Comparison between sequential and parallel programming on Data Encryption Standard (DES) with asyncio

Alessio Chen
E-mail address
alessio.chen@stud.unifi.it

Abstract

This report aims to compare the performance of sequential and parallel Data Encryption Standard (DES) algorithms using the asyncio concurrent library.

1. Introduction

In this report, we explore the performance of sequential and parallel implementations in simulating brute force attacks on DES-encrypted passwords. Our focus is to evaluate the efficacy of asyncio, a concurrent library, in comparison to traditional sequential algorithms. Through benchmarking execution times, measuring speedups and efficiency gains, and examining scalability across different input sizes, we aim to provide insights into the capabilities of asyncio in cryptography's tasks.

1.1. Asynchronous I/O

Asyncio is a Python library introduced in Python 3.4. It serves as a powerful framework for dealing with asynchronous operations in Python. It excels in scenarios requiring high performance and scalability, such as web servers, database queries, and network connections. By enabling code to be paused and resumed at specific points, particularly during I/O operations, Asyncio maximizes resource utilization and enhances application responsiveness.

It operates on the premise of an event loop, which manages the execution of asynchronous tasks. When a task encounters an I/O operation, it is paused, and control is returned to the event loop. The event loop proceeds to execute other tasks until the I/O operation completes. Once finished, the task is resumed, ensuring efficient utilization of CPU resources and improved responsiveness.

1.2. Data Encryption Standard DES

DES is a symmetric key algorithm developed by IBM in the early 1970s. Recognized for its pivotal role in cryptography, DES was formally adopted as a federal standard for the United States by the National Institute of Standards and Technology (NIST) in 1977. It quickly became the go-to encryption algorithm for the U.S. government, military, and various industries, owing to its robust security and efficiency in electronic data encryption.

At its core, DES operates by transforming 64-bit blocks of plain-text into corresponding cipher-text blocks. This process unfolds through a series of steps organized within a Feistel network structure (1, a design that enhances both security and computational efficiency.

The breakdown of the DES process involves several key steps:

1. **Initial Permutation (IP)**: The 64-bit plain-text block undergoes an initial permutation, where the positions of its bits are rearranged according to a predefined table.

2. **Key Generation**: DES employs a unique key for encryption and decryption operations. The 56-bit key undergoes a series of transformations to generate sixteen 48-bit subkeys, one for each round of encryption.

3. **Rounds of Encryption**: Each round of DES encryption involves intricate operations (Figure 2) on the plain-text block. This includes expansion, substitution through S-boxes, permutation via P-boxes, and XOR operations with the round-specific subkey.

4. **Final Permutation (FP):** After 16 rounds of encryption, the cipher-text block undergoes a final permutation. This step completes the encryption process, yielding the final 64-bit block of cipher-text.

1.3. Setup

The development was conducted on a Mac Mini M2 (2023) with Apple Silicon M2, an 8-core (4 performance + 4 efficiency) chip, running macOS Ventura 14.2.1.

