tiny Encryption Algorithm (TEA) is a symmetric key block cipher that operates on 64-bit blocks of data. It was designed to provide a simple yet effective encryption solution. TEA follows a Feistel network structure and uses a fixed-size 128-bit key, typically represented as two 64-bit values. The algorithm performs a series of iterations, typically 32 rounds, on the input block, applying a set of mathematical operations involving key addition, bit shifting, and XOR operations [1, 2]. TEA is known for its compactness and efficiency, as it requires minimal computational resources and has a small code footprint. Despite its simplicity, TEA has shown to

offer a reasonable level of security for various applications that require lightweight encryption [1, 3]. Table 1 presents the primary steps of the algorithm.

Table 1 The primary steps of the TEA algorithm

key: array represents the key portions used in each round. **+**: represents bitwise addition. **XOR**: operator XOR represents bitwise XOR operation. **shift operators** (<< and >>): represent left and right bit shifts, respectively.
Procedure: **Key Setup:** Input: 128-bit key (represented as two 64-bit values) **Block Encryption:** Input: 64-bit block Divide the block into two 32-bit halves: left and right **Round Iterations:** Input: Number of rounds (usually 32**)** for i = 1 to rounds do: left = left + ((right << 4) + key[0]) XOR (right + key[1]) XOR ((right >> 5) + key[2]) right = right + ((left << 4) + key[3]) XOR (left + key[4]) XOR ((left >> 5) + key[5]) **Block Output:** Output: Encrypted block (final values of left and right halves) **Block Decryption:** Input: Encrypted block (final values of left and right halves) for i = rounds down to 1 do: right = right - ((left << 4) + key[3]) XOR (left + key[4]) XOR ((left >> 5) + key[5]) left = left - ((right << 4) + key[0]) XOR (right + key[1]) XOR ((right >> 5) + key[2])

Eman1 Super-Computer

In 2014, the Eman1 super-computer was launched as the fastest computer in Jordan, with the aim of being the fastest computer in the entire Middle East while keeping costs low. This impressive computer utilized approximately 2260 devices, enabling it to perform an astonishing 25 trillion ultra-fast calculations [4]. As a result, it is recognized as one of the most efficient processing units globally. The primary purpose of this system is to support scientific research across various digital fields. Notable projects include the design and simulation of the core for the Jordanian research nuclear reactor, the development of a model to study synchrotron light beam dynamics, parallel processor projects in operating systems and cyber security, innovation projects in medicines and medical tissue analysis, image analysis projects, and systems in the field of medical images, as well as sustainable energy projects.

Related Works

Encryption algorithms have garnered significant interest among researchers due to their importance in securing sensitive data. This section focuses on exploring and highlighting notable discoveries in the XXTEA Algorithm.

Wheeler and Needham (1995) [1] The proposed approach introduces a concise program that is compatible with various programming languages and can be executed on a wide range of machines. It employs a compact routine based on the Feistel iteration, utilizing a substantial number of rounds to achieve a balance between security and simplicity. The design emphasizes safety through the extensive cycling of the encoding process and the length of the key. Furthermore, the implementation optimizes computational power by employing word-level operations instead of less efficient byte or 4-bit operations. Hunn et al. (2012) [2] the authors introduce a cryptographic algorithm tailored to improve efficiency and minimize storage demands, making it well-suited for resource-limited systems such as embedded systems. The proposed model achieves a harmonious blend of security and simplicity by leveraging mixed algebraic group operations and a substantial number of rounds for both encryption and decryption. The encryption process incorporates 2,883 gates and exhibits a delay time of 16.72ns across sixty-four Feistel rounds, while the decryption process utilizes 2,805 gates with a delay time of 14.78ns. In a study by Yarrkov (2010) [3], XXTEA, also referred to as Corrected Block TEA, is introduced as a member of the TEA series of algorithms. The XXTEA algorithm, although simple in nature, was subjected to a plaintext attack using approximately 259 queries and minimal computational effort. The findings of the study revealed that XXTEA did not achieve the intended level of 128-bit security as originally envisioned.

Many research papers tackle the diverse challenges involved in the implementation of encryption algorithms, with a specific focus on addressing issues such as execution time. Moreover, the exploration of parallel computation utilizing multi-core processors is extensively discussed as a promising approach to improve the overall performance of encryption algorithms. Hunn et al. (2012) [5] The author suggests that TEA fulfills all the criteria for an effective GPU Pseudo Random Number Generator (PRNG). It has shown promising results compared to existing approaches. The evaluation of TEA using standard randomness test suites, Perlin noise, and

a Monte-Carlo shadow algorithm confirms its ability to generate high-quality noise. Additionally, TEA outperforms MD5 in terms of speed while maintaining equivalent levels of randomness. The empirical findings of this study support the use of TEA, specifically with 8 rounds, as a faster alternative to MD5 for generating high-quality random numbers.

