Michael Ott

Virginia Commonwealth university

Parallel Hash-Based Password Cracking Using OpenMP and MPI

Abstract

Knapcrack is a CPU-optimized tool designed to crack MD5 password hashes efficiently using parallel algorithms. Unlike most existing solutions that rely on GPU acceleration, Knapcrack leverages OpenMP for multi-threaded execution, MPI for distributed memory parallelism, and a novel bin-packing heuristic to partition tasks effectively across CPU resources. This approach maximizes computational efficiency and balances workloads to achieve competitive performance without specialized hardware. By focusing on CPU-based optimization, Knapcrack bridges the gap between GPU-accelerated tools and traditional CPU solutions. Its design demonstrates the potential of parallel algorithms in tackling real-world cybersecurity challenges, providing a scalable framework for tasks such as penetration testing and hash recovery.

Introduction

Hashing is a fundamental concept in cybersecurity, widely used to securely store sensitive information such as passwords. Cryptographic hash functions, like MD5, transform input data into unique, fixed-length outputs that are computationally infeasible to reverse. Despite its historical popularity, MD5 has become a target for attackers due to its vulnerabilities, making efficient hash-cracking tools essential for ethical penetration testing and cybersecurity research.

Most existing tools for cracking hashes leverage GPU acceleration to achieve high performance. While effective, these solutions often require specialized hardware, which can be inaccessible or impractical in certain scenarios. This creates a need for CPU-based tools that balance accessibility with computational efficiency. Addressing this gap is critical for environments where GPUs are unavailable or cost prohibitive.

Knapcrack is designed to meet this need by offering an efficient solution for cracking MD5 hashes on CPU systems. It employs OpenMP for multi-threaded execution, MPI for distributed memory parallelism, and a novel bin-packing heuristic to partition tasks effectively across available resources. By optimizing parallel algorithms for CPU architecture, Knapcrack achieves competitive performance without relying on GPUs, showcasing the potential of innovative methodologies in cybersecurity tasks.

This report outlines the design, implementation, and performance of Knapcrack, emphasizing its practical applications and contributions to the field of cybersecurity. The following sections detail its technical approach, challenges faced, and results achieved, providing insights into the development of efficient, CPU-based solutions for hash-cracking and similar computational challenges.

Related Works

The field of password hash-cracking has been significantly advanced by tools like *John the Ripper* and *Hashcat*, which are widely regarded as industry standards. These tools have set a high benchmark for efficiency and functionality, leveraging advanced techniques to address the computational challenges of cracking cryptographic hashes.

John the Ripper is a versatile and highly configurable password-cracking tool that supports various cryptographic hash types, including MD5. Its primary strength lies in its simplicity and flexibility, allowing users to employ rule-based attacks, wordlists, and custom scripts. While *John the Ripper* supports multi-threading and some level of parallel execution, its performance on CPU systems can be limited when compared to GPU-accelerated tools.

Hashcat has further expanded the possibilities of password cracking by prioritizing GPU acceleration for enhanced performance. Its robust framework enables a wide range of attacks, including brute-force, mask, and dictionary-based methods. With support for modern GPU architectures, *Hashcat* is capable of processing massive workloads in parallel, making it the go-to solution for high-speed hash-cracking tasks. However, its reliance on GPUs can be a drawback in environments where such hardware is unavailable or impractical.

Both tools highlight the trade-off between performance and hardware dependency in hash-cracking solutions. While GPU acceleration offers superior speed, it requires specialized equipment that may not always be accessible. This limitation underscores the need for innovative CPU-based alternatives, such as Knapcrack, which bridges the gap by optimizing parallel algorithms for general-purpose processors. Unlike *John the Ripper* and *Hashcat*, Knapcrack focuses solely on CPU efficiency, leveraging OpenMP, MPI, and task partitioning via a bin-packing heuristic to achieve competitive performance.

By building upon the foundational principles established by *John the Ripper* and *Hashcat*, Knapcrack aims to offer a practical and accessible alternative, further enriching the field of hash-cracking methodologies.

