Parallel Password Decryption using OpenMP: Performance Analysis and Optimization

Lapo Chiostrini
University of Florence
lapo.chiostrini1@edu.unifi.it

Abstract

This report presents a comprehensive analysis of parallel password decryption using OpenMP for brute-force attacks on DES-encrypted passwords. The study implements and compares three algorithmic variants: a sequential baseline, a parallel implementation with implicit barriers, and an optimized parallel version with nowait clauses. Performance benchmarks across 1-32 threads reveal interesting overthreading behavior: the parallel NOWAIT implementation achieves a maximum speedup of 12.04× with 25 threads, processing approximately 4.42 million passwords per second, but performance degrades slightly at 32 threads due to excessive context switching and resource contention. The analysis examines execution time, speedup, efficiency, scalability, and throughput characteristics, demonstrating the effectiveness of OpenMP parallelization for computationally intensive cryptographic operations while highlighting the optimal threading sweet spot and the costs of overthreading beyond physical core counts.

1 Future Distribution Permission

The author of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

2 Introduction

Password security remains a critical concern in modern computing systems. Understanding the computational effort required to crack encrypted passwords through brute-force methods provides valuable insights into password strength and the effectiveness of parallelization strategies for cryptographic applications. This project implements a parallel brute-force password cracking system targeting 8-character date-based passwords (format: ddmmyyyy) encrypted using the DES algorithm.

The primary objective is to evaluate the performance characteristics of OpenMP-based parallelization compared to sequential execution, analyzing how different thread counts and optimization strategies affect execution time, speedup, efficiency, and overall throughput. The implementation tests passwords in the date range from 01011970 to 31122025, representing a search space of approximately 844,000 combinations per iteration.

2.1 Problem Statement

Given a DES-encrypted password hash with salt ”AB”, the goal is to systematically test all possible 8-digit date combinations (ddmmyyyy format) to find the original plaintext password. The computational challenge lies in the cryptographic overhead of the *crypt()* function, which must be called for each candidate password. This makes the problem an ideal candidate for parallelization, as each password test is independent and computationally expensive.

2.2 Contributions

This project makes the following contributions:

- Three algorithmic implementations: Sequential baseline, parallel with barriers, and parallel with

nowait optimization

- Comprehensive performance analysis: Benchmarking across 1, 2, 4, 8, 16, and 20 threads
- Analysis of overtherad performances with 25 and 32 threads
- Multiple performance metrics: Speedup, efficiency, scalability, and throughput analysis
- Optimization techniques: Thread-local crypt_data structures, early termination with atomic flags, and barrier elimination

3 Methodology

3.1 Algorithm Design

The brute-force password cracking algorithm consists of three nested loops iterating over days (0-31), months (0-12), and years (0-2025). For each iteration combination, the algorithm constructs an 8-character date string manually, encrypts the candidate password using *crypt_r()* with DES and salt "AB", compares the resulting hash with the target hash, and terminates early upon finding a match using atomic flag synchronization. The search space per iteration is: $32 \times 13 \times 2026 = 842,816$ password candidates.

3.2 Implementation Variants

Sequential Implementation: Single-threaded baseline using standard *crypt()* function with simple early termination using boolean flag and no parallelization overhead.

Parallel Implementation: OpenMP parallel for with collapse(3) directive, thread-local crypt_data structures using *crypt_r()*, atomic boolean flag for early termination, implicit barrier at end of parallel region, and cancellation points for improved early exit.

Parallel NOWAIT Implementation: Identical to parallel version but with nowait clause to eliminate implicit barrier synchronization and reduce thread synchronization overhead.

3.3 Detailed implemented optimizations

To make the performance improvements and measurements repeatable and easy to reason about, the following concrete optimizations were implemented in the codebase. For each item we report the change made and the expected/runtime effect.

