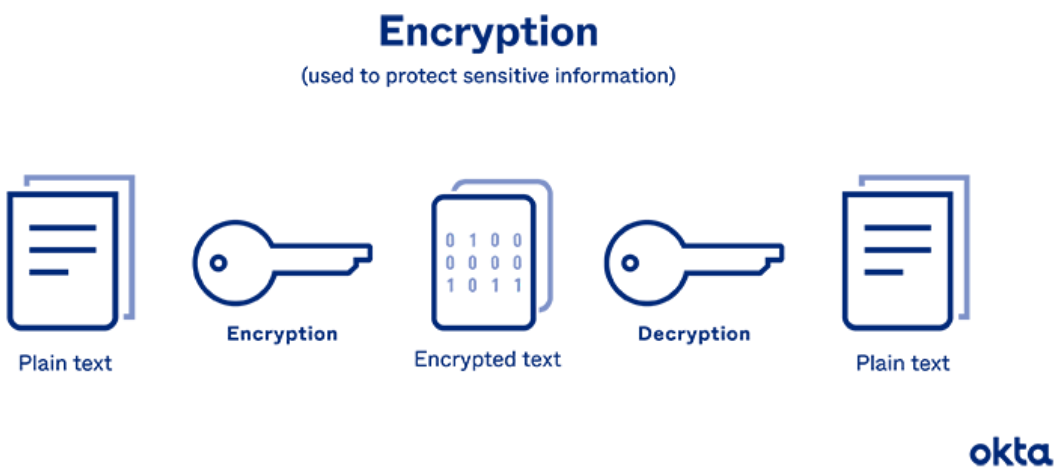
TECH REPORT:
PASSWORD DECRYPTION



By Clara Alejos | Gonzalo Ortiz
Parallel Programming For Machine Learning

INDEX

1. INTRODUCTION

1.1 Objective

The main objective of this work is to analyze and compare the performance of an algorithm designed to decrypt encrypted passwords using two approaches: sequentially and in parallel using OpenMP.
The sequential approach iterates linearly through a search space of possible passwords, while the parallel approach divides the task into different threads, reducing execution time in most cases. This task aims to observe the advantages and limitations of parallelization in this type of problem.

1.2 Importance

Password decryption is a crucial task in the field of cybersecurity. Moreover, this problem becomes quite inefficient in sequential systems as the size of the passwords increases.

2. PROBLEM DESCRIPTION

2.1 DES Encryption

DES operates on 64-bit data blocks using a 56-bit key. The "crypt" function used is an implementation for password hashing. Decryption also involves brute force, searching through a predefined space of possible combinations.

2.2 Search Space

For this work, we have tested different lengths. Finally:

- The passwords are 5 characters long.
- The character set includes [a-zA-Z0-9./] (64 possible characters).
- We use a known SALT of two characters "1g".

2.3 Challenges

- Managing the high computational cost due to the large search space.
- The need for efficient task division in parallel execution

3. METHODOLOGY

3.1 Before the implementations

To carry out the password decryption experiments, it was necessary to generate a list of encrypted passwords with the following characteristics:

- Fixed-length passwords (5 characters).
- Encrypted using the C "crypt" function, with a known SALT of two characters ("1g" in this case).
- Based on common passwords, randomly selected from an input list (rockyou.txt).
- Limited to a total of 100 encrypted passwords.

The main goal of this program is to provide a set of real password hashes to simulate a controlled environment where the performance and effectiveness of the implementations can be measured.

- **Reading Common Passwords**
- **Random Selection of Passwords**
- **Password Encryption**
- **Storing the Result**

3.2 Sequential Implementation

All possible password combinations are generated, evaluating them one by one until a match is found with the target hash. The key steps are:

- **Password Generation:**
  A recursive function is used to explore all possible password combinations. Each combination is tested by generating its hash using "crypt" and comparing it with the target hash.
  If a password generates a hash that matches the target, the program stops the search and prints the found password.
- **Time Measurement:**
  The execution time of the search is measured using the standard "chrono" library, recording the start and end times of the decryption process.
- **Execution:**
  For each hash read from the file, a function is called that invokes the recursive function to test all possible combinations. The result (success or failure) is logged in a file.