Methods

This section details the approach employed by Knapcrack to efficiently crack MD5 hashes using CPU-based parallelization techniques. The methodology integrates distributed-memory parallelism through MPI, shared-memory parallelism through OpenMP, and a bin-packing heuristic for workload distribution, aiming to balance computational effort across multiple processes and threads.

Parallelization Framework
Knapcrack leverages MPI (Message Passing Interface) to distribute workloads among multiple compute nodes or processes. By dividing the task into separate partitions, each MPI rank handles a portion of the input data. Within each process, OpenMP is employed to fully utilize the available CPU cores by threading the MD5 hash computation and comparison tasks. This dual-layer parallelization model ensures that both inter-node (MPI) and intra-node (OpenMP) concurrency is exploited, enhancing scalability and execution speed without relying on GPU hardware.

Hash Loading and Distribution
At the start of the execution, the root process (rank 0) reads a set of target MD5 hashes from a file (e.g., "hashes.txt"). These hashes are stored in a dynamically allocated array. To provide all ranks with the same search targets, the root process broadcasts both the count and the actual hash data. Non-root ranks reconstruct identical hash arrays locally, ensuring a consistent global view of the target hashes to be cracked.

Initial Consideration of Knapsack vs. Bin-Packing Approaches
Initially, Knapcrack considered using a knapsack-based optimization strategy to partition the workload. The knapsack approach was aimed at finding an optimal distribution of dictionary entries (treated as items with weights) to maximize load balance. However, this approach proved overly complex and computationally expensive for large inputs. After practical evaluation, it became evident that a simpler yet effective strategy was necessary.

Dictionary Weighting and Bin-Packing Heuristic
Instead of a full-fledged knapsack method, Knapcrack employs a more streamlined bin-packing heuristic. The dictionary lines from "rockyou.txt" are first weighted by their length, with longer lines considered “heavier.” These entries are then sorted in descending order of weight. Using a first-fit decreasing bin-packing approach, the heaviest entries are placed first among the ranks, followed by lighter entries. This heuristic balances the workload effectively and is more manageable than the knapsack approach, striking a practical compromise between optimality and computational feasibility.

Data Broadcasting and Local Dictionary Construction
After partition assignments are determined on the root process, these assignments—along with the global array of line weights—are broadcast to all ranks. Each rank identifies the

subset of dictionary lines it must process and reads only those lines directly from "rockyou.txt." By performing localized file reads, Knapcrack reduces unnecessary data transfers and ensures that each rank can quickly access the dictionary entries it has been assigned, further enhancing scalability.

MD5 Computation and Comparison

With their assigned subset of the dictionary loaded into memory, each rank's worker threads, managed by OpenMP, compute the MD5 hash of each candidate password. The computed hash is then compared against the global array of target MD5 hashes. By distributing these computations across multiple threads, Knapcrack accelerates the search for matches, efficiently utilizing the CPU cores available to each rank. When a match is found, it is recorded locally.

Result Aggregation and Performance Measurement

Following the completion of all assigned workloads, each rank reports its number of found matches. MPI's reduction operations then aggregate these results at the root process, providing a global total of matched hashes. Performance metrics, such as elapsed time, are measured using MPI's timing functions. Finally, all dynamically allocated memory is released and MPI is finalized to ensure clean termination.

Testing

The testing phase of Knapcrack involved evaluating the performance and accuracy of three implementations: a serial hash cracker (crack.c), an initial MPI-based parallel version with no partitioning (knapcrack0.c), and the final optimized solution (knapcrack.c). The testing methodology aimed to compare the computational efficiency and scalability of these implementations under various conditions.

Testing Environment

All testing was conducted on a powerful Kali Linux virtual machine configured with maximum CPU cores and memory allocation. The VM was hosted on an MSI GL65 Leopard laptop, ensuring consistent performance and access to adequate computational resources for all experiments.

