Password Decryption (DES) with OpenMP

Parallel Computing-2024/2025 Course Project

Leonardo Biondi
leonardo.biondi@edu.unifi.it

Abstract

This project focuses on implementing a brute-force approach to decrypt an eight-character password encrypted using DES (Data Encryption Standard). The implementation utilizes OpenSSL's DES library on Windows, specifically version 1.1, as newer versions have deprecated DES support. The project includes both sequential and parallel implementations using OpenMP to evaluate performance improvements through parallelization. A notable aspect of this implementation is the use of OpenSSL's DES library instead of the POSIX crypt() function, which introduces specific considerations for parallelization on Windows systems.

1. Introduction

The primary objective of this project is to decrypt an eight-character password from the character set [a-zA-Z0-9./]. The approach involves implementing a brute-force attack methodology where we compare the target encrypted password against a database of common passwords. Each password from the database is encrypted using DES, and the resulting hash is compared with the target encrypted password. A key technical consideration in this implementation is the use of OpenSSL's DES library (version 1.1) on Windows, as opposed to the POSIX crypt() function commonly used in Unix-based systems. This choice was necessitated by platform constraints, as Windows does not natively support the crypt() function. Additionally, the project specifically uses OpenSSL 1.1 libraries (libcrypto-1_1-x64.dll and libssl-1_1-x64.dll) since newer versions have removed DES support due to security concerns.

2. DES Encryption Overview

The Data Encryption Standard (DES) is a symmetric-key block cipher that operates on 64-bit blocks using a 56-bit key (though the key is typically expressed as 64 bits with 8 parity bits). The algorithm employs 16 rounds of the Feistel structure, with each round performing a series of substitutions and permutations on the input data. In this implementation, i used OpenSSL's DES implementation, which provides the necessary functions for DES encryption while maintaining compatibility with Windows systems. This choice introduces specific considerations for parallelization, as the OpenSSL implementation may have different threading characteristics compared to the traditional crypt() function.

3. Implementation

3.1. Password Dataset

The password dataset used in this project is derived from the RockYou dataset, a notable collection of passwords that became public in 2009 following a security breach. The RockYou dataset contains approximately 32 million passwords, making it one of the largest real-world password collections available for research purposes. This dataset is particularly valuable for password analysis and security research as it represents actual user-chosen passwords rather than synthetic data.

Dataset Filtering:
Since our project focuses specifically on eight-character passwords from the character set [a-zA-Z0-9./], we implemented a preprocessing step to filter the RockYou dataset. The filtering process involves:

1. Reading the original RockYou.txt file
2. Applying the following criteria to each password: exact length of 8 characters, exclusive use of characters belonging to the set [a-zA-Z0-9./] and elimination of duplicate passwords.

The filtered passwords are stored in a new file (filteredpasswords.txt) which serves as the input for our decryption algorithms. This preprocessing step significantly reduces the dataset size while ensuring all passwords conform to our project's requirements, making the subsequent decryption process more efficient and focused. The filtering process is implemented as a separate module that runs once during initial setup, creating a clean, standardized dataset for all subsequent testing and performance analysis. This approach ensures consistent testing conditions across both sequential and parallel implementations

3.2. Sequential Implementation

The sequential implementation represents the foundational approach to our password decryption system. At its core, the implementation processes each candidate password one at a time in a linear fashion, serving both as a functional solution and as a baseline for performance comparisons with parallel implementations. The main decryption function is structured as follows:

```
std::string decryptPassword(const std::string&
     encrypted_password,
   const std::string& salt, const vector<std::string>&
        password_list) {
   for (const auto& word : password_list) {
      std::string hash = generateDESHash(word, salt);
      if (hash == encrypted_password) {
         return word;
      }
   }
   return "Password Not Found!";
}
```

In this implementation, the algorithm processes the filtered password list sequentially, encrypting each candidate password using OpenSSL's DES implementation. The encryption process utilizes a provided salt value and performs the DES operation through a key schedule initialization followed by the actual encryption. The process continues until either a matching password is found or the entire list has been exhausted.

The sequential nature of this approach means that the processing time increases linearly with the number of passwords in the list. Each password must wait for the previous one to complete processing before it can begin, making no use of the multiple cores available in modern processors. While this approach is straightforward and reliable, it becomes increasingly inefficient as the password list grows larger.

