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

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

Abstract

This report aims to compare the efficiency of sequential and parallel approaches in cracking passwords encrypted with the Data Encryption Standard (DES) algorithm, using the `asyncio` *concurrent library.*

1. Introduction

This report investigates the performance of sequential and parallel implementations for simulating brute force attacks on cracking passwords encrypted with DES. Our objective is to assess the effectiveness of `asyncio`, a concurrent library, compared to traditional sequential algorithms. By benchmarking execution times, analyzing speedups and efficiency gains, and evaluating scalability across varying password lengths, we aim to gain insights into asyncio's capabilities in cryptography 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 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** (Figure 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.

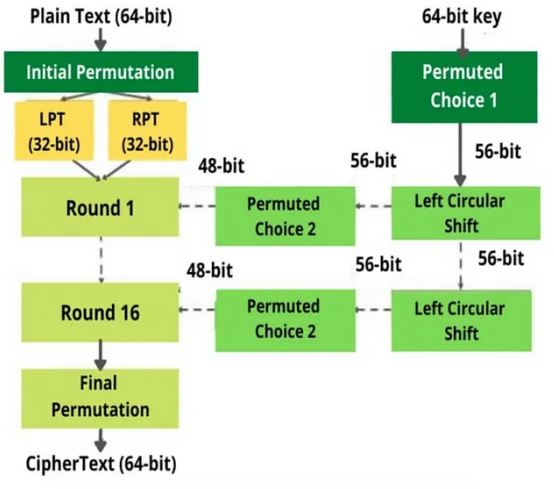

Figure 1: The overall Feistel structure of DES

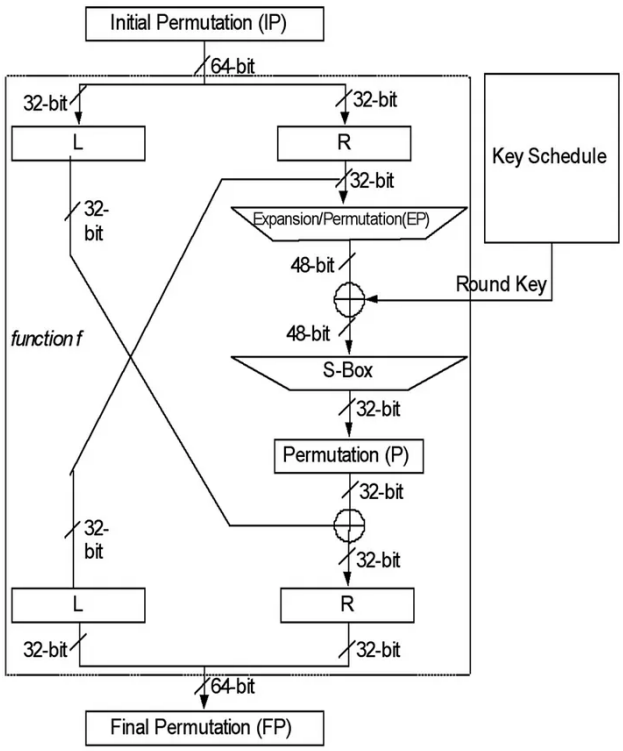

Figure 2: Permutations scheme

2. Methods

In this section, we will present the code for comparing the performance of cracking 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 [0-9./]**. These passwords are specifically designed as 8-digit numbers representing dates (es. 20240325).

To do this, we will follow these steps:

- Generate N cipher-text from random plain text.
- For each generated cipher-text, iterate through a list of all possible passwords, encrypting each one by one, and check if it matches the input cipher-text.

To generate the cipher-texts, we utilize the method described in Listing 1.

Listing 1: Cipher-texts generation

```
def generate_random_passwords(N: int,
    start_date: datetime, end_date: datetime)
    -> list[str]:

    passwords = []
    for i in range(N):

        random_date = start_date + datetime.
            timedelta(days=random.randint(0, (
            end_date - start_date).days))

        encrypted_password = des.encrypt(
            random_date.strftime("%Y%m%d"))

        passwords.append(encrypted_password)

    return passwords
```

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 function for cracking encrypted passwords takes two parameters:

- `passwords`, a list of encrypted passwords to be cracked, and
- `dates`, a list of candidate plain-text passwords (specifically, dates) to try for decryption.