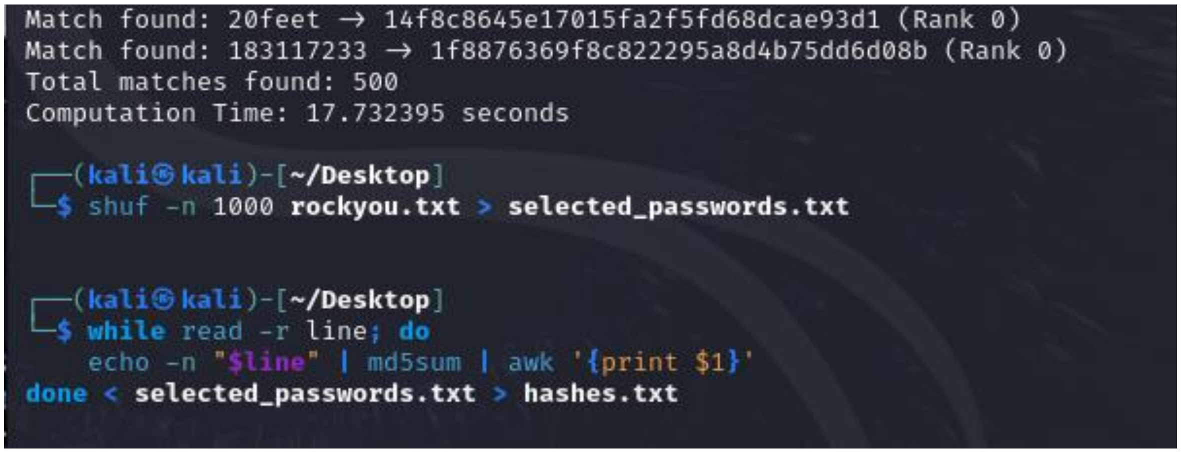
Data Preparation

To generate test data, the shuf command was used to randomly extract entries from the rockyou.txt dictionary. These entries were then converted into MD5 hashes using a custom script and stored in a file named hashes.txt. Multiple datasets of varying sizes were created to simulate different workloads, containing 10, 100, 250, 500, 1000, 2500, 5000, and 10000 hashes, respectively.

Testing Process

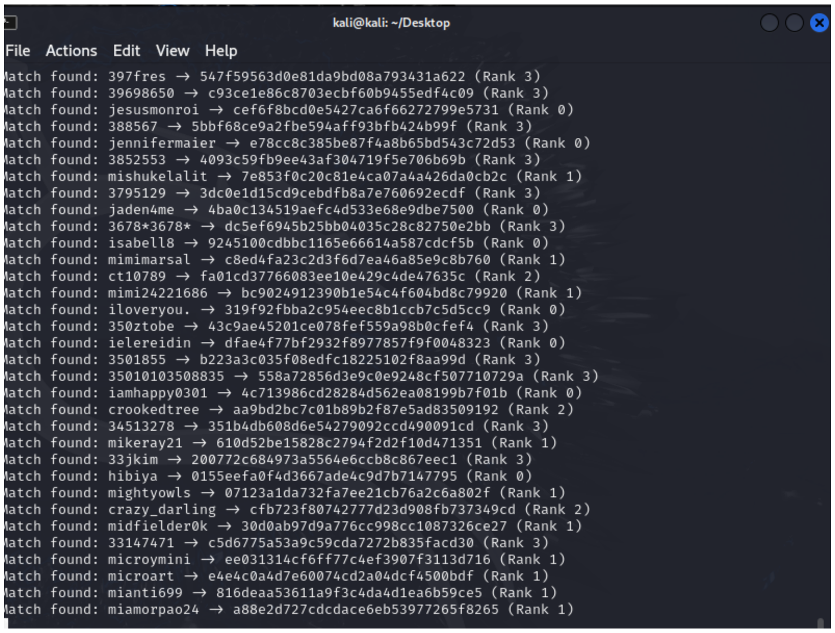
1. **Serial Testing**:
   The crack.c implementation was used as a baseline to measure the performance of the serial approach. This implementation processed the generated hashes.txt files sequentially, providing a reference for evaluating the benefits of parallelization.
2. **Basic MPI Testing**:
   The knapcrack0.c implementation utilized MPI for basic parallelization but did not incorporate any task partitioning. This version was run on 4 and 6 processors to assess the impact of simple parallel execution without load balancing.
3. **Optimized Testing**:
   The final version, knapcrack.c, incorporated the bin-packing heuristic for workload partitioning alongside MPI and OpenMP. It was also tested on 4 and 6 processors for all dataset sizes to evaluate scalability and efficiency.