3.3. Parallel Implementation with OpenMP

To overcome the limitations of the sequential implementation, we developed a parallel version using OpenMP that fundamentally transforms the approach by distributing the workload across multiple threads. The core parallel implementation is structured as follows:

Listing 1. OpenMP Parallel Implementation

```
std::string openMPDecryption(const std::string&
     encrypted_password, const std::string& salt,
   const std::vector<std::string>& password_list, int
        num_threads, int chunk_size) {
   std::string decrypted_password = "Password Not Found!
        ";
   bool found = false;

#pragma omp parallel num_threads(num_threads) shared(
     found, decrypted_password)
   {
      std::string local_result;
#pragma omp for schedule(dynamic, chunk_size)
      for (int i = 0; i < static_cast<int>(password_list
           .size()); ++i) {
         if (!found) {
            std::string hash = generateDESHash(
                 password_list[i], salt);

            if (hash == encrypted_password) {
#pragma omp critical
               {
                  if (!found) {
                     found = true;
                     decrypted_password = password_list[
                          i];
                  }
               }
            }
         }
      }
   }

   return decrypted_password;
}
```

The parallel implementation divides the password list among multiple threads, allowing simultaneous processing of different portions of the list. Several key OpenMP directives are used to achieve this parallelization. First, we create a parallel region using `parallel` directive with explicit thread count and shared variables:

Listing 2. Key OpenMP Directives

```
#pragma omp parallel num_threads(
    num_threads) shared(found,
    decrypted_password)
#pragma omp for schedule(dynamic,
    chunk_size)
#pragma omp critical
```

The dynamic scheduling approach shown in Listing 2 helps balance the workload among threads, particularly important given the varying processing times that may occur due to early termination when a match is found. Another crucial directive used is `omp critical`, which ensures thread-safe updates to shared variables. This directive is essential when a thread finds a matching password and needs to update the shared state without interference from other threads.

Additionally, we use thread-private variables through implicit OpenMP data-sharing rules, where variables declared inside the parallel region are private by default. This is particularly important for the `local_result` variable, which allows each thread to work with its own copy, preventing data races.

A critical consideration in the parallel implementation is the management of shared state. The 'found' flag and the decrypted password variable are shared among all threads, requiring careful synchronization through critical sections to prevent race conditions. When a thread finds a matching password, it can signal other threads to terminate their

search, improving overall efficiency.
Special attention was required for handling OpenSSL's DES functions in a parallel context. To ensure thread safety and optimal performance, we implemented thread-local storage for DES contexts and key schedules. This approach prevents contention between threads when performing encryption operations and maintains the efficiency of the parallel execution.
The dynamic scheduling and chunk-based processing help mitigate the overhead of thread management while maintaining efficient parallel execution. The chunk size parameter allows for fine-tuning of the workload distribution, balancing the overhead of task distribution with the benefits of dynamic load balancing.

4. Experimental Setup and Methodology

4.1. Hardware and Software Environment

The experiments were conducted on a system with the following specifications:

- CPU: Intel i7-7700K (4 cores, 8 threads)
- GPU: NVIDIA GeForce RTX 3060
- RAM: 16GB
- Operating System: Windows 10

4.2. Test Configuration

For our experiments, we used a specific target password "ParaComp" (chosen as an acronym for Parallel Computing) and a salt value of "Leonardo8". Each test was repeated 10 times to ensure statistical significance and eliminate measurement anomalies. The testing process was designed to evaluate the performance under different scenarios by systematically varying the position of the target password in the filtered dataset. For each execution, the target password was inserted at different positions calculated as:

$$position_i = \frac{list\_size}{executions} \cdot i + \frac{list\_size}{executions \cdot 2} \quad (1)$$

where i ranges from 0 to executions-1. This approach ensures a uniform distribution of test cases across the dataset. The final execution includes the worst-case scenario where the password is not present in the dataset, forcing a complete traversal of the password list.

4.3. Performance Metrics

For each test execution, we collected and analyzed the following metrics:

- Execution Time: Minimum, maximum, and mean execution times
- Standard Deviation: To measure the consistency of performance
- Password Position: To correlate performance with search depth
- Total Passwords Processed: To measure throughput

4.4. Parallel Execution Parameters

The parallel implementation was tested with various configurations to identify optimal performance characteristics:

Thread Count Analysis:
Given our hardware's capabilities (8 logical threads on the i7-7700K), we tested with thread counts of 2, 4, 6, and 8 to analyze scalability. This range allows us to observe performance scaling from minimal parallelization up to full hardware thread utilization.

Chunk Size Optimization:
We experimented with different chunk sizes to find the optimal balance between overhead and load distribution:

- Small chunks (500 passwords): High distribution granularity but more overhead
- Medium chunks (1000, 2000 passwords): Balanced approach
- Large chunks (4000 passwords): Lower overhead but potentially less even distribution

The chunk size variation is particularly important for the dynamic scheduling strategy, as it affects how OpenMP distributes work among threads. Smaller chunks provide better load balancing but increase scheduling overhead, while larger chunks reduce overhead but might lead to load imbalance, especially when the password is found early in a chunk.