Tallapally and Manjula (2020) [6] The authors conducted an evaluation of various block cipher techniques, including Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR), using the Java platform. The study revealed that the Counter mode exhibited the best performance among the evaluated techniques. Elkabbany et al. (2015) [7] proposed model that introduces a high-performance security algorithm using the AES algorithm with parallel and pipelining approaches. The model effectively implements pipelining for all rounds and parallelizes the Add_Round_Key and Mix_Column transformations. Experimental results demonstrate significant performance improvements, with a 95% enhancement in the pipeline approach and a remarkable 98% improvement in the parallelizing approach. This model showcases the potential for efficient hardware and software implementations of the AES algorithm for enhanced security. Celikel et al. (2006) [8] Authors design and execute DES encryption in the ECB mode using parallelization schemes, namely pipeline, block, and plain-text. Among these schemes, parallelization based on plain-text demonstrated the most favorable results. Additionally, the authors observed that the speed of DES was not dependent on the source language used.

Esmaeel (2012) [9] Author presents a comprehensive overview of how a block cipher can be constructed, encompassing an examination of its historical context, the inventors involved, and the algorithms employed, with a particular focus on the TEA block ciphers. Furthermore, it discusses the programming approach taken, emphasizing aspects such as modularity, simplicity, and resource allocation. Rajesh et al. (2019) [10] authors propose a novel tiny symmetric encryption algorithm (NTSA) aimed at enhancing security for the transfer of text files through IoT networks. The algorithm introduces dynamic key confusions for each round of encryption, thereby bolstering the overall security of the system. In order to validate the effectiveness of NTSA in an IoT network, extensive experiments were conducted, which encompassed analyzing the avalanche effect, as well as measuring the encryption and decryption time. The experiments were conducted on various embedded devices commonly found in IoT networks.

III. METHODOLOGY

This section describes the architecture of the proposed model for sequential and parallel approaches.

Sequential Approach (SA)

The proposed Sequential Approach (SA) of the TEA algorithm presented using NetBeans as the development environment, C/C++ Compiler, Windows 10 operating system, Core(TM) i7 Duo-3.4GHz processor, and 8 GB RAM. The process of this approach is implemented in Table 2.

Table 2 The processes of the SA approach for the TEA algorithm

Algorithm: Process_Sequential _implementation ()

1. Read the list of data files. (sizes: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 k-bytes).
2. For each DataFile in the list of data files:
   For each KeySize in [512, 256, 128, 64, 32, 16]:
   Apply the XXTEA function to the data in DataFile using the current KeySize.
   Add the result to DataFileResults.
   Add DataFileResults to Results.
3. Return the Results.

Parallel Approach (PA)

The proposed Parallel Approach (PA) for (TEA) presented using multicore architectures using IMAN1 Supercomputer. The implementation is done in C++ with the MPI library to enable parallel encryption and decryption. Fig. 1 provides an overview of the approach's structure. Further the algorithm for this methodology is presented in Table 3.

Table 3 The processes of the PA approach for the TEA algorithm

Algorithm: Process_Parallel_implementation ()

1. Read the list of data files. (sizes: 1, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB).
2. Determine the number of processors (n).
3. If n is greater than 3:
   Assign one of the processors as the master processor.
   Assign the remaining processors as slaves (workers).
   Split each file among the slave processors based on the formula: file size / (n-1).
   Assign each piece of data to the corresponding slave processor.
4. If n is not greater than 3:
   Assign one of the processors as the master processor and slave.

Assign the remaining processors as slaves (workers).
Split each file among the slave processors based on the formula: file size / n.
Assign each piece of data to the corresponding slave processor.

5. Execute the XXTEA function for each data in its respective slave processor for all key sizes: 512, 64, and 16 bytes.
6. After all slave processors have finished, the master processor gathers all the data.
7. Save and return the collected data.

