ParalCrypto: Enhancing Cryptographic Algorithms through Parallel Computing

Xihang Yu*

College of Literature, Science, and the Arts, University of Michigan.

Abstract

This project delves into the potential for parallel processing within cryptographic algorithms. It underscores the capability of symmetric key algorithms like the Advanced Encryption Standard (AES), especially in Counter Mode (CTR) and Electronic Code Book (ECB) modes, to encrypt or decrypt multiple data blocks in parallel, leveraging their independent block processing feature. Although the encryption process in Cipher Block Chaining (CBC) mode is inherently sequential, the decryption process benefits from parallelization, offering a significant performance boost. The study also covers hash functions, specifically how the Merkle tree structure can facilitate parallel hashing across its various branches, even though hash functions based on the Merkle–Damgård construction, like SHA-256, are sequential in their basic form. Overall, the project provides an application-focused analysis of core cryptographic algorithms, offering a detailed technical examination and practical examples from each category to demonstrate the application of parallel processing in enhancing cryptographic operations.[1][2][3]

1 Problem Formulation

1.1 Symmetric Key Algorithms

AES (Advanced Encryption Standard): In Advanced Encryption Standard (AES) operations, particularly within Counter Mode (CTR) and Electronic CodeBook (ECB) modes, each block of data is an independent candidate for encryption or decryption, facilitating parallel processing. While Cipher Block Chaining (CBC) mode follows this paradigm for decryption, allowing blocks to be decrypted concurrently, its encryption process does not lend itself to parallelization. Nevertheless, the intrinsic ability of these modes to process multiple blocks simultaneously can markedly enhance performance, capitalizing on the strengths of parallel computing.

1.2 Hash Functions

Merkle Tree: The nature of the compression function in hash algorithms inherently precludes parallel processing. Take SHA-256, a hash function based on the Merkle–Damgård construction, as an example. Its sequential design requires each message block to be processed in order, with the hashing of one block reliant on the completion of the previous one, thus negating the possibility of parallel execution. Despite this limitation within individual hash functions, the structure of a Merkle tree circumvents this by enabling parallel hashing at each level of its depth. As a result, while the hash functions themselves, such as those devised using the Merkle–Damgård construction, cannot be parallelized, the Merkle tree efficiently leverages these functions across its multiple branches to optimize hash generation.

*Course project in EECS475 Introduction to Cryptography at University of Michigan. Email: xihangyu@umich.edu

[1] For the experiments, I want to compare the speeds the paralleled algorithm with unparalleled one. For fair comparison, all code will be written in Python.

[2] Code is published in this link: https://github.com/XihangYU630/ParalCrypto.

[3] Video is in this link: https://drive.google.com/file/d/1Aq5ewWQzG0O9ZdRiB3s8UDNwoqMgyGuU/view?usp=drive_link.

(a) Original (a) CTR-parallel (a) CTR-sequential

Figure 1: Illustration of CTR-parallel and CTR-sequential decryption, against the original image.

Table 1: Performance comparison in running time (s) ↓ between parallelized CTR and sequential CTR

	CTR-parallel (Ours)		CTR-sequential	
	Encryption	Decryption	Encryption	Decryption
image	**2.1453**	**2.1483**	2.6472	2.6288
text	**0.0021**	**0.0016**	0.0024	0.0018

2 Experiments

We have carried out experimental investigations on Symmetric Key Algorithms and Hash Functions. In the following section, we will show the experimental setup and performance analysis.