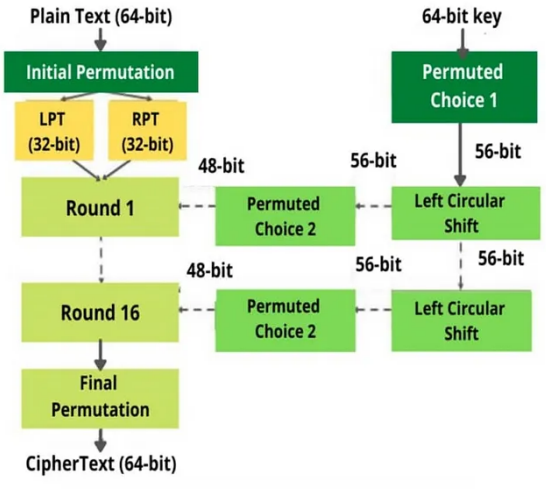

Figure 1: The overall Feistel structure of DES

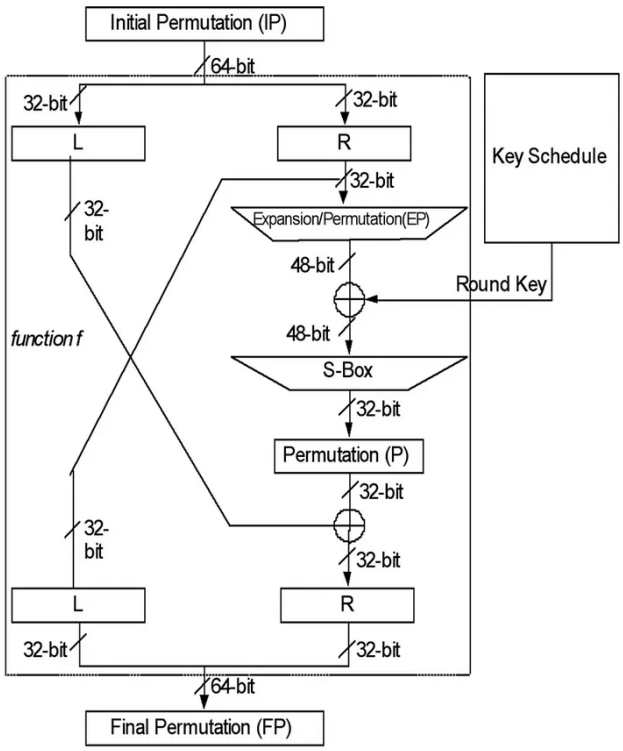

Figure 2: Permutations scheme

2. Methods

In this section, we will present the code for comparing the performance of decrypting encrypted passwords using both sequential and parallel approaches. Our objective is to decrypt a set of randomly generated passwords, each consisting of 8 characters selected from the set [a-zA-Z0-9./].

To do this, we will follow these steps:

- Generate a file containing N random plain text with corresponding cipher-text.
- Read the file, decrypt the cipher-text one by one, and check if it matches the input plain text.

To collect the input data, we utilize the methods described in Listing 1. This method saves the input data in a map where the key represents the plain-text, and the corresponding value corresponds to the cipher-text.

Listing 1: Read From File

```
def generate_words(N: int, length: int = 8) ->
    None:

    char_set = string.ascii_letters +
               string.digits + './'
    words = {}

    for _ in range(N):
        w = ''.join(random.choice(char_set)
                    for _ in range(length))
        words[w] = des.encrypt(w)

    json_object = json.dumps(words, indent=4)

    with open("words.json", "w") as f:
        f.write(json_object)

def read_words_from_file() -> dict[str,str]:
    with open("words.json", 'r') as f:
        words = json.load(f)
    return words
```

To execute this analysis, we utilized a standard implementation of the DES algorithm to encrypt and decrypt the passwords. We chose not to focus extensively on the code related to the implementation of the algorithm since our primary focus was on optimizing the search concurrency in the parallel version, rather than the functions within it.

2.1. Sequential Implementation

The sequential approach iterates through a dictionary containing plain-text passwords and their corresponding encrypted versions. For each pair, it converts the encrypted password to binary, decrypts it using a DES decryption function, and compares the result with the original plain-text password. If a mismatch is detected, an exception is raised.

Listing 2: Sequential implementation

```
def sequential_solver(words: dict[str, str])
    -> None:

    for password, encrypted in words.items():
        enc_to_bin = str_to_bin(encrypted)
        decrypted = des.decrypt(enc_to_bin)

        if password != decrypted:
            raise Exception(f"{password} and {
                decrypted} are not matching")
```

2.2. Parallel Implementation

The asynchronous function orchestrates the parallel execution of decryption tasks using asyncio.

The function initiates the process by segmenting the input data into intervals, a task accomplished by invoking the *get_intervals* function (see Listing 4). These intervals delineate distinct portions of the input data, each earmarked for concurrent processing.

Following the interval setup, the function concurrently executes decryption tasks utilizing asyncio in tandem with a ProcessPoolExecutor. Each thread undertakes the execution of the function described in Listing 5.