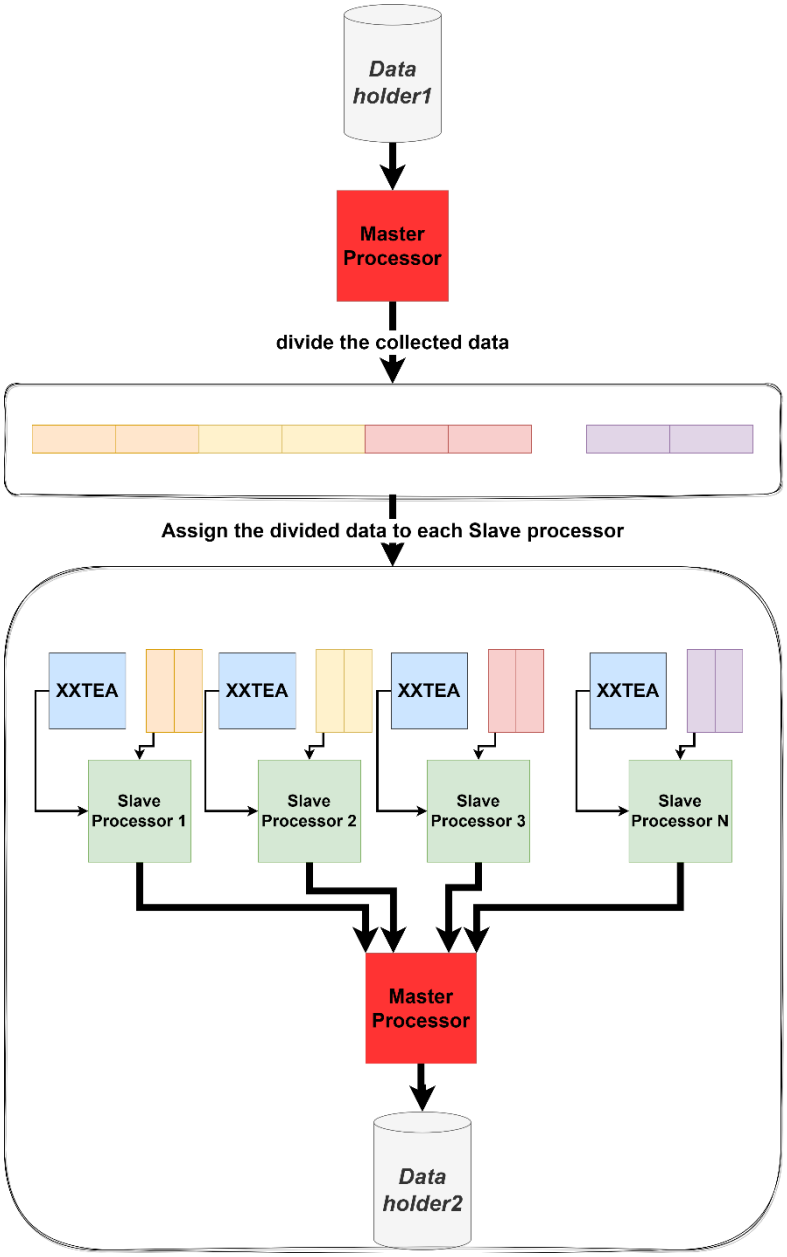

Figure 1 The architecture of the PA approach for the TEA algorithm

In the architecture of the PA approach, the first step is to read the data from the data holder1, which contains the input files. To distribute the workload efficiently, one of the processors is assigned as the master processor, while the remaining processors are designated as slaves or workers. The input files are then split among the slave processors, with each piece of data assigned to the corresponding slave processor for further processing. In each slave processor, the XXTEA function is executed on the assigned data. Once all the slave processors have completed their computations, the master processor gathers all the processed data from the slaves. Finally, the collected data is saved to the data holder2, ensuring that the results are stored for further analysis or use in subsequent stages of the system. This distributed processing approach allows for efficient and parallel execution of the XXTEA function, improving overall performance and scalability.

In summary, the utilization of a distributed processing approach with a master-slave architecture for executing the XXTEA function brings notable advantages to the overall system. It enables efficient parallelization, significantly improving performance by reducing processing time. Additionally, the system's scalability is enhanced, ensuring the ability to handle larger datasets with ease. This approach demonstrates the effectiveness of distributed computing in optimizing the security and efficiency of data transfer in IoT networks.

Evaluation Metrics

The performance evaluation of the proposal methodology encompasses various metrics includes Execution time (T), Speedup (S), Performance (P), Throughput (Th), and Scalability (Sca) [11].

Execution time is the duration of time needed for a task to complete its execution. It determined by the difference between its start time and end time. Equation 1 shows how to calculate the T.

$$T = Te - Ts \quad (1)$$

The speedup of a parallel algorithm over a corresponding sequential algorithm is the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm. Equation 2 shows how to calculate the S.

$$S = Ts/Tp \quad (2)$$

Where, Ts is the compute time for the sequential algorithm, Tp is the compute time for the parallel algorithm.

Performance is a characteristic that quantifies the efficiency of an algorithm in terms of its execution time. It is commonly understood that as performance increases, the execution time decreases. Equation 3 shows how to calculate the P.

$$P = 1/T \quad (3)$$

Throughput refers to the quantity or volume of work (W) completed within a specified time period (T). It represents the overall efficiency or productivity achieved during a particular duration. Equation 4 shows how to calculate the Th.

$$Th = W/T \quad (4)$$

Scalability is a characteristic that measures the performance improvement of an algorithm when executed on varying numbers of cores or processors.