This is a screengrab example of how hashes.txt was prepared for each iteration of testing.



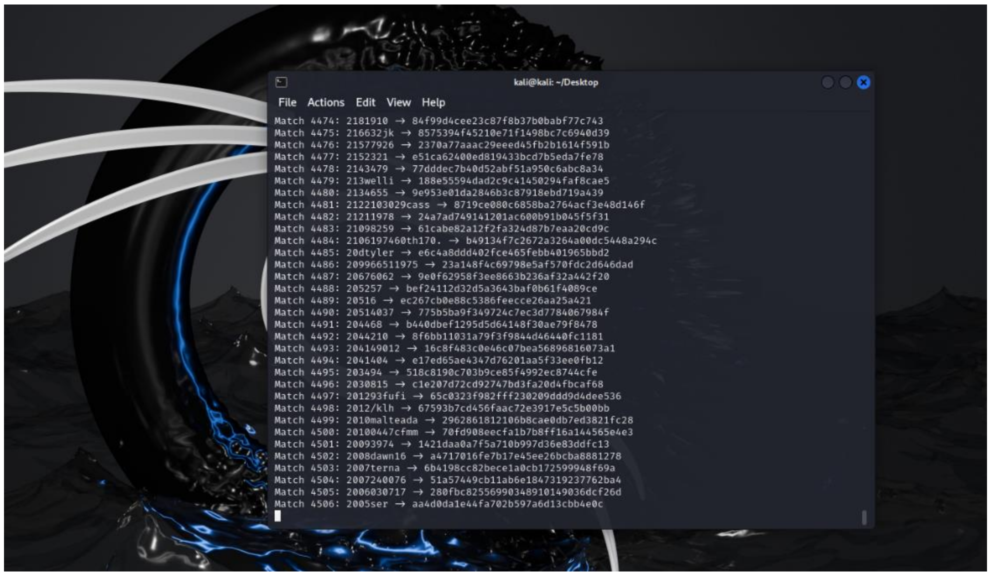
```
Match found: 20feet → 14f8c8645e17015fa2f5fd68dcae93d1 (Rank 0)
Match found: 183117233 → 1f8876369f8c822295a8d4b75dd6d08b (Rank 0)
Total matches found: 500
Computation Time: 17.732395 seconds

┌──(kali㉿kali)-[~/Desktop]
└─$ shuf -n 1000 rockyou.txt > selected_passwords.txt


┌──(kali㉿kali)-[~/Desktop]
└─$ while read -r line; do
    echo -n "$line" | md5sum | awk '{print $1}'
done < selected_passwords.txt > hashes.txt
```

Additional screengrabs during testing:



