Benchmarking

Unoptimized	1 Thread	2 Threads	4 Threads	8 Threads
Run 1	2.47	1.36	0.83	0.57
Run2	2.47	1.4	0.83	0.56
Run 3	2.47	1.36	0.83	0.56
Average Time	2.47	1.373333	0.83	0.563333

Optimized	1 Thread	2 Threads	4 Threads	8 Threads
Run 1	1.88	1.03	0.62	0.44
Run2	1.85	1.03	0.62	0.43
Run 3	1.86	1.03	0.61	0.43
Average Time	1.863333	1.03	0.616667	0.433333

	2 Threads	4 Threads	8 Threads
tocrack.txt (Optimized)	0.85	1.26	1.42
tocrack.txt (Unoptimized)	1.48	1.9	2.2
org.txt (Optimized)	0.82	1.25	1.43
org.txt (Unoptimized)	1.46	1.92	2.25

Analysis

In devising my parallel solution, I adopted a strategy centered on the efficient division of the common password file among threads, where each thread processed a specific segment. This involved allocating individual dictionaries to each thread optimizing the overall performance. The program reads the cracking file in a single thread and then checks the hash against all the thread dictionaries. Noteworthy changes from my Part I solution included a refined approach to reading the common password file and an enhanced dictionary construction process, aligning with the individualized dictionaries for each thread. These adjustments are aimed at streamlining the parallel execution and bolstering the program's overall efficiency.

A password salt is a random value that is combined with a password before it is hashed. This salt is unique for each user, and the resulting hashed password is stored along with the salt in the database. Salting passwords is crucial for several reasons, such as password uniqueness preventing rainbow table attacks, and it helps prevent dictionary attacks. Now, SHA1 is considered weak due to vulnerabilities that allow for collision attacks, where two different inputs can produce the same hash. This compromises the integrity of the hashing algorithm, making it susceptible to manipulation by attackers. In the context of password security, SHA1 is also fast and lacks the computational expense required to resist brute-force attacks effectively. Modern parallel architectures significantly boost the speed of password cracking. GPUs, with their parallel processing capabilities, can perform many hash computations simultaneously. This parallelism enables attackers to attempt a vast number of password guesses in a short time, especially when coupled with distributed systems or cloud computing. Multicore CPUs also contribute to increased cracking speed, though GPUs are particularly effective due to their architecture optimized for parallel tasks. Now, different forms of attack would be better to use in certain situations. For example, rainbow tables are useful when attacking hashed passwords stored without salt. They are precomputed tables that map hashes to plaintext passwords, speeding up the password recovery process. However, they are less effective against salted passwords because each salted password requires a unique table. Meanwhile, brute force attacks become more useful when dealing with longer and more complex passwords. Rainbow tables are limited by the key space they cover, and as passwords become longer or include special characters, the table size increases exponentially. Brute force attacks systematically attempt all possible password combinations, making them effective against complex passwords but resource-intensive and time-consuming.

The most impactful revelation in my exploration of parallel computing and its implications for password security and cracking was the substantial boost in program speed achievable through multithreading. While I previously understood the application of multicore processors in servers to handle concurrent requests, witnessing its potential to accelerate program execution was eye-opening, considering we were able to crack hundreds of more passwords in a fraction of the time it took to crack them in a single-threaded program. The challenge that posed a significant struggle for me was the task of effectively dividing a common password file among multiple threads. This process demanded careful consideration to ensure seamless coordination and prevent conflicts between threads. Despite the initial difficulties, overcoming this hurdle became the most valuable lesson, emphasizing the importance of efficient thread management and synchronization in parallel computing for optimal program performance. Overall, it was an extremely challenging project; however, it taught me a lot about parallel programming.