2.1 Symmetric Key Algorithms

2.1.1 Counter Mode(CTR)

Setup: We developed our own versions of CTR algorithms from the ground up. While there are existing Python libraries offering fast CTR algorithms, these might incorporate complex parallelism techniques and be highly optimized. Our aim is to explore various levels of parallelism and have a full understanding of how parallelism works, so using these pre-optimized solutions would not be ideal. Our implementation of CTR algorithms is entirely original, with the exception of the Pseudorandom Function (PRF). Implementing PRF is challenging, so we opted to utilize the function from an existing library. However, this does not interfere with our experiments. In both sequential and parallel forms of the CTR algorithms, the execution time for a single run of PRF remains consistent, thus not impacting our analysis. The pycryptodome library in Python was used for this purpose. We segmented the data into 16-byte chunks, along with a 16-byte IV. Due to Python numpy's lack of a 128-bit data type, we stored the Initialization Vector (IV) in two separate 8-byte blocks. We conducted experiments on both image data and text data. For our image data, we downloaded images of size 512×512. For text data, we utilized material sourced from the internet: *Low Water Levels and Rising Temperatures: Recently, the Negro River in the Amazon rainforest near Manaus, Brazil, reached its lowest level in 120 years, dropping to just 12.70 meters. In Lake Tefe, located approximately 500 kilometers west, over 150 river dolphins were found dead, likely due to temperatures nearing 40 degrees.* We run 10 Monte-Carlo experiments and calculate the average running time for each algorithm and data combination.

Table 2: Performance comparison between different paralleled CTR implementations

	CTR-parallel (Ours)		CTR-parallel-PRF	
	Encryption	Decryption	Encryption	Decryption
image	2.1453	2.1483	10.9705	10.6112
text	0.0021	0.0016	2.1052	2.1108

Table 3: Performance comparison in running time (s) ↓ between parallelized ECB and sequential ECB

	ECB-parallel (Ours)		ECB-sequential	
	Encryption	Decryption	Encryption	Decryption
image	**0.1288**	**0.1274**	0.9390	0.9473
text	0.000896	0.000606	**0.000881**	**0.000596**

Results: The decryption outcomes are depicted in Figure 1. We can see both parallel CTR and sequential CTR correctly decrypts the original message. As Table 1 illustrates, the parallelized version of the CTR mode demonstrates superior performance over its sequential counterpart in terms of both encryption and decryption speeds. Notably, the time taken for encryption and decryption in CTR is roughly equivalent. The enhanced efficiency of parallel CTR compared to sequential CTR can be attributed to two main factors: (1) In sequential CTR, counter addresses are generated one by one in a for loop and incremented with each iteration, whereas parallel CTR generates all counter addresses simultaneously. (2) Parallel CTR performs XOR operations in a batch, in contrast to the sequential CTR, which processes data incrementally. It's important to note that the Pseudo-Random Function (PRF) $F_k(x)$ operates sequentially in both parallel and sequential CTR modes. Further discussion on CTR with parallel PRF will be included in the ablation study. **Ablation and Analysis:** In our study, we also conducted an ablation analysis on the implementation of a parallelized Pseudorandom Function (PRF) for CTR, referred to as CTR-parallel-PRF. We use ProcessPoolExecutor library in Python for parallel computing. The results, as detailed in Table 2, reveal that CTR-parallel-PRF does not outperform the standard parallel CTR. We attribute this observation to two primary factors: (1) Process Management Overhead: Employing parallel processing through ProcessPoolExecutor incurs additional overhead, including the creation of processes, context switching, and communication between processes. This overhead can be counterproductive, especially for smaller datasets or tasks that are not extremely demanding in terms of computation, thus negating the advantages of parallel processing. (2) Limited Cores/Threads: The effectiveness of parallelism is also contingent on the number of cores or threads a machine has. With a limited number of cores, the improvements in performance may not be substantial. Consequently, we conclude that selecting an appropriate level of parallelism is crucial for achieving efficient cryptographic implementations.

2.1.2 Electronic Codebook Mode (ECB)

Setup: For similar reasons previously mentioned, we crafted our own Electronic Codebook (ECB) algorithms from scratch. Our approach to ECB algorithm development is completely original, except for the Pseudorandom Permutation (PRP) function. Given the complexity of implementing PRP, we integrated an existing library's function. This integration does not adversely affect our experiments, as the execution time for a single PRP operation remains stable in both the sequential and parallel versions of our ECB algorithms, therefore not influencing our results. The pycryptodome library in Python facilitated this process. Data was divided into 16-byte blocks, along with a 16-byte Initialization Vector (IV). Owing to the absence of a 128-bit data type in Python's numpy, we stored the IV in two 8-byte segments. Our experiments were

(a) Original (a) ECB-parallel (a) ECB-sequential

Figure 2: Illustration of ECB-parallel and ECB-sequential decryption, against the original image.

