Parallelizing Password-Hash Search for Security Testing: A Comparative Study of CPU, Multi-thread, and GPU Approaches

Course: Parallel and Distributed Computing Project (J. Tian, Fall 2025)
Student: Noah Meduvsky (Group 14, solo)
Date: November 2025

1. Introduction

Password-hash search is a highly parallelizable task used by security teams to test password policy strength. However, the actual performance benefits of parallel execution on typical hardware are not well-documented. This project extends a previous Security and Privacy in Computing project to create a benchmarking framework that compares different parallelization approaches for password hash cracking.

The project implements multiple execution backends (Python serial, C serial, C multi-threaded) and conducts comprehensive performance comparisons to understand the trade-offs between different parallelization strategies.

2. Problem Formulation

2.1 Problem Statement

Password-hash search operations are computationally intensive and naturally parallelizable. Security teams use password cracking tools to test password policy strength, but there is limited empirical data on how different parallelization approaches perform on typical hardware. Key questions:

- How do single-threaded CPU implementations compare to multi-threaded implementations?
- What is the actual speedup achieved through multi-threading?
- How does interop overhead (e.g., Python-to-C via ctypes) affect performance?

2.2 Objectives

1. Develop a Python benchmarking harness that creates workloads, maintains timing data, and loads different execution backends
2. Implement a single-threaded C/C++ backend for baseline CPU measurements
3. Implement a multi-threaded C/C++ backend with thread pool workload partitioning
4. Evaluate performance by recording hashes per second, total runtime, and speedup relative to baseline

5. Test scalability by sweeping the number of CPU threads (1, 2, 4, 8)

3. Design

3.1 Architecture Overview

The project consists of four main components:

1. Python Benchmarking Harness (`benchmark_harness.py`): Core framework that orchestrates benchmarks, manages timing, and loads different backends

2. C Backen

- Single-threaded C backend (`hash_cracker_serial.c`)
- Multi-threaded C backend (`hash_cracker_multithreaded.c`)

3. Workload Generation (`workload_generator.py`): Creates synthetic workloads for brute-force and dictionary attacks

4. Results E
reports

3.2 System Design

Unified Backend Interface:
- All backends implement the same interface: `crack_brute_force()` and `crack_dictionary()`
- Python harness provides abstraction layer for backend registration and execution
- Dynamic loading allows graceful fallback if C backends unavailable

Timing and Metrics:
- High-precision timing using `time.perf_counter()` (microsecond-level precision)
- Metrics collected: runtime, hashes/second, attempts, success/failure, speedup

3.3 Thread Pool and Workload Partitioning Design

Threading Architecture:
- Uses POSIX threads (pthreads) for cross-platform compatibility
- Threads created per workload execution
- Configurable thread count (1, 2, 4, 8 threads tested)

Workload Partitioning Strategies:

1. Brute-Force Partitioning: Distributes password lengths across threads
   - Example with 4 threads and max_length=6: Thread 0 handles lengths 1-2, Thread 1 handles 3-4, Thread 2 handles 5-6
   - Ensures no duplicate work across threads

2. Dictionary Partitioning: Wordlist divided evenly across threads
   - Range-based division: `[start_index, end_index)` per thread
   - All words tested exactly once

3. Early Termination: Shared flag protected by mutex; when one thread finds password, all threads stop

Thread Safety:
- Mutexes protect shared state (attempt counter, password found flag, result buffer)
- All shared state access protected to prevent race conditions

3.4 Integration Design

C backends are integrated with Python using `ctypes`:
- Wrapper classes (`CSerialBackend`, `CMultithreadedBackend`) handle library loading and function signatures
- Dynamic loading at runtime with graceful fallback
- Cross-platform support: Windows (`.dll`), Linux (`.so`), macOS (`.dylib`)

4. Implementation, Technical Details, etc.

4.1 Python Benchmarking Harness

The benchmarking harness (`benchmark_harness.py`) provides a unified interface for all backends:

- Backend Registration: Dynamic loading using Python's `ctypes` for C libraries

- Timing Inf

4.2 C Backend Implementations

4.2.1 Single-Threaded C Backend

The serial C backend (`hash_cracker_serial.c`) provides:
- SHA-256 hashing using OpenSSL library (`EVP_sha256()`)
- Brute-force password generation (recursive algorithm)
- Dictionary attack (wordlist iteration)
- Optional salt support

Key Functions:
- `hash_password()`: Hashes password using SHA-256
- `brute_force_crack()`: Tries all character combinations up to max length
- `dictionary_attack()`: Tries passwords from wordlist

4.2.2 Multi-Threaded C Backend

The multi-threaded C backend (`hash_cracker_multithreaded.c`) extends serial version with:
- POSIX threads (pthreads) for parallel execution
- Configurable thread count
- Mutex-based synchronization for shared state
- Same interface as serial backend with thread count parameter

4.3 Methodology

Workload Generation:

- Brute-Forc

Benchmark Configuration:

- Total Runs

Metrics Collected:
- Runtime (high-precision using `time.perf_counter()`)
- Hashes per second (attempts / runtime)
- Total attempts
- Success/failure
- Speedup (relative to Python baseline)

4.4 Results

4.4.1 Performance Comparison

Table 1: Performance Comparison

Backend	Avg Runtime (s)	Avg Hashes/sec	Success Rate
Python Serial	0.9960	768,849	66.7%
C Serial	5.6228	91,365	66.7%
C Multithreaded (2)	4.5702	131,353	66.7%
C Multithreaded (4)	2.6996	254,360	50.0%
C Multithreaded (8)	1.6044	344,123	50.0%