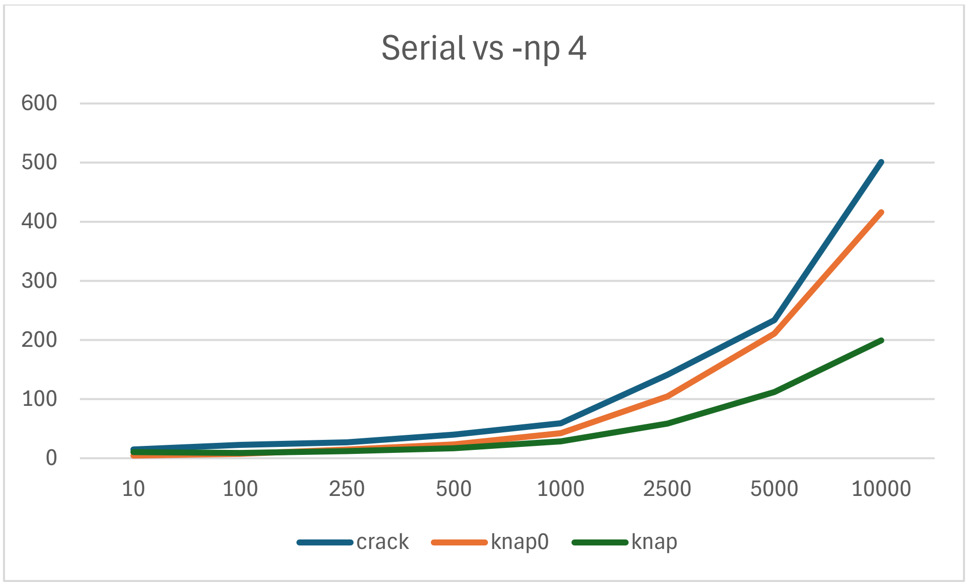
```
atch found: 397fres → 547f59563d0e81da9bd08a793431a622 (Rank 3)
atch found: 39698650 → c93ce1e86c8703ecbf60b9455edf4c09 (Rank 3)
atch found: jesusmonroi → cef6f8bcd0e5427ca6f66272799e5731 (Rank 0)
atch found: 388567 → 5bbf68ce9a2fbe594aff93bfb424b99f (Rank 3)
atch found: jennifermaier → e78cc8c385be87f4a8b65bd543c72d53 (Rank 0)
atch found: 3852553 → 4093c59fb9ee43af304719f5e706b69b (Rank 3)
atch found: mishukelalit → 7e853f0c20c81e4ca07a4a426da0cb2c (Rank 1)
atch found: 3795129 → 3dc0e1d15cd9cebdfb8a7e760692ecdf (Rank 3)
atch found: jaden4me → 4ba0c134519aefc4d533e68e9dbe7500 (Rank 0)
atch found: 3678*3678* → dc5ef6945b25bb04035c28c82750e2bb (Rank 3)
atch found: isabell8 → 9245100cdbbc1165e66614a587cdcf5b (Rank 0)
atch found: mimimarsal → c8ed4fa23c2d3f6d7ea46a85e9c8b760 (Rank 1)
atch found: ct10789 → fa01cd37766083ee10e429c4de47635c (Rank 2)
atch found: mimi24221686 → bc9024912390b1e54c4f604bd8c79920 (Rank 1)
atch found: iloveryou. → 319f92fbba2c954eec8b1ccb7c5d5cc9 (Rank 0)
atch found: 350ztobe → 43c9ae45201ce078fef559a98b0cfef4 (Rank 3)
atch found: ielereidin → dfae4f77bf2932f8977857f9f0048323 (Rank 0)
atch found: 3501855 → b223a3c035f08edfc18225102f8aa99d (Rank 3)
atch found: 35010103508835 → 558a72856d3e9c0e9248cf507710729a (Rank 3)
atch found: iamhappy0301 → 4c713986cd28284d562ea08199b7f01b (Rank 0)
atch found: crookedtree → aa9bd2bc7c01b89b2f87e5ad83509192 (Rank 2)
atch found: 34513278 → 351b4db608d6e54279092ccd490091cd (Rank 3)
atch found: mikeray21 → 610d52be15828c2794f2d2f10d471351 (Rank 1)
atch found: 33jkim → 200772c684973a5564e6ccb8c867eec1 (Rank 3)
atch found: hibiya → 0155eefa0f4d3667ade4c9d7b7147795 (Rank 0)
atch found: mightyowls → 07123a1da732fa7ee21cb76a2c6a802f (Rank 1)
atch found: crazy_darling → cfb723f80742777d23d908fb737349cd (Rank 2)
atch found: midfielder0k → 30d0ab97d9a776cc998cc1087326ce27 (Rank 1)
atch found: 33147471 → c5d6775a53a9c59cda7272b835facd30 (Rank 3)
atch found: microymini → ee031314cf6ff77c4ef3907f3113d716 (Rank 1)
atch found: microart → e4e4c0a4d7e60074cd2a04dcf4500bdf (Rank 1)
atch found: mianti699 → 816deaa53611a9f3c4da4d1ea6b59ce5 (Rank 1)
atch found: miamorpao24 → a88e2d727cdcdace6eb53977265f8265 (Rank 1)
```