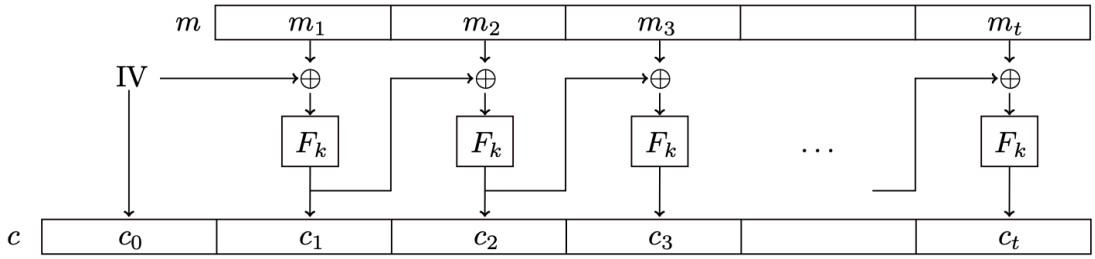

Figure 3: Illustration of CBC encryption.

conducted on both image and text data, similar to those in our CTR studies. We performed 10 Monte-Carlo simulations for each algorithm and data type, calculating the average running time to assess performance.

Results: The results of the decryption process are shown in Figure 2. It is evident that both parallel and sequential versions of ECB (Electronic Codebook) mode successfully decrypt the original message. As demonstrated in Table 3, for image data, the parallel ECB mode outperforms the sequential one. However, with text data, the sequential ECB mode has a slight edge. This difference in performance can be attributed to two key reasons: (1) The smaller size of the text data means that the computational burden of the sequential implementation, which relies on a for loop, is not excessive. (2) The additional overhead in the parallel ECB due to the creation of padded data.

2.1.3 Cipher Block Chaining (CBC)

We now shift our attention to the Cipher Block Chaining (CBC) algorithm. Observing Figures 3 and 4, we note that in the encryption process, the encryption of each block is contingent on the preceding ciphertext, precluding parallelization. Conversely, in decryption, although each block's decryption is also reliant on the previous ciphertext, all ciphertexts are already stored in memory. This allows for the parallel computation of each block's decryption. This characteristic informs the following implementation strategy.

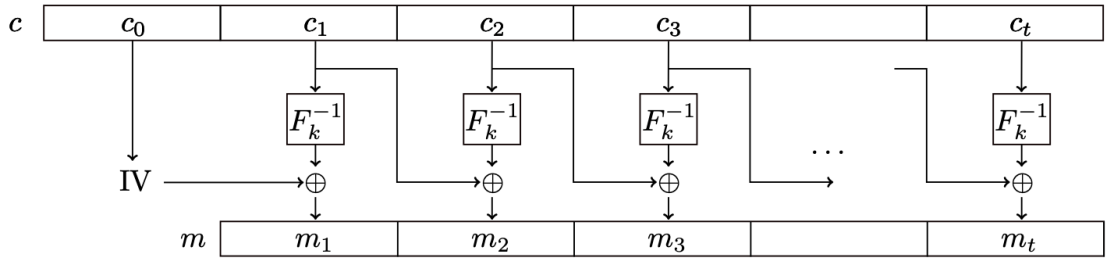

Figure 4: Illustration of CBC decryption.

(a) Original (a) CBC-parallel (a) CBC-sequential

Figure 5: Illustration of CBC-parallel and CBC-sequential decryption, against the original image.

Setup: As previously indicated, we developed our own Cipher Block Chaining (CBC) algorithms from the ground up. Our method of constructing the CBC algorithm is entirely unique, with the exception of the Pseudorandom Permutation (PRP) function. Owing to the complexities involved in implementing PRP, we incorporated a function from an existing library, specifically using *AES.new(key, AES.MODE_ECB)* in Python. This incorporation does not negatively impact our experimental results, as the time taken for a single PRP operation is consistent across both sequential and parallel versions of our Electronic Codebook (ECB) algorithms, thereby not skewing the outcomes. We segmented data into 16-byte blocks, accompanied by a 16-byte Initialization Vector (IV). The PRP key size is 32 bytes. Our research encompassed both image and text data types, paralleling those used in our Counter Mode (CTR) analyses. To evaluate performance, we conducted 50 Monte-Carlo simulations for each algorithm and data type, computing the average execution time.