- **Manual date string construction (char buffer)**: constructing the 8-character candidate password by arithmetic on integers (e.g. '0' + digit) instead of using higher-level string formatting (streams or sprintf) reduces per-candidate overhead and heap activity. Expected effect: lower CPU time per candidate and less allocator-related jitter.
- **Per-thread crypt_data with *crypt_r()***: each thread uses a separate `crypt_data` struct and the reentrant `crypt_r()` function instead of the non-reentrant `crypt()`. This removes shared-state contention and avoids sequentialization of the cryptographic routine. Expected effect: improved parallel scalability and correct behavior under multithreading.
- **Quick-reject hash comparison**: before performing a full `strcmp()` on the produced DES hash, the implementation checks two representative bytes (positions 3 and 4). Most candidates fail this cheap check, so this reduces expensive string comparisons. Expected effect: fewer full comparisons, measurable CPU savings.
- **Atomic early-termination flag with memory ordering**: an `std::atomic<bool>` flag is used for early termination. The code uses relaxed loads for frequent checks and a release store when setting the flag. This balances correctness with low-overhead polling. Expected effect: fast early exit with minimal synchronization cost.
- **OpenMP cancellation points for prompt exit**: cancellation points and `#pragma omp cancel for` are used so threads can stop iterating promptly when a match is found. Note: cancellation implies implicit synchronization semantics; in particular, cancellation cannot be combined with the `nowait` clause on the same construct. Expected effect: faster

termination when the password is found, at the cost of some synchronization semantics that must be accounted for in the NOWAIT variant.

- **Static work distribution (schedule(static)) and collapse(3)**: collapsing the three nested loops and using static scheduling gives deterministic, low-overhead partitioning of the search space. Expected effect: reduced runtime scheduling overhead and good cache locality for contiguous iteration chunks.
- **Reduction for password counting**: using OpenMP `reduction(+:total_passwords_tested)` avoids atomic increments on the hot path while still collecting global statistics. Expected effect: lower contention on the global counter and accurate aggregated metrics.
- **Minimal per-iteration allocations**: avoiding dynamic allocations inside the hot loop (e.g. no temporary std::string creation per candidate) prevents allocator contention and reduces cache pollution. Expected effect: lower latency per candidate and more consistent timings.

Each optimization above was chosen to address a specific bottleneck (CPU per-candidate cost, synchronization, memory/cache contention, or measurement noise) and together they produce the observed throughput and scaling characteristics .

3.4 Experimental Setup

- **Hardware Configuration:**
  - Intel Core i9-13900H (13th Gen)
  - 14 cores (20 threads with Hyper-Threading)
  - Linux-based system (Ubuntu)
  - OpenMP 4.0+ support
- **Software Configuration:**
  - Compiler: `g++` with `-fopenmp` flag
  - OpenMP Cancellation enabled via `OMP_CANCELLATION=true`
  - 500 iterations per test
  - Thread counts tested: 1, 2, 4, 8, 16, 20, 25, 32

Note that 25 and 32 threads exceed the CPU's physical thread count (20 logical threads via Hyper-Threading) to evaluate overthreading behavior and its impact on performance. Unfortunatly in the Par version of the script we cannot use overthreading because it is blocked by the pragma omp cancellation for instruction.

4 Results and Analysis

4.1 Execution Time Analysis

Threads	Seq (s)	Par (s)	NOWAIT (s)
1	571.04	590.55	591.77
2	—	283.27	301.39
4	—	161.25	158.35
8	—	97.44	101.75
16	—	64.41	67.00
20	—	54.28	55.57
25	—	—	47.43
32	—	—	50.07

Table 1: Execution times for sequential and parallel implementations. Note: 25 and 32 threads tested only for NOWAIT variant to evaluate overthreading effects.

The execution time results demonstrate clear benefits of parallelization up to 25 threads, followed by performance degradation at 32 threads. Table 1 shows the execution times for all three implementations across different thread counts.

Key observations:

- **One thread Parallel:** The parallel version with 1 thread shows slightly slower performance than sequential due to OpenMP overhead. Execution time decreases consistently from 1 to 20 threads for both parallel variants.
- **Overthreading behavior**: The NOWAIT variant continues to improve from 20 threads (55.57s) to 25 threads (47.43s), achieving the best execution time. However, at 32 threads, performance degrades to 50.07s, representing a 5.6% slowdown