Within the function, it iterates over each password in the passwords list and, for each password, iterates over each date in the dates list. It then encrypts each date using the DES algorithm and compares the result with the encrypted password. If a match is found, it implies the correct password has been decrypted.

Listing 2: Sequential implementation

```
def sequential_crack(passwords: list[str],
    dates: list[str]) -> None:

    for password in passwords:
        for date in dates:

            encrypted = des.encrypt(date)

            if encrypted == password:
                print(f"Password found : {des.
                    decrypt(utils.str_to_bin(
                    encrypted))}")

                break
```

2.2. Parallel Implementation

The parallel approach initially divides the list of dates into chunks based on the number of threads (Listing 4). Each chunk represents a subset of dates to be processed concurrently. It then utilizes the `ProcessPoolExecutor` from the `concurrent.futures` module to create a pool of processes, enabling parallel execution.

For each password in the passwords list, the function creates a set of coroutines, each responsible for processing a chunk of dates. These coroutines are executed concurrently using `asyncio.gather()`, facilitating efficient parallel processing and maximizing resource utilization (Listing 3).

Listing 3: Parallel implementation

```
async def parallel_crack(passwords: list[str],
     dates: list[str], n_threads: int) -> None
    :

    chunks = get_chunks(n_threads, len(dates))

    with ProcessPoolExecutor() as process_pool
        :
        for password in passwords:
            loop: AbstractEventLoop = asyncio.
                get_event_loop()

            calls: List[partial] = [
                partial(parallel, password,
                    chunk, dates)
                for chunk in chunks
            ]

            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_chunks(n_threads: int, N: int) -> list
    [list[int, int]]:

    chunk_size = N // n_threads
    chunks = []
    start_i = 0
    for _ in range(n_threads):
        end_i = start_i + chunk_size
        if end_i > N:
            end_i = N - 1
        chunks.append(([start_i, end_i]))
        start_i = end_i + 1

    return chunks
```

Listing 5: Code executed by each thread

```
def parallel(password: str, chunk: [[int, int
    ]], dates: list[str]) -> None:

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

    for i in range(start_i, end_i):
        encrypted = des.encrypt(dates[i])
        if encrypted == password:
            print(f"Password found: {des.
                decrypt(utils.str_to_bin(
                encrypted))}")
            break
```

3. Tests

Two distinct tests were carried out:

- The first test involved assessing performance variations by incrementally increasing the number of threads to crack a single password.
- The second test focused on evaluating performance changes by expanding the number of passwords searched and variating the number of threads.

Both tests utilized a dataset comprising dates from **1900/01/01 to 2024/12/31**, resulting in a search space of 45,260 plain-text passwords. To mitigate any potential bias arising from the randomness of the generated cipher-text, each test was repeated **10** times. Average results were then calculated and analyzed to provide a more reliable assessment of performance across different scenarios.

3.1. Fixed passwords number

The results for the first test, where the number of passwords is fixed to 1 and the number of threads is increased up to 32, are shown in Figure 3.

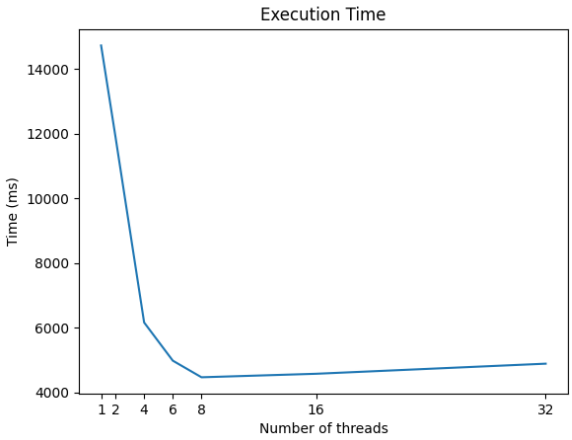

Figure 3: Time cracking 1 password, incrementing threads

As expected, the time decreases from 14732 ms in the sequential version down to 4474 ms in the parallel version with 8 threads (which was the expected best performance in this setup), then increases as the number of threads increases.