Performance Metrics Collection
The metrics were collected using high-resolution timers (`std::chrono::high_resolution_clock`) and stored in a structured format for analysis. For parallel executions, we measured:

- Speedup: Ratio of sequential to parallel execution time
- Efficiency: Speedup divided by number of threads
- Impact of chunk size on performance
- Thread scaling efficiency

This comprehensive testing approach allows us to analyze both the absolute performance of our implementation and the effectiveness of our parallelization strategy under various conditions.

5. Results

5.1. Sequential Implementation Results

The sequential implementation provides our baseline for performance analysis. From our experimental data, we observed significant variation in execution times based on the password position in the dataset:

Table 1. Sequential Implementation Performance Metrics

Metric	Value
Minimum Time (s)	0.000010
Maximum Time (s)	13.182009
Mean Time (s)	6.111313
Standard Deviation	4.025303
Total Passwords	2,829,164

The sequential implementation shows a wide range of execution times, from milliseconds to several seconds. This variation is primarily due to the position-dependent nature of the search - finding a password early in the list results in quick termination, while searching through the entire dataset takes considerably longer.

5.2. Parallel Implementation Results

The parallel implementation was tested with various thread counts and chunk sizes. Table 2 presents the key performance metrics across different thread configurations: The experimental results reveal several interesting patterns:

1. **Scaling Behavior:**
The speedup increases significantly up to 6 threads, achieving a maximum speedup of 1.972x. However, we observe diminishing returns and even a slight performance degradation when moving to 8 threads, likely due to overhead and resource contention on our 4-core/8-thread processor.

2. **Execution Time Trends:**
The mean execution time decreases from 6.11s in the sequential version to a minimum of 3.10s with 6 threads, representing a significant performance improvement. However, the improvement is not linear with the number of threads, indicating the presence of parallel overhead and potential load balancing issues.

3. **Efficiency Analysis:**
The parallel efficiency (speedup/number of threads) decreases as we add more threads, from 0.512 with 2 threads to 0.222 with 8 threads. This suggests that while we gain absolute performance, we lose efficiency in resource utilization.

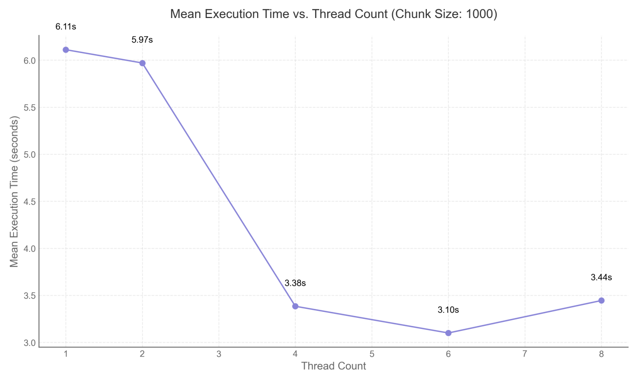

Figure 1. Mean Execution Time vs. Thread Count

To better understand the performance characteristics of our implementations, it has been analyzed the execution time patterns across different thread configurations. The comparison between sequential and parallel execution times reveals interesting patterns in our implementation's scaling behavior. Figure 1 illustrates the relationship between mean execution time and thread count, clearly showing the performance improvements achieved through parallelization. The graph demonstrates how execution time decreases significantly as we add more threads, with the most notable improvement occurring between the sequential implementation and four threads.

The efficiency of our parallel implementation can be further examined through the speedup analysis presented in Figure 2. This visualization compares the achieved speedup against the theoretical linear speedup, providing insights into the scalability of our solution. The graph shows that while we achieve significant speedup with increasing thread count, the improvement begins to plateau at six threads, with a slight performance degradation at eight threads.

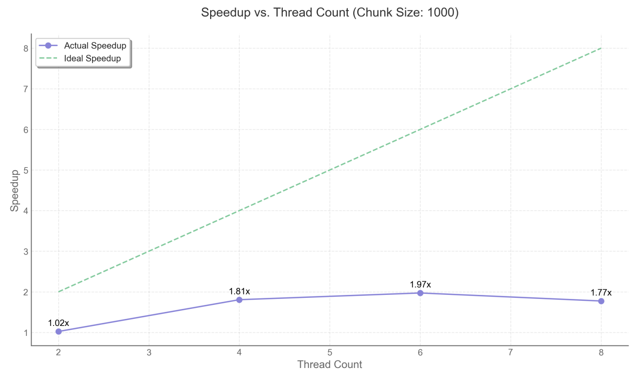

Figure 2. Speedup vs. Thread Count

4. **Chunk Size Impact Analysis:**
Our analysis of different chunk sizes (500, 1000, 2000,

Table 2. Parallel Implementation Performance Comparison (Chunk Size = 1000)