IV. RESULTS AND DISCUSSIONS

In this section, the performance evaluation of each proposed approach is presented. The effectiveness of both approaches is analyzed under different execution strategies, considering various plain texts, key sizes, and different numbers of processors. The implementation of these approaches utilizes the C/C++ programming language, along with the MPI library, and additional libraries such as stdio.h, ctype.h, string.h, and math.h.

SA Results

The execution times in seconds for different plaintext sizes in bytes, along with their respective key sizes, are presented in Table 4. Based on the results obtained, the following observations have been identified:

- When the size of the plaintext increases while keeping the key size fixed, there is a significant increase in the execution time. This observation can be attributed to the larger amount of data being processed sequentially, which requires more execution time. This suggests that employing a multi-processor system becomes essential for the TEA algorithm when dealing with substantial data volumes in encryption and decryption operations.
- When the size of the plaintext is fixed and the key size remains unchanged, the execution time increases. This observation is due to the fact that TEA operations are influenced by the input data size rather than the key size. Therefore, it is possible to encrypt data using a larger key (for increased security) at a minimal increase in execution time.

These observations highlight the trade-offs between plaintext size, key size, and execution time in the TEA algorithm, emphasizing the need for considering system capabilities and security requirements when employing TEA for data encryption.

Table 4 The proposed SA approach results in different plaintext sizes, along with their respective key sizes

		Key size in byte					
		16	**32**	**64**	**128**	**256**	**512**
Plain text size	**1K**	0.635	0.756	0.825	2.562	6.515	8.971
	2K	0.879	0.978	1.758	2.958	8.145	11.101
	4K	1.060	1.780	2.243	4.961	10.145	14.801
	8K	1.320	3.622	3.991	5.948	14.615	23.314
	16K	1.919	5.744	6.092	10.454	19.162	23.297
	32K	2.187	6.319	9.698	15.501	30.145	33.383
	64K	3.772	7.509	13.953	19.049	39.144	43.306
	128K	5.241	8.062	14.420	21.338	45.155	53.488

	256K	9.349	13.312	19.266	26.111	50.152	59.333
	512K	10.145	17.961	24.450	30.415	59.156	63.056
	1M	15.622	24.589	31.079	39.761	60.545	61.320
	2M	19.780	28.225	38.070	49.747	65.125	73.404
	4M	22.241	36.989	47.433	51.338	72.956	74.261

PA Results

In first experiment for the PA, different plaintext sizes in bytes with 16-byte key size, along with multiple processors are applied in the PA, the of this experiment presented in Table 5. Based on the obtained results, the following observations have been identified:

- When executing the same plaintext with different numbers of processors, a significant decrease in execution time is observed until a certain threshold for the number of processors. Beyond this threshold, the execution time suddenly increases. This observation is attributed to the lack of significant parallelism in TEA for a large number of processors. Additionally, the overhead of parallelization outweighs the benefits gained in this scenario.
- When executing a large plaintext, parallelism performs better compared to executing a small plaintext. This is because the dependencies among tasks in small data make parallel execution challenging and less efficient.
-

Table 5 The proposed PA approach results in the first experiment

		Plain text size byte												
		1K	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**	**256K**	**512K**	**1M**	**2M**	**4M**
Number of Processors	1	0.215	0.356	0.420	0.465	0.624	1.459	1.525	2.524	3.466	3.851	4.715	4.812	5.512
	2	0.146	0.328	0.410	0.453	0.612	1.325	1.452	2.152	2.515	2.712	3.612	3.726	4.901
	4	0.128	0.278	0.300	0.426	0.652	1.192	1.354	1.025	2.125	2.350	3.735	4.736	4.365
	8	0.252	0.295	0.420	0.685	0.859	1.158	1.255	0.951	2.525	2.922	3.912	4.998	5.125
	16	0.356	0.385	0.650	0.823	0.956	1.332	1.452	1.458	3.151	4.160	5.165	7.370	8.376
	32	0.475	0.482	0.626	1.025	1.158	1.445	1.528	1.929	5.924	6.288	7.433	10.844	11.257
	64	0.559	0.782	0.845	1.253	1.454	1.855	1.935	3.255	7.357	8.370	10.024	13.527	15.125
	128	0.635	0.879	1.060	1.320	1.919	2.187	3.772	5.241	9.349	10.145	15.622	19.780	22.241

The speedup recodes for the first experiment is calculated and presnted in Fig. 2 and Fig. 3. the following observations have been identified:

- As the number of processors increases, there is a significant improvement in execution time, resulting in a higher speedup. This is attributed to the parallel nature of the algorithm, where tasks can be divided and processed simultaneously by multiple processors.
- However, there is a point of diminishing returns, where increasing the number of processors beyond a certain threshold does not lead to further improvements in speedup. This can be attributed to factors such as communication overhead and resource contention, which can limit the scalability of the parallel execution.
- It is important to note that the achieved speedup is highly dependent on the problem size and the degree of parallelism inherent in the algorithm. Larger problem sizes tend to exhibit better speedup due to a higher potential for parallelization.