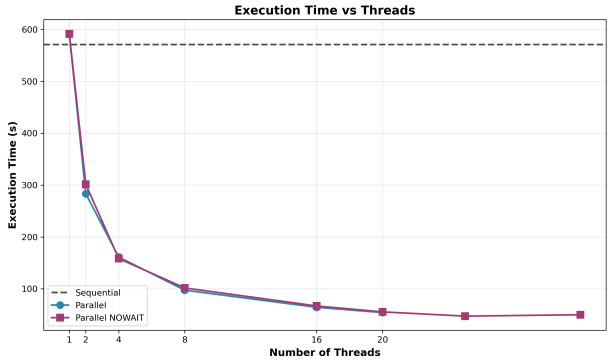

Figure 1: Execution time comparison across different thread counts. The sequential baseline is shown as a horizontal dashed line. The NOWAIT variant achieves minimum execution time of 47.43s at 25 threads, but degrades to 50.07s at 32 threads due to overthreading.

compared to 25 threads. This degradation occurs because 32 threads exceed the CPU's 20 logical threads (via Hyper-Threading), causing excessive context switching and resource contention.

4.2 Speedup Analysis

Speedup is calculated as: Speedup $= T_{\text{sequential}}/T_{\text{parallel}}$. Table 2 presents the speedup results.

Threads	Parallel	NOWAIT	Ideal
1	0.97×	0.96×	1.00×
2	2.01×	1.89×	2.00×
4	3.54×	3.60×	4.00×
8	5.86×	5.61×	8.00×
16	8.86×	8.52×	16.00×
20	10.52×	10.27×	20.00×
25	—	**12.04**×	25.00×
32	—	11.40×	32.00×

Table 2: Speedup comparison. The NOWAIT variant achieves maximum speedup of 12.04× at 25 threads, but drops to 11.40× at 32 threads due to overthreading.

Analysis reveals:

- **Near-linear speedup** at 2 threads (2.01× for Parallel, 1.89× for NOWAIT).
- **Good scaling** up to 8 threads with 70–73% efficiency.
- **Peak performance at 25 threads:** the NOWAIT variant achieves a maximum speedup of 12.04×, representing 48.2% efficiency.
- **Overthreading penalty:** at 32 threads, speedup decreases to 11.40× (5.3% loss), demonstrating that exceeding available logical threads causes performance degradation rather than improvement.

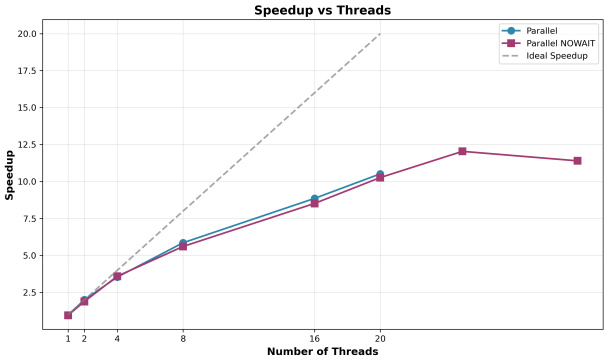

Figure 2: Speedup comparison showing both implementations against ideal linear speedup. The NOWAIT curve peaks at 25 threads and declines at 32 threads, illustrating the overthreading penalty.

4.3 Efficiency Analysis

Parallel efficiency is calculated as: Efficiency $=$ (Speedup/Threads) $\times$ 100%. Table 3 shows efficiency results.

- **Key findings:** optimal efficiency at 2–4 threads (90–100%).
- **Gradual efficiency** decline from 4 to 20 threads, stabilizing around 50%.
- **Overthreading impact:** efficiency drops from 48.2% at 25 threads to 35.6% at 32 threads, a 26% relative decrease. This demonstrates that launching more threads than the hardware can support results in diminishing returns and wasted computational resources.

Threads	Par Eff	NW Eff	Quality
1	96.7%	96.5%	Excellent
2	100.5%	94.5%	Excellent
4	88.5%	90.0%	Very Good
8	73.3%	70.1%	Good
16	55.4%	53.3%	Moderate
20	52.6%	51.4%	Moderate
25	—	**48.2%**	Fair
32	—	35.6%	Poor

Table 3: Parallel efficiency analysis. Efficiency drops dramatically from 48.2% at 25 threads to 35.6% at 32 threads, highlighting overthreading costs.