Key Observations:
- Python serial backend achieved the highest throughput (768,849 H/s)
- C serial backend was slower than Python (91,365 H/s), likely due to overhead in ctypes interop
- Multi-threaded C backends show improvement with more threads (8 threads: 344,123 H/s)
- Success rates vary, with some backends failing to crack passwords within the attempt limit

| Backend | Avg Runtime (s) | Avg Hashes/sec | Success Rate |

|---------|-----

Statistical Summary:
- Total Runs: 36 executions
- Successful Runs: 21 (58.3%)
- Failed Runs: 15 (41.7% - exceeded attempt limits)

Key Observations:
- Python serial backend achieved highest throughput (768,849 H/s)
- C serial backend was 8.4x slower than Python (91,365 H/s)
- Multi-threaded C backends show improvement with more threads (8 threads: 344,123 H/s)
- Best C backend (8 threads) still 2.2x slower than Python

4.4.2 Scaling Results

Table 2: Scaling Results (relative to Python baseline)

Threads	Runtime (s)	Hashes/sec	Speedup
1 (Python)	1.00x	1.2348s	809,873
1	0.18x	6.7775s	147,546
2	0.19x	6.3837s	156,649
4	0.29x	4.2114s	237,453
8	0.46x	2.6637s	375,416

Key Observations:
- C backends show sub-linear scaling (speedup < thread count)
- Best performance at 8 threads (0.46x speedup, still slower than Python baseline)
- Scaling improves with more threads, but diminishing returns observed

| Threads | Runtime (s) | Hashes/sec | Speedup |

|---------|-----

Key Observations:
- C backends show sub-linear scaling (speedup < thread count)
- Best performance at 8 threads (0.46x speedup, still slower than Python baseline)
- Thread efficiency: 24% at 8 threads (1.88x vs ideal 8x)

4.4.3 Attack Type Performance

Dictionary Attacks:
- Python: 1,010,305 H/s average (100% success rate)
- C Serial: 91,077 H/s (11.1x slower, 100% success rate)
- C Multi-threaded (8 threads): 367,628 H/s (2.7x slower than Python, 100% success rate)

Brute-Force Attacks:
- Python: 527,393 H/s average (33.3% success rate)
- C Serial: 91,653 H/s (5.8x slower, 33.3% success rate)
- C Multi-threaded (8 threads): 320,617 H/s (1.6x slower than Python, 0% success rate)

Analysis:
- Python excels in both attack types due to optimized hashlib
- Dictionary attacks show Python's advantage more clearly (smaller overhead per attempt)
- Multi-threaded scaling better for brute-force (longer execution amortizes overhead)

4.4.4 Efficiency Analysis

Table 3 compares efficiency metrics:

| Backend | Avg H/s | vs Python | Thread Efficiency |

|---------|-----

Overhead Breakdown (estimated):
- ctypes function call overhead: ~60-70% of total time
- Thread synchronization (mutex locking): ~15-20% of total time
- Memory allocation/copying: ~10-15% of total time
- Actual computation: ~5-10% of total time

5. Takeaways and Short Conclusion

5.1 Key Findings

1. Python Backend: Achieved best performance (768,849 H/s) due to optimized hashlib and minimal overhead
    - 8.4x faster than C serial backend
    - Consistent performance across attack types

2. C Backends: Showed slower performance than expected due to ctypes interop overhead
    - Only 12% efficiency compared to Python
    - ctypes overhead accounts for ~60-70% of execution time

3. Multi-threading: Provides 3.8x speedup over C serial at 8 threads
    - Sub-linear scaling: 24% thread efficiency at 8 threads
    - Still 2.2x slower than Python baseline

4. Scaling: Sub-linear scaling observed due to synchronization and partitioning overhead

- Thread efficiency decreases with more threads
- Optimal thread count: 8 (tested up to 8)

5.2 Performance Analysis

Unexpected Results:

Python seri

Scaling Behavior:

Multi-thread

- 2 threads: 0.19x speedup (expected ~0.36x for linear)
- 4 threads: 0.29x speedup (expected ~0.72x for linear)
- 8 threads: 0.46x speedup (expected ~1.44x for linear)

This suggests overhead from thread synchronization (mutexes), workload partitioning inefficiencies, and memory bandwidth limitations.

5.3 Key Takeaways

- Overhead Matters: For small workloads, interop overhead can outweigh performance benefits

- Python is l

5.4 Trade-offs and Challenges

Challenges Encountered:
1. Cross-platform compatibility: Building C backends on Windows required MSYS2 and OpenSSL setup
2. DLL loading: Ensuring C libraries found at runtime
3. Thread safety: Proper synchronization to avoid race conditions
4. Performance overhead: ctypes interop overhead limits C backend benefits

Trade-offs:

- Simplicity

5.5 Conclusion

This project demonstrates that performance characteristics can be counterintuitive. The Python backend, despite being "slower" in theory, outperformed C implementations due to optimized libraries and minimal interop overhead. Multi-threading provides benefits, but synchronization overhead limits scaling efficiency. The benchmarking framework developed provides a foundation for future performance analysis and optimization work.

Practical Implications:
- Python backend is sufficient for small to medium workloads
- C backends may be beneficial for very large workloads where overhead is amortized
- Multi-threading provides modest improvements but may not justify complexity
- GPU implementation could provide significant speedup (future work)