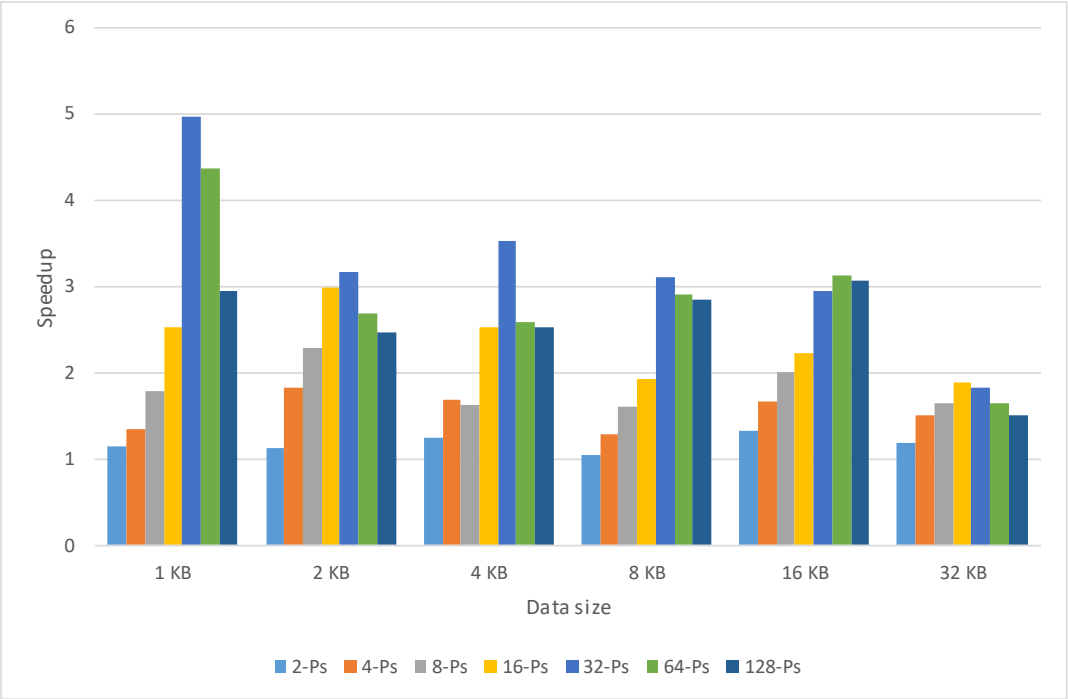

Figure 2 The speedup records for the first experiment, considering plaintext sizes ranging from 1 KB up to 32 KB