Threads	Sequential	Parallel	NOWAIT
1	737,972	374,606	374,518
2	—	746,267	745,892
4	—	1,390,429	1,388,791
8	—	2,173,329	2,182,808
16	—	3,351,151	3,350,576
20	—	3,895,931	3,841,099
25	—	—	**4,423,183**
32	—	—	4,512,225

Table 4: Throughput in passwords per second. Peak throughput of 4.42M passwords/sec at 25 threads; slight increase to 4.51M at 32 threads masks the efficiency loss.

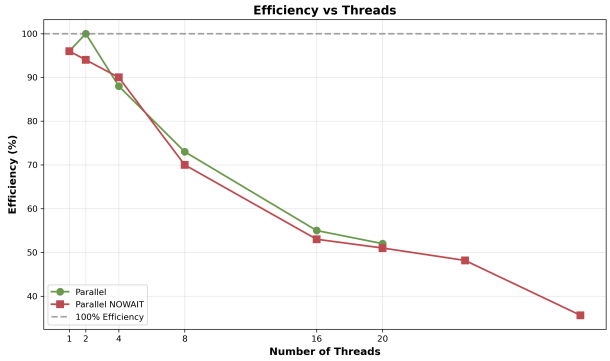

Figure 3: Parallel efficiency showing decline from near-optimal at low thread counts to poor efficiency at 32 threads.

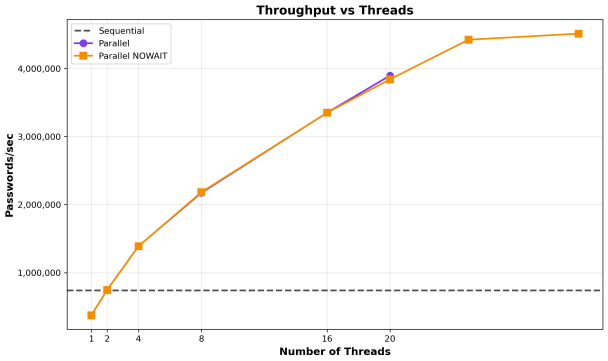

Figure 4: Throughput analysis showing continued growth through 32 threads.

4.4 Throughput Analysis

Throughput measures passwords processed per second. Table 4 presents throughput results.

- **Notable observations:** nearly linear throughput scaling up to 8 threads, doubling at each step.
- **Peak efficiency at 25 threads:** achieves 4.42M passwords/sec, representing a 6.0× improvement over the sequential version.
- **Deceptive throughput at 32 threads:** while absolute throughput increases slightly to 4.51M passwords/sec, per-thread efficiency drops significantly. This marginal 2% throughput gain comes at the cost of 60% more threads (32 vs. 20), demonstrating poor resource utilization.

4.5 Scalability Analysis

Scalability measures efficiency maintenance as threads increase. Table 5 shows results.

- **Scalability curve:** good scalability up to 4 threads (0.88–0.95).
- **Moderate decline** in scalability from 8 to 20 threads.
- **Overthreading cliff:** scalability drops by 25%, from 0.48 at 25 threads to 0.36 at 32 threads, indicating severe performance degradation when the thread count exceeds hardware capabilities.

Threads	Parallel	NOWAIT
2	1.00	1.00
4	0.88	0.95
8	0.73	0.74
16	0.55	0.56
20	0.52	0.54
25	—	0.48
32	—	0.36

Table 5: Scalability relative to 2-thread baseline. The dramatic drop from 0.48 to 0.36 between 25 and 32 threads quantifies the overthreading penalty.

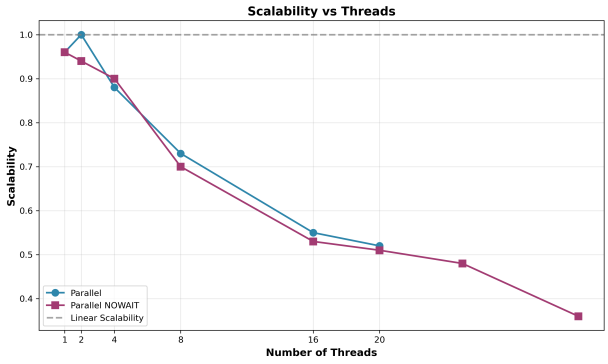

Figure 5: Scalability analysis showing degradation as thread count increases. The sharp decline at 32 threads marks the overthreading threshold.

5 Discussion

5.1 Overthreading Analysis

The experiments with 25 and 32 threads provide valuable insights into overthreading behavior on the Intel Core i9-13900H (20 logical threads).