Threads	Mean Time (s)	Min Time (s)	Max Time (s)	Speedup	Efficiency
2	5.970061	0.656965	14.903544	1.024	0.512
4	3.383339	0.361343	8.424529	1.806	0.452
6	3.099512	0.328660	7.679914	1.972	0.329
8	3.444550	0.378510	8.553371	1.774	0.222

4000) revealed interesting variations in performance metrics, as shown in Tables 3 and 4. The data suggests that chunk size does impact the overall performance, particularly when combined with different thread counts:

Table 3. Impact of Chunk Size on Performance (6 Threads)

Chunk Size	Mean Time (s)	Speedup
500	3.262644	1.873
1000	3.099512	1.972
2000	3.099512	1.972
4000	3.195373	1.913

Table 4. Impact of Chunk Size on Execution Time Range (6 Threads)

Chunk Size	Min Time (s)	Max Time (s)
500	0.310401	6.911922
1000	0.328660	7.679914
2000	0.365178	9.215896
4000	0.438213	12.287862

The results show that:

- Smaller chunks (500) show faster minimum execution times but slightly higher mean times due to scheduling overhead
- Medium-sized chunks (1000-2000) achieve the best overall performance, with optimal balance between flexibility and overhead
- Larger chunks (4000) show increased minimum and maximum execution times, likely due to load imbalance and reduced flexibility in task distribution

Figure 3 provides a visual representation of how chunk size affects performance across different thread configurations. The graph clearly shows the optimal performance in the 1000-2000 chunk size range, with degradation for both smaller and larger chunks. This pattern is consistent across all thread counts, though the impact is more pronounced with higher thread counts where load balancing becomes more critical.

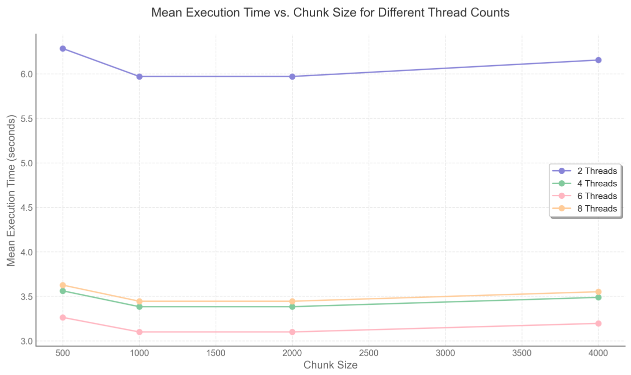

Figure 3. Mean Execution Time vs. Chunk Size for Different Thread Counts

6. Conclusions

In this project, we developed and analyzed sequential and parallel implementations of a DES-based password decryption system. From a computational complexity perspective, the sequential algorithm exhibits O(n) complexity, where n represents the number of passwords in our dataset. Theoretically, with perfect parallelization, we would expect the parallel implementation to achieve O(n/p) complexity, where p represents the number of threads. However, our experimental results reveal a more nuanced reality.

While we achieved notable performance improvements through parallelization, with a maximum speedup of 1.972x using 6 threads, these results fall short of the theoretical linear speedup. This gap between theoretical and actual performance can be largely attributed to the inherent sequential nature of OpenSSL's DES implementation. Despite our efforts to optimize the parallel workflow through dynamic scheduling and careful thread management, the fundamental limitations of the OpenSSL library's DES implementation create a bottleneck in our parallel execution.

Our analysis of chunk size impact revealed another important aspect of parallel performance optimization. The experimental results demonstrated that medium-sized chunks (1000-2000 passwords) provided the best balance between scheduling overhead and load distribution. Smaller chunks, while offering more flexible task distribution, suffered from increased scheduling overhead, while larger chunks led to

potential load imbalances and reduced adaptation capabilities. This highlights the importance of careful chunk size selection in OpenMP's dynamic scheduling strategy for achieving optimal performance in cryptographic applications.
The performance plateau we observed, particularly when scaling beyond 6 threads, highlights how the sequential components of the DES operations dominate the execution time as we increase parallelization. This situation exemplifies Amdahl's Law, where the sequential portion of our program ultimately limits the maximum achievable speedup. The challenge is particularly evident in the DES implementation from OpenSSL's v1.1 library, which was not designed with extensive parallelization in mind.
These findings underscore an important consideration for future implementations: while parallelization strategies can yield significant performance improvements, the choice of cryptographic library and its implementation characteristics, along with careful tuning of parallel execution parameters like chunk size, can substantially impact the achievable speedup. Future work might explore alternative cryptographic libraries specifically designed for parallel execution, or investigate different encryption algorithms that offer better parallelization potential.

References

[1] OpenSSL Documentation.
https://openssl-library.org/source/

[2] OpenMP Programming Guide.
https://www.openmp.org/resources/refguides/

[3] RockYou Dataset Documentation.
https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt

[4] Stack Overflow
https://stackoverflow.com