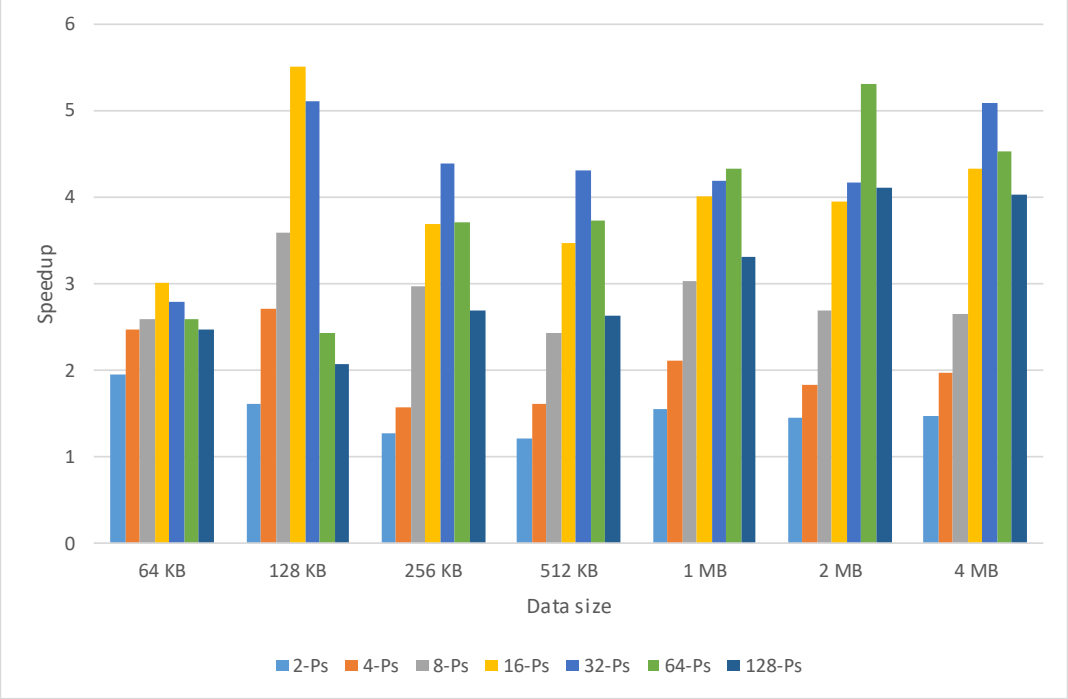

Figure 3 The speedup records for the first experiment, considering plaintext sizes ranging from 64 KB up to 4 MB

In second experiment for the PA, different plaintext sizes in bytes with different key size, along with multiple processors are applied in the PA, the of this experiment presented in Table 6, Table 7, and Table 8. Based on the obtained results, the following observations have been identified:

- When using larger key sizes, there is a decrease in execution time. This due to the increasing of security measures and complexity in the encryption process.
- Smaller key sizes can result in faster execution times but may sacrifice some level of security.

Table 6 The proposed PA approach results in the second experiment with 512-byte key size

	Plain text size in byte		
	1K	**64K**	**512K**
Number of Proce 1	8.971	43.306	63.056

	2	6.515	29.257	39.560
	4	4.156	21.456	25.645
	8	2.149	12.154	14.126
	16	1.892	7.125	12.125
	32	1.515	5.685	7.155
	64	2.125	9.658	11.698
	128	3.584	11.580	13.258

Table 7 The proposed PA approach results in the second experiment with 64-byte key size

		Plain text size in byte		
		1K	**64K**	**512K**
Number of Processors	**1**	2.562	19.049	30.415
	2	1.955	11.256	19.566
	4	1.215	7.256	11.257
	8	0.763	4.126	8.156
	16	0.415	3.561	5.257
	32	0.512	5.698	8.966
	64	0.563	6.986	9.955
	128	0.653	7.986	10.398

Table 8 The proposed PA approach results in the second experiment with 16-byte key size

		Plain text size in byte		
		1k	**64k**	**512k**
Number of Processors	**1**	0.635	3.772	10.145
	2	0.559	1.935	8.370
	4	0.475	1.528	6.288
	8	0.356	1.452	4.160
	16	0.252	1.255	2.922
	32	0.128	1.354	2.350
	64	0.146	1.452	2.712
	128	0.215	1.525	3.851

The speedup recodes for the second experiment is calculated and presented in Fig. 4, Fig. 5, and Fig. 6. the following observations have been identified

- Increasing the number of processors leads to a notable enhancement in execution time and a higher speedup. This is due to the parallel nature of the algorithm, allowing tasks to be divided and processed concurrently by multiple processors. However, there is a point of diminishing returns where adding more processors does not yield further speedup improvements. This can be attributed to factors like communication overhead and resource contention, limiting the scalability of parallel execution.
- Utilizing smaller key sizes can result in faster execution times, but it may come at the expense of compromising some level of security.
- When dealing with larger plaintext, parallel execution demonstrates superior performance compared to smaller plaintext. This is primarily due to the complexities of task dependencies in small data, making parallel execution less efficient and challenging.