Serial implementation handling large hash count

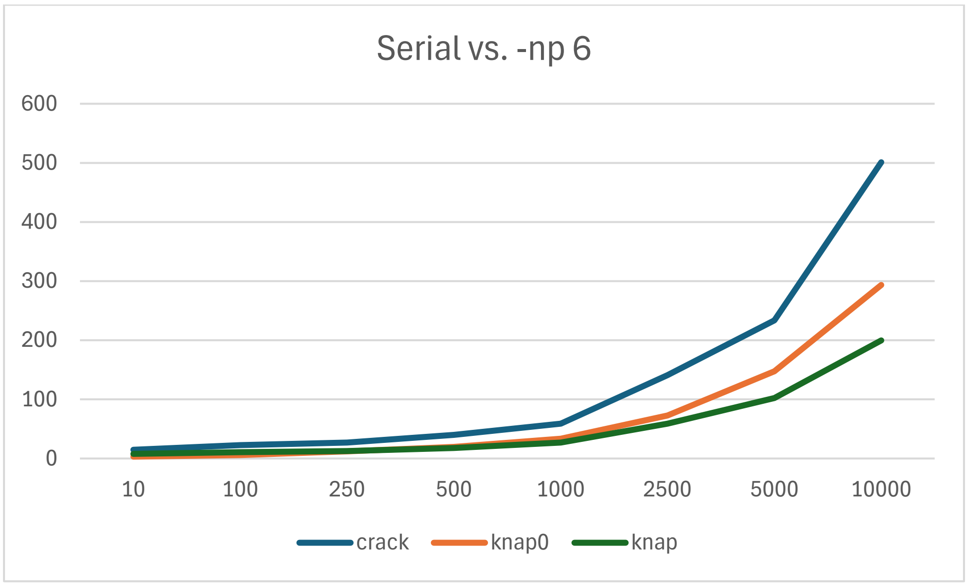


```
Match 4474: 2181910 → 84f99d4cee23c87f8b37b0babf77c743
Match 4475: 216632jk → 8575394f45210e71f1498bc7c6940d39
Match 4476: 21577926 → 2370a77aaac29eeed45fb2b1614f591b
Match 4477: 2152321 → e51ca62400ed819433bcd7b5eda7fe78
Match 4478: 2143479 → 77dddec7b40d52abf51a950c6abc8a34
Match 4479: 213welli → 188e55594dad2c9c41450294faf8cae5
Match 4480: 2134655 → 9e953e01da2846b3c87918ebd719a439
Match 4481: 2122103029cass → 8719ce080c6858ba2764acf3e48d146f
Match 4482: 21211978 → 24a7ad749141201ac600b91b045f5f31
Match 4483: 21098259 → 61cabe82a12f2fa324d87b7eaa20cd9c
Match 4484: 2106197460th170. → b49134f7c2672a3264a00dc5448a294c
Match 4485: 20dtyler → e6c4a8ddd402fce465febb401965bbd2
Match 4486: 209966511975 → 23a148f4c69798e5af570fdc2d646dad
Match 4487: 20676062 → 9e0f62958f3ee8663b236af32a442f20
Match 4488: 205257 → bef24112d32d5a3643baf0b61f4089ce
Match 4489: 20516 → ec267cb0e88c5386feecce26aa25a421
Match 4490: 20514037 → 775b5ba9f349724c7ec3d7784067984f
Match 4491: 204468 → b440dbef1295d5d64148f30ae79f8478
Match 4492: 2044210 → 8f6bb11031a79f3f9844d46440fc1181
Match 4493: 204149012 → 16c8f483c0e46c07bea56896816073a1
Match 4494: 2041404 → e17ed65ae4347d76201aa5f33ee0fb12
Match 4495: 203494 → 518c8190c703b9ce85f4992ec8744cfe
Match 4496: 2030815 → c1e207d72cd92747bd3fa20d4fbcaf68
Match 4497: 201293fufi → 65c0323f982fff230209ddd9d4dee536
Match 4498: 2012/klh → 67593b7cd456faac72e3917e5c5b00bb
Match 4499: 2010malteada → 2962861812106b8cae0db7ed3821fc28
Match 4500: 20100447cfmm → 70fd908eecfa1b7b8ff16a144565e4e3
Match 4501: 20093974 → 1421daa0a7f5a710b997d36e83ddfc13
Match 4502: 2008dawn16 → a4717016fe7b17e45ee26bcba8881278
Match 4503: 2007terna → 6b4198cc82bece1a0cb172599948f69a
Match 4504: 2007240076 → 51a57449cb11ab6e1847319237762ba4
Match 4505: 2006030717 → 280fbc82556990348910149036dcf26d
Match 4506: 2005ser → aa4d0da1e44fa702b597a6d13cbb4e0c
```