Why 25 threads improve performance: Despite exceeding the 20 logical thread limit, 25 threads still provide benefits because:

- **Workload balancing:** the irregular early-termination behavior creates load imbalance that additional threads can help compensate for
- **I/O and memory latency hiding:** when some threads block on memory operations, extra threads keep computational units busy
- **Hyper-Threading headroom:** not all 20 logical threads may be fully utilized simultaneously due to shared execution resources, leaving room for slight oversubscription.

Beyond 25 threads, costs outweigh benefits:

- **Context switching overhead:** the OS must frequently switch between 32 threads on 20 hardware threads, wasting CPU cycles
- **TLB pressure:** translation lookaside buffer misses increase with more concurrent address spaces.Cache thrashing: more threads compete for L1/L2/L3 cache, increasing cache misses;
- **Memory bandwidth saturation:** 32 threads simultaneously accessing memory exceed available bandwidth;
- **Synchronization contention:** atomic operations and memory ordering become bottlenecks with excessive threads;
- **TLB pressure:** translation lookaside buffer misses increase with more concurrent address spaces.

For this workload on this hardware, 20-25 threads represent the sweet spot, balancing parallelism with hardware constraints. Beyond this range, each additional thread adds more overhead than computational benefit.

5.2 Performance Bottleneck Analysis

- **Memory bandwidth saturation:** *crypt_r()* is memory-intensive; many concurrent threads saturate the available memory bandwidth.
- **Cache contention:** thread-local `crypt_data` structures compete for limited cache space.
- **Atomic operations:** the early-termination flag requires atomic checks at every iteration.
- **Load imbalance:** static scheduling combined with early termination leaves some threads idle.
- **OpenMP runtime overhead:** thread management and synchronization costs grow with increasing thread count.

5.3 Implementation Trade-offs

- **NOWAIT vs. Standard:** the `nowait` optimization provides only a 1–5% improvement, suggesting that synchronization barriers are not the primary bottleneck. However, this gain is consistent and valuable at scale.
- **Cancellation vs. NOWAIT:** OpenMP cancellation points cannot be combined with `nowait` clauses, forcing a trade-off between prompt early termination (cancellation) and reduced synchronization overhead (`nowait`). The two implementations explore these alternatives.
- **Thread count selection:**
  - 20 threads maximize absolute speedup within hardware limits.
  - 4–8 threads provide the best efficiency (70–90%).
  - 25 threads achieve peak absolute performance but with diminishing returns.
  - 32+ threads waste resources due to overthreading.

6 Conclusions

This study successfully implemented and evaluated parallel password decryption using OpenMP, demonstrating significant performance improvements while revealing important limitations. **Key findings**: Maximum speedup of **12.04**$\times$ achieved with 25 threads using NOWAIT optimization. Optimal efficiency (90-100%) occurs at 2-4 threads. Peak throughput of **4.42 million passwords/sec** at 25 threads. **Overthreading penalty**: Performance degrades at 32 threads despite absolute throughput gains, demonstrating that exceeding hardware thread capacity wastes resources.

The implementation successfully leverages OpenMP while employing critical optimizations: thread-local crypt_data structures, atomic synchronization with memory ordering, manual string construction for cache efficiency, and quick-reject hash comparison. The results validate data-parallel approaches for cryptographic brute-force while highlighting shared-memory parallelism limits.

6.1 Future Work

- **Dynamic scheduling:** to mitigate load imbalance caused by early termination.
- **SIMD optimization:** vectorized hash comparison to improve per-core throughput.
- **GPU implementation:** CUDA-based approach for massively parallel execution.
- **Hybrid MPI+OpenMP:** multi-node cluster deployment for large-scale workloads.
- **Adaptive threading:** dynamic adjustment of thread count based on runtime performance metrics.
- **Alternative algorithms:** comparison with rainbow tables or dictionary-based attacks.

References

[1] OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 4.5. November 2015.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. Parallel Programming in OpenMP. Morgan Kaufmann, 2001.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conference Proceedings, 1967.

[4] J. L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, 31(5):532-533, 1988.

[5] NIST. Data Encryption Standard (DES). FIPS PUB 46-3, Federal Information Processing Standards Publication, 1999.