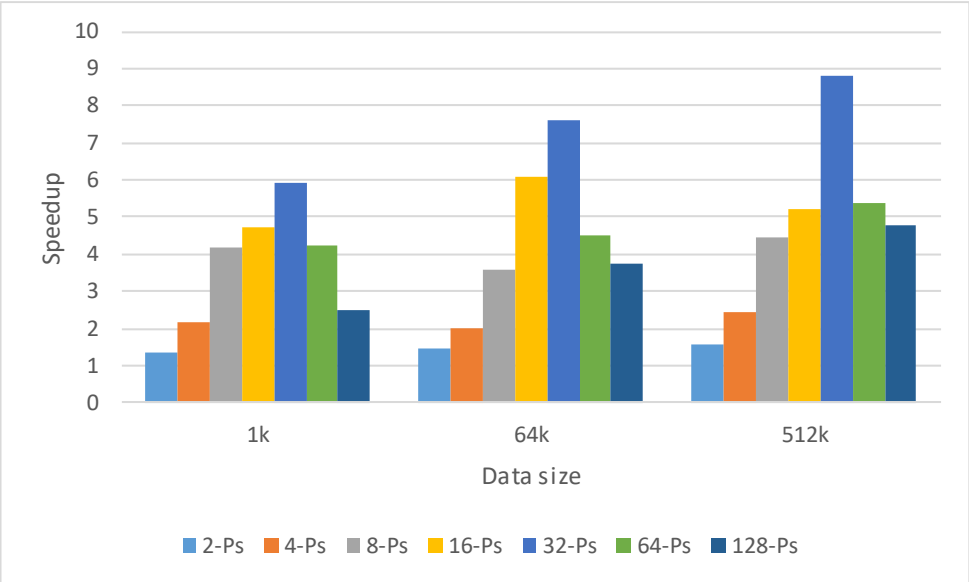

Figure 4 The speedup records for the second experiment with 512-byte key size

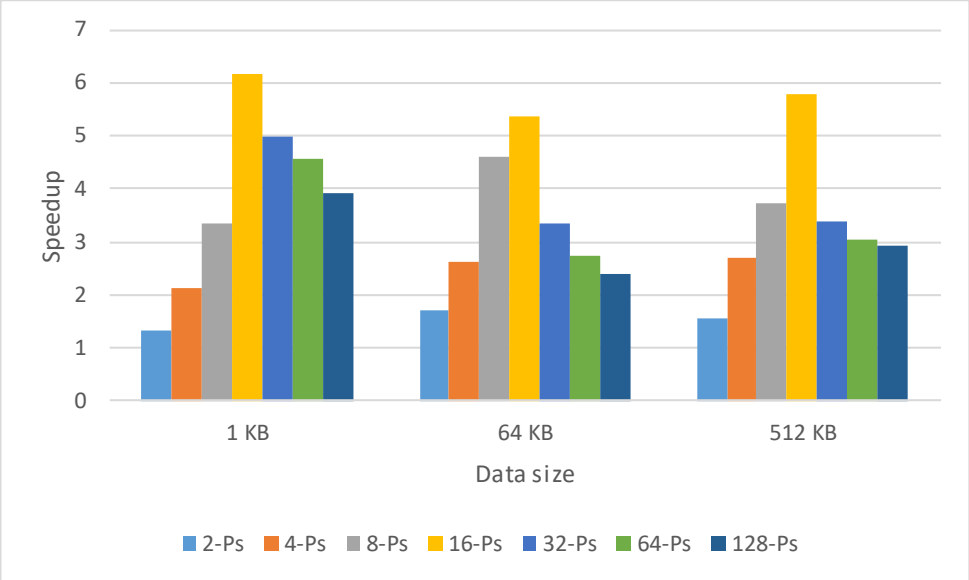

Figure 5 The speedup records for the second experiment with 64-byte key size

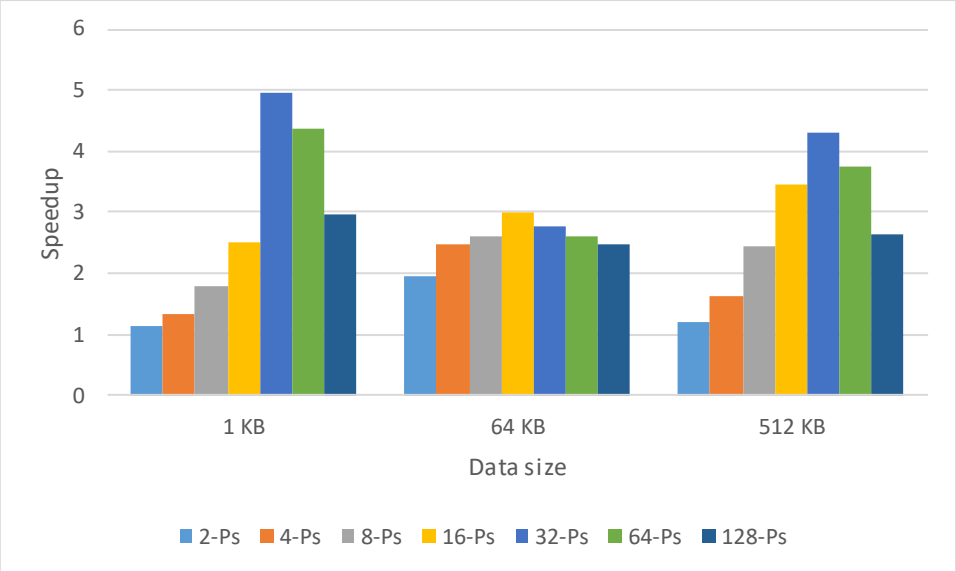

Figure 6 The speedup records for the second experiment with 16-byte key size

It is noteworthy that finding the right balance between security and computational efficiency is crucial for any application. When selecting a key size for parallel execution, it is important to align it with the security requirements and available computing resources. This decision should be based on a comprehensive analysis of the specific application's needs, taking into account factors such as data sensitivity and the desired trade-off between security and performance. By carefully considering these aspects, the optimal key size can be determined to ensure an effective and efficient encryption process in parallel execution. In summary, increasing the number of processors can significantly improve execution time and speedup in parallel execution. Smaller key sizes can enhance execution speed but may have security implications, and parallelism is more effective for larger plaintext due to reduced task dependencies. Careful consideration should be given to optimize the number of processors, key sizes, and plaintext size for achieving the desired trade-off between performance, security, and parallel efficiency.

V. CONCLUSIONS

In conclusion, this research paper presented a comprehensive analysis of parallel and sequential programming approaches for the Tiny Encryption Algorithm (TEA) and conducted a thorough performance evaluation considering various execution parameters. Factors such as data size, key size, processing type, and the number of processors were examined. The findings revealed that, contrary to expectations, the parallel implementation of TEA on the IMAN1 Supercomputer resulted in longer encryption and decryption times compared to the sequential implementation. However, it is important to note that the study achieved a remarkable reduction in execution time, with a notable record of 13.258 seconds observed for a 512k plaintext and 512 key size on a 128-CPU machine.

Furthermore, the research demonstrated significant speed-up across different approaches, indicating notable advancements in this field. These achievements provide a solid foundation for future research endeavors in parallel and sequential programming of encryption algorithms. It is important to consider that parallel execution offers advantages in terms of speedup and efficiency. However, the selection of the number of processors, key size, and data size should be carefully balanced to achieve the desired trade-off between performance and security. Future studies can build upon these findings to further enhance the parallel and sequential programming of encryption algorithms.