Results



As a reminder, crack (blue) is the serial implementation, knap0 (orange) is the basic MPI without partitioning, and knap (green) is the solution. Knap clearly scales better than other 2 implementations with the basic MPI struggling to even differentiate itself from serial at 4 processor count. The vertical axis is time in seconds. These were long tests consisting of several minutes.



The basic MPI had the most noticeable increase in performance using 6 processors while knap held relatively steady.

4 processor results

	10	100	250	500	1000	2500	5000	10000
crack	14.83	22.58	26.97	40.07	59.07	141.1	233.6	501.05
knap0	4.44	7.37	14.72	23.29	42.41	104.55	210.91	416.08
knap	10.39	8.92	12.17	16.82	28.5	58.69	111.93	199.26

6 processor results

	10	100	250	500	1000	2500	5000	10000
crack	14.83	22.58	26.97	40.07	59.07	141.1	233.6	501.05
knap0	3.15	5.95	12.04	19.68	33.54	72.75	147.53	293.51
knap	7.88	10.62	12.77	17.73	27.06	58.88	102.35	199.76

Conclusion

The development and testing of Knapcrack have demonstrated the significant impact of thoughtful workload partitioning and parallel algorithms on the efficiency of CPU-based hash-cracking tasks. By incorporating a bin-packing heuristic into the final implementation, Knapcrack achieves superior performance compared to its predecessor, Knapcrack0, even when operating with fewer processors. Notably, Knapcrack with 4 processors outperformed Knapcrack0 with 6 processors, showcasing the critical role of balanced task distribution in parallel computing.

These findings highlight that simply increasing the number of processors does not guarantee better performance. Without effective workload management, computational resources can be underutilized, leading to suboptimal results. The integration of MPI and OpenMP in Knapcrack, combined with the bin-packing heuristic, ensures that tasks are evenly distributed, minimizing idle time and enhancing overall execution speed.

The testing results also underscore the potential of CPU-based solutions in hash-cracking applications. While GPU-accelerated tools dominate the field, Knapcrack demonstrates that optimized CPU implementations can provide competitive performance, offering a more accessible alternative for environments lacking specialized hardware.

Knapcrack's success serves as a foundation for future research and development in parallel algorithms for cybersecurity. Potential extensions could include exploring dynamic partitioning techniques to adapt to varying workloads or extending support to other cryptographic hash functions. By building on the principles demonstrated here, further advancements can continue to bridge the gap between accessibility and performance in critical cybersecurity tasks.

In conclusion, Knapcrack is not just an efficient tool for cracking MD5 hashes; it is a proof of concept that emphasizes the importance of combining theoretical principles with practical implementations to address real-world challenges. The results presented in this report validate the effectiveness of its design and pave the way for continued exploration in the intersection of parallel computing and cybersecurity.

References

Tools

Openwall Project. John the Ripper Password Cracker. Openwall, www.openwall.com/john/. Accessed 2 Dec. 2024.

"Hashcat." Hashcat - Advanced Password Recovery, hashcat.net/. Accessed 2 Dec. 2024.

Websites and Forums

OWASP Foundation. "Password Storage Cheat Sheet." OWASP: Open Worldwide Application Security Project, owasp.org/www-project-cheat-sheets/password-storage-cheat-sheet. Accessed 2 Dec. 2024.

Cybersecurity & Infrastructure Security Agency (CISA). "Cyber Essentials Starter Kit." CISA, www.cisa.gov/cyber-essentials. Accessed 2 Dec. 2024.