Results: The outcomes of the decryption procedure are illustrated in Figure 5. It is clear from these results that both the parallel and sequential forms of the CBC algorithm effectively restore the original message. As highlighted in Table 4, the parallel version of CBC demonstrates a superior performance in decrypting both image and text data, reducing time cost of the sequential version by a margin of 70%-80%. This significant improvement underscores the efficiency of the parallelized CBC algorithm. The encryption time of CBC-parallel and CBC-sequential are approximatedly the same.

Table 4: Performance comparison in running time (s) ↓ between parallelized CBC and sequential CBC

	CBC-parallel (Ours)		CBC-sequential	
	Encryption	Decryption	Encryption	Decryption
image	2.5493	**0.3498**	2.5355	1.9588
text	0.0026	**0.0003**	0.0017	0.0013

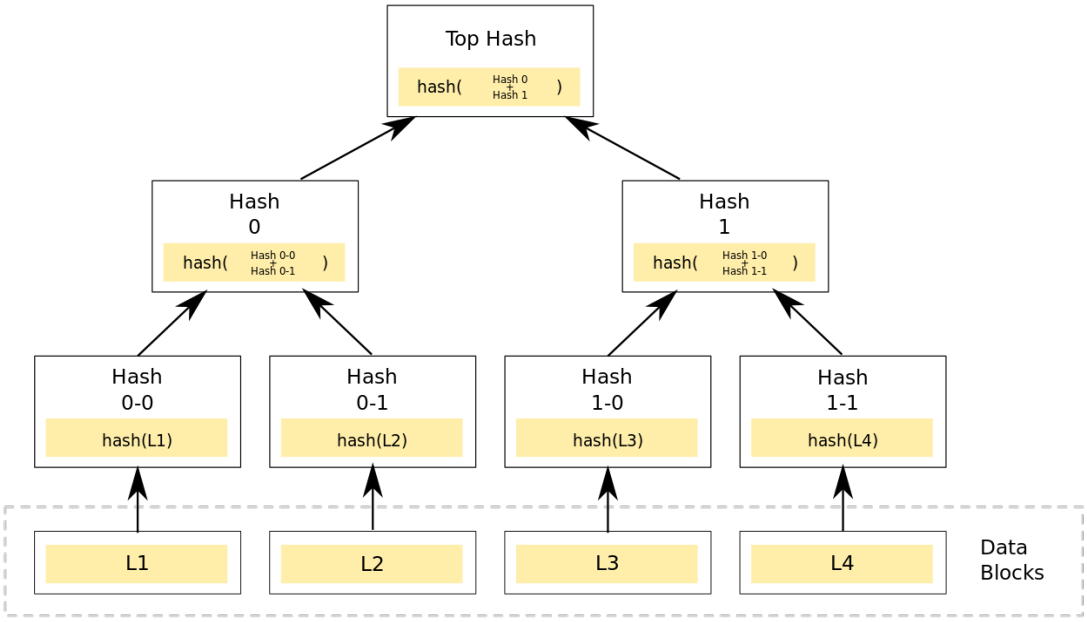

Figure 6: Illustration of Merkle tree architecture.

2.2 Hash Functions

We focus on Merkle tree (also called hash tree). Merkle tree is to verify the integrity of data. Figure 6 illustrates a typical hierarchical structure of a Merkle tree, which is a type of binary tree used in cryptography to summarize a set of data blocks through hashing. It is a practical example of constructing a Merkle tree, a widely-used structure in ensuring data integrity in distributed systems and blockchain technology. At the bottom of Figure 6, we have the leaf nodes labeled L1, L2, L3, and L4. These represent individual blocks of data that are to be summarized. Each leaf node is hashed, resulting in hash values Hash 0-0, Hash 0-1, Hash 1-0, and Hash 1-1. The hash function is represented by hash(). Hashing is a process that takes an input (or 'message') and returns a fixed-size string of bytes. The output (the hash value or hash code) is typically a 'digest' that represents the original string of data. These hashed values are then combined in pairs and hashed again to produce the parent nodes, Hash 0 and Hash 1. For example, Hash 0 is a result of hashing the combination of Hash 0-0 and Hash 0-1. Finally, the hash values of the parent nodes are combined and hashed to produce the "Top Hash," which is the root of the Merkle tree. This hash is a single value that uniquely represents the entire set of data blocks (L1, L2, L3, and L4). The parallelization of Merkle tree can be achieved as follows: For each level of depth, we hash the level in parallel rather than do it in sequence. This is because the