VI. ACKNOWLEDGEMENT

I would like to express my gratitude to everyone who contributed to the completion of this study. I extend special thanks to those who reviewed the study and the staff at the IMAN1 Supercomputer. Your collaboration and efforts have greatly contributed to presenting this study with the highest level of quality and accuracy.

REFERENCES:

[1] D. Wheeler, R. Needham, TEA, “A tiny encryption algorithm. In: Preneel B. (eds) Fast Software Encryption”, Lecture Notes in Computer Science, 1995, vol. 1, pp. 1008.
[2] S. A. Yee Hunn, S. Z. binti Md. Naziri and N. binti Idris, “The development of tiny encryption algorithm (TEA) crypto-core for mobile systems”, IEEE International Conference on Electronics Design, Systems and Applications (ICEDSA), Kuala Lumpur, 2021, vol.1, pp. 45-49.
[3] E. Yarrkov, “Cryptanalysis of XXTEA. Yarrkov 2010 Cryptanalysis OX”, 2010, 254-260.
[4] Royal Hashemite Court,” IMAN1 Supercomputer”. Retrieved June 13, 2023, from https://rhc.jo/en/gallery/infograph/iman1-supercomputer
[5] S. A. Yee Hunn, S. Z. binti Md. Naziri and N. binti Idris. “The development of tiny encryption algorithm (TEA) crypto-core for mobile systems”, IEEE International Conference on Electronics Design, Systems and Applications (ICEDSA), Kuala Lumpur, Malaysia. vol. 1 pp. 45-49.
[6] T. Sampath, B. Manjula, “Suitable encrypting algorithms in Parallel Processing for improved efficiency”, IOP Conference Series: Materials Science and Engineering, 2020, vol.1, pp. 981.
[7] E., Ghada, A., Heba, R., Mohamed, “A Design of a Fast Parallel-Pipelined Implementation of AES: Advanced Encryption Standard”, International Journal of Computer Science and Information Technology, 2014, vol. 6, pp. 39-59.
[8] E. Celikel, J. Davidson, and C. Kern, “Parallel performance of des in ecb mode”. In Computer Networks, 2006 International Symposium on”, 2006, vol. 1, pp. 134–139.
[9] E.Hana, “Apply Block Ciphers Using Tiny Encryption Algorithm (TEA)”. Baghdad Science Journal, 7, 1061-1069.
[10] R. Sreeja, P. Varghese, M. Varun, and K, Mohammad. A Secure and Efficient Lightweight Symmetric Encryption Scheme for Transfer of Text Files between Embedded IoT Devices. Symmetry, 2019, vol. 11, pp. 293-314.
[11] E. Donald, A. Vincent, “Performance Evaluation of Parallel Algorithms”, International Journal of Computer Science and Engineering, 2022, vol. 9, pp. 10-14.