This approach ensures that all combinations are explored sequentially, but the execution time increases exponentially with the number of characters and password length.

3.3 Parallel Implementation

The parallel version uses OpenMP to distribute the password generation and verification across multiple threads. This approach speeds up the process by allowing multiple combinations to be tested simultaneously. The key steps are:

- **Iterative Password Generation:**
  Unlike the recursive implementation, an iterative approach based on indexes is used here. Each possible password combination is associated with a unique index generated by calculating all possible combinations.
  The indexes are dynamically distributed among the threads. Each thread generates and evaluates passwords within its assigned index range.
  To ensure that only one thread logs the correct password in case of a match, a critical section is used.
- **Synchronization and Data Sharing:**
  A shared variable is employed so that threads can detect if a password has already been found. This prevents unnecessary searches after a solution is found.
- **Parallel Execution per Hash:**
  For each hash in the input list, a set of threads is assigned to perform the parallel search.
- **Time Measurement:**
  Similar to the sequential version, the execution time is measured for each hash, allowing for a performance comparison between the implementations.
- **Result Output:**
  The decryption results, along with execution times, are logged in a file

3.4 Implementation Comparisons

The main differences between the two implementations are:

- In the sequential version, the combinations are generated and evaluated recursively, which limits performance.
- In the parallel version, the combinations are divided among several threads, speeding up the process through parallelization. Additionally, the iterative approach is better suited for dynamic work distribution.

The parallel implementation proves to be significantly more efficient for large hash lists, making the most of the available hardware resources. However, more care is needed to avoid synchronization issues between threads.

3.5 Experimental Setup

- **Hardware:** 2-core-CPU
- **Software:** Compiler GCC with support OpenMP, environment Google Colab
- **Libraries:** <iostream>, <fstream>, <string>, <vector>, <unistd.h>, <chrono>, <omp.h>, <cmath>, <algorithm>

4. RESULTS

4.1 Performance Metrics

- **Execution Time:** Total time taken by the sequential and parallel implementations.
- **Speedup and Efficiency**

$$S = \frac{Tsequential}{Tparallel} \qquad E = \frac{S}{Num\ of\ threads}$$

4.2 Observations

The sequential version was executed for only 5 passwords due to its high computational cost. It took 4557.402 seconds, so to estimate the time it would have taken to process 100 passwords, we assume the time grows linearly. By calculating the average time per password and multiplying by 100, we estimate this value.

4.3 Tables

Number of Threads	Execution Time (sec)	Time per Password (sec)
Sequential	91148.04	911.48
2 Threads	4742.92	47.43
4 Threads	4597.31	45.97
8 Threads	4469.97	44.70
16 Threads	2934.88	29.35

Number of Threads	Speedup (S)	Efficiency (E)
2 Threads	19.22	9.61 (96%)
4 Threads	19.38	4.96 (49%)
8 Threads	20.39	2.55 (25%)
16 Threads	31.06	1.94 (12%)

5. DISCUSSION

5.1 Comparison

- The parallel implementation significantly reduced execution time compared to the sequential version.
- Speedup shows diminishing returns as the number of threads increases, likely due to synchronization overhead and resource contention.
- The best performance gain was observed between 2 and 8 threads, after which the efficiency began to decline.

5.2 Lessons Learned

Managing multiple threads introduces overhead, which can reduce the benefits as concurrency increases. If parallelization were perfect, the speedup should be close to N threads. However, in this case, efficiency drops from 4.96 (4 threads) to 2.55 (8 threads) and then to 1.94 (16 threads), indicating that overhead starts to impact performance.

6. CONCLUSIONS

6.1 Key Findings

- The parallel implementation significantly outperformed the sequential one, achieving a maximum speedup of 31.06x.
- Efficiency decreases as the number of threads increases, indicating diminishing returns due to system overhead.
- The optimal number of threads for this problem appears to be 8, as it provides a good balance between speed and efficiency.

6.2 Future Improvements

- **Better Load Balancing:** Implement dynamic work distribution among the threads.
- **Vectorization:** Use SIMD instructions to improve performance per thread.

7. REFERENCES

- OpenMP: https://www.openmp.org/
- GNU C Library: https://www.gnu.org/software/libc/