Table 5: Performance comparison in running time (s) ↓ between parallelized hash tree and sequential hash tree

	Merkle tree-parallel (Ours)	Merkle tree-sequential
image	**0.5767**	0.6698
text	**0.00055**	0.00158

hashing of each block in the same level does not affect other blocks. The data blocks and their combined hashes are processed in batches, which is a more efficient approach than processing them sequentially.

Uses: Merkle trees serve as robust verification tools for data that is stored, managed, and transferred across computer systems. They are particularly valuable in peer-to-peer networks for ensuring that data blocks are intact and authentic, safeguarding against the receipt of corrupted or falsified blocks. A significant benefit of Merkle trees is their ability to validate data incrementally: individual branches can be downloaded and verified for integrity in isolation, without the need for the entire tree. For instance, as illustrated in Figure 6, the authenticity of data block L2 can be confirmed promptly if the tree already includes hash 0-0 and hash 1. By hashing L2 and sequentially combining it with hash 0-0, and then hash 1, we can compare the outcome with the top hash to verify its integrity. This method is efficient because it allows for the division of files into very small data blocks, making it so that only minimal amounts need to be retransmitted in case of any damage. Using a Merkle tree enables rapid branch downloads, immediate integrity checks, and the sequential initiation of data block downloads.

Setup: Our method employs a streamlined parallelized process to construct a Merkle tree, a pivotal component in the domains of cryptography and blockchain technology. The foundational elements of our methodology are delineated below: We deploy the *Crypto.Hash* library to execute cryptographic hashing and utilize numpy for its robust numerical computation and efficient array handling capabilities. Within this framework, the SHA256 algorithm from the Crypto.Hash library is used to hash individual blocks of data. For the assembly of the Merkle tree, the data is segmented into blocks with a predefined size of 32 bytes. Padding is applied where necessary to ensure uniform block sizes. The tree is then incrementally built by hashing individual blocks and their subsequent parent nodes, continuing this process until we arrive at the singular root hash, which serves as the cryptographic fingerprint of the entire data set.

Rsults: The table presents a comparative analysis of the running times for constructing Merkle trees using parallelized and sequential approaches, with a clear emphasis on the improved efficiency of the parallelized method. For image data, the parallelized Merkle tree construction, as indicated by our method, recorded a significant reduction in time, taking only 0.5767 seconds, as opposed to the 0.6698 seconds required by the sequential approach. Similarly, with text data, the parallelized version demonstrated a more pronounced decrease in running time, completing the task in just 0.00055 seconds, compared to the 0.00158 seconds of the sequential counterpart. These results underscore the time-saving advantages of the parallelized process in Merkle tree construction. However, it is important to note that the time cost of the Merkle tree is significantly affected by the sequential encoding of each depth level, a process that inherently cannot be parallelized, indicating a limit to the time savings possible even with a parallelized approach.

3 Conclusions

In conclusion, this project has successfully highlighted the effective utilization of parallel processing in cryptographic algorithms, particularly within symmetric key algorithms like AES in Counter Mode (CTR) and Electronic Code Book (ECB) modes, and in the decryption process of Cipher Block Chaining (CBC) mode. Additionally, the application of parallel hashing in the

Merkle tree structure demonstrates the vast potential of parallelism in enhancing the efficiency of cryptographic operations. However, it is important to acknowledge that this study's scope was limited to specific algorithms and methods. Notably, there are other parallelizable cryptographic algorithms, such as RSA, which we could not explore due to time constraints. Future research should extend into these areas, including the exploration of designing cryptographic algorithms optimized for GPU architecture. Moreover, with the advent of quantum computing, it is also interesting to explore adapting and developing cryptographic algorithms to leverage the quantum computing architecture.