Listing 3: Parallel implementation

```
async def parallel_solver(words: dict[str, str
    ], passwords: list[str], n_threads: int)
    -> None:

    intervals = get_intervals(words, n_threads
        )

    with ProcessPoolExecutor() as process_pool
        :
        loop: AbstractEventLoop =
            asyncio.get_event_loop()
        calls: List[partial] =
            [
                partial(parallel, words,
                    passwords, interval)
                for interval in intervals
            ]
        call_coros = []

        for call in calls:
            call_coros.append(loop.
                run_in_executor(process_pool,
                call))


        await asyncio.gather(*call_coros)
```

Listing 4: Calculate intervals

```
def get_intervals(words: dict[str, str],
    n_threads: int) -> [[int, int]]:
    N = len(words)
    interval_size = N // n_threads
    intervals = []
    start_i = 0
    for _ in range(n_threads):
        end_i = start_i + interval_size
        if end_i > N:
            end_i = N - 1

        intervals.append([start_i, end_i])
        start_i = end_i + 1

    return intervals
```

Listing 5: Code executed by each thread

```
def parallel(words: dict[str, str], passwords:
     list[str], interval: [[int, int]]) ->
    None:

    start_i, end_i = interval[0], interval[1]

    for i in range(start_i, end_i):
        enc_to_bin = utils.str_to_bin(words[
            passwords[i]])
        decrypted = des.decrypt(enc_to_bin)

        if passwords[i] != decrypted:
            raise Exception(f"{password} and {
                decrypted} are not matching")
```

3. Results

To evaluate the performance of the parallel solution, we conducted tests with varying numbers of passwords, and the results are shown in Figure 3.

Each test was repeated **10 times** to ensure statistical robustness and reliability in the results.

As observed in the graph, all timelines exhibit a consistent pattern, with the execution time decreasing as the number of threads increases up to 8 threads. Beyond 8 threads, the execution time stabilizes, suggesting that the asynchronous execution of decryption tasks using asyncio scales effectively with the number of threads, resulting in improved performance up to a certain threshold.

Additionally, we analyzed the related speedups, as shown in Figure 4. The maximum speedup achieved was approximately 4.8x with 8 threads.

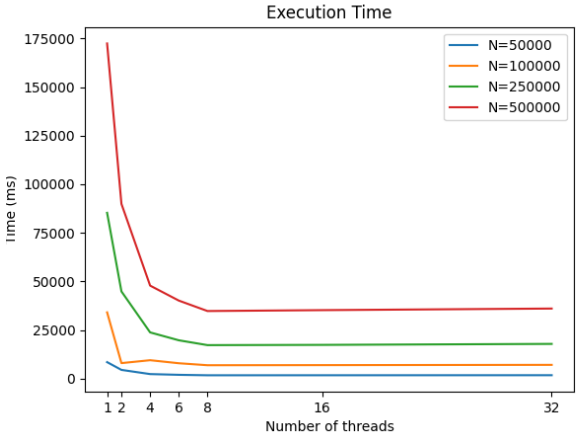

Figure 3: Time decrypting passwords, incrementing threads

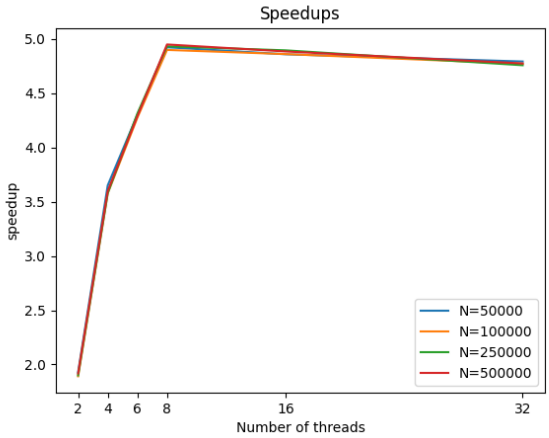

Figure 4: Speedups decrypting passwords, incrementing threads

4. Conclusion

In conclusion the utilization of asyncio and parallel processing techniques allowed for efficient and concurrent execution of decryption tasks, resulting in reduced execution times compared to the sequential approach. This highlights the effectiveness of leveraging parallelism to enhance the performance of DES algorithm implementations.