Then we can see related speedups shown in Figure 4.

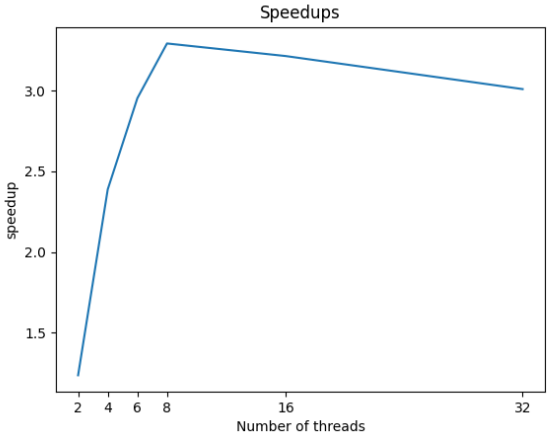

Figure 4: Speedups cracking 1 password, incrementing threads

3.2. Fixed threads numbers

The results of the second test, where the number of threads increases and the number of passwords searched is increased up to 20, are shown in Figure 5.

The decrease in performance observed with 2 threads compared to 1 thread can be attributed to the overhead associated with parallelization, particularly noticeable when the number of threads is limited. This overhead includes task scheduling, inter-thread communication, and synchronization, all of which can introduce delays and consequently increase execution times.

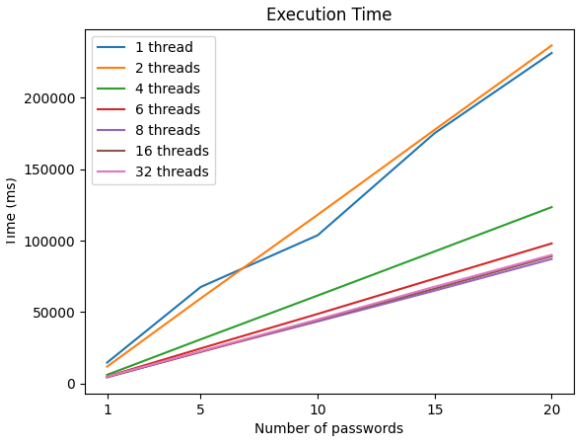

Figure 5: Time cracking N password, incrementing threads

However, as a general trend, increasing the number of threads tends to result in improved performance in terms of execution time. With more threads available, the potential benefits of parallelization become more pronounced.

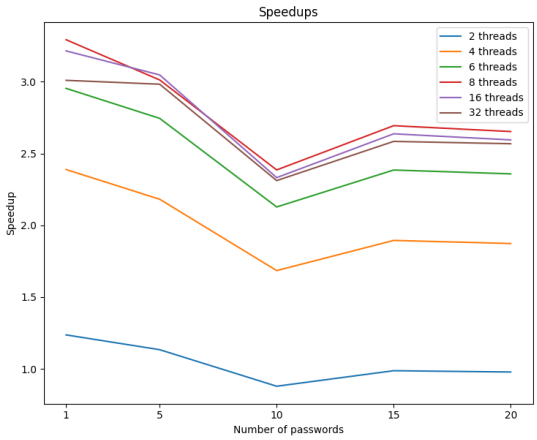

Figure 6: Speedups cracking N password, incrementing threads

Then we see the related speedups shown in Figure 6.

Overall, the speedup tends to increase as the number of threads increases, indicating that parallel execution provides increasingly better performance gains compared to sequential execution. However, the speedup may not scale linearly with the number of threads due to factors such as overhead and resource contention.

4. Conclusion

In conclusion, the comparison between sequential and parallel approaches for cracking DES-encrypted passwords using `asyncio` revealed important insights into their respective performances. While parallelization generally led to improved execution times, especially with larger thread counts, overhead associated with limited thread counts could sometimes outweigh the benefits, as seen in the case of 2 threads. Nonetheless, overall trends showcased the scalability and efficiency gains achievable through parallel processing.