What is my project / how does it work?

For my project I chose the Brute-Force cryptography search problem from the list of embarrassingly parallel topics.

I designed a system which reads in a number of hashed passwords (using MD5 hash), along with the max length that each password might be. This input data is stored in the data/in/ directory in JSON format. My system first sequentially reads this data, and generates a list of hashed password / maxLength tuples. For each tuple, a set of "crackTasks" is generated. Each crack task holds a single char ("prefix") from the set of valid printable characters, as well as the hashed password and max length. A worker thread executes a task by searching every possible string <= maxLength, that begins with the target prefix. The sequential version of my system has a single main thread which loops over every prefix from validChars and tests all possible strings <= target length. The rest of my write up describes the parallel implementation of the system

Note: To avoid pathologically bad inputs, the production version of my system (triggered using "p" flag at start time) randomly shuffles the list of valid chars when search jobs are allocated. This does however create a large degree of randomness in execution time (since there are "good" and "bad" shuffles based on a single target hash), so the benchmark version keeps the char set static to make the analysis of performance per thread more accurate.

Once a list of crack tasks has been generated for a given job, these tasks are pushed into my work-stealing thread scheduler, and the scheduler is launched (call to Run() is made). Each crack task passed into the scheduler also contains a pointer to a channel ('results chan') held by the main thread to store results. My scheduler holds tasks in a DEqueue (henceforth referred to as DEQ) data structure implemented using a doubly linked list (per Prof Samuels suggestion for graduating students, implemented using coarse grained locking). When the scheduler is launched, with a given capacity, it spins up the correct number of workers and stores them in a worker pool. It also builds an array of DEQs, with each worker in the pool being assigned a single DEQ. When the workers receive a signal from the scheduler to run() they launch on a seperate go routine which takes work from their own DEQ, task by task, and executes it until their DEQ is empty. They then attempt to steal work from other DEQs until all DEQs are empty. At this point they exit their run method. When a crack task finds a key that matches the target hash, it sends that key back to the main thread on the results channel.

The scheduler is also set up to have a kill switch that can signal threads to stop working before they have finished executing every task in their DEQ. When the calling thread calls scheduler.Done(), the scheduler calls worker.kill() for each worker in the pool. Each worker has a designated channel (killChan) which it listens to between the execution of each task. When a worker receives a kill() call, it first checks whether it is running, and if it is running it sends a signal to its kill chan which tells the running thread to exit. My implementation uses the Done() method to shut down the system after a valid key is found.

The scheduler also has a Wait() method which allows the calling thread to block until all workers running in the scheduler finish execution. This is implemented using a single wait group. When Run() is called this wait group increments by the number of threads running in the system. When each thread finishes execution it tells the wait group to decrement. This Wait() method is used in my implementation by the calling thread to handle situations an invalid job was provided by the user (i.e. hash isn't solved by key <= max length). After launching the scheduler, the main thread also launches a "canary" which waits for the scheduler to finish. The main thread uses a for loop wrapping a select-case statement to listen to both this canary and the results thread. When a signal is received from either channel, the system exits the loop and returns the result (empty string if invalid job).

Once all jobs have been executed, the system writes the list of result keys to the output path provided by the user.

Challenges Faced:

I initially struggled to determine the best way to divide up my search space so that it could be allocated between tasks. Realizing that I could split it up based on prefix was a major breakthrough that helped me solve this. I was also challenged to implement a Done() and Wait() method in my scheduler, and had to rely on a somewhat convoluted set of channels to pass data between threads that allowed me to solve this.

Hotpots / bottlenecks:
Because reading / writing are trivial with the types of inputs I chose to use, the biggest bottleneck by far was searching the key space for strings that yielded the correct hash. Luckily, this problem is extremely well suited to parallel computation so I was able to reduce the impact of this bottleneck in my concurrent implementation

Testing Machine:

Ran on macbook

What slowed down my parallel implementation?

A few things which were suboptimal about my implementation were coarse grained locking on dequeue and sequential reading and writing. These bottlenecks were much more trivial relative to "crack time" on the large dataset, since this dataset had less tasks than smaller datasets but each crack task was much larger. I believe this fact is what led the system to receive its largest speeds up relative to sequential on the large file.

Possible Optimizations / if i had more time:

This system was set up to crack a small number of hard passwords i.e. (5 or 6 characters). If however I wanted to crack a long list of small passwords I would make some changes. In that case, I would read in crack jobs concurrently from input, and have my scheduler crack many passwords at once, instead of having each thread tackle the same password before shutting down. In a real system I would also consider saving results we had seen previously in a hash table in case they appeared again. I would also have liked to implement my DEqueue using atomic pointer swapping instead of coarse grained locking to allow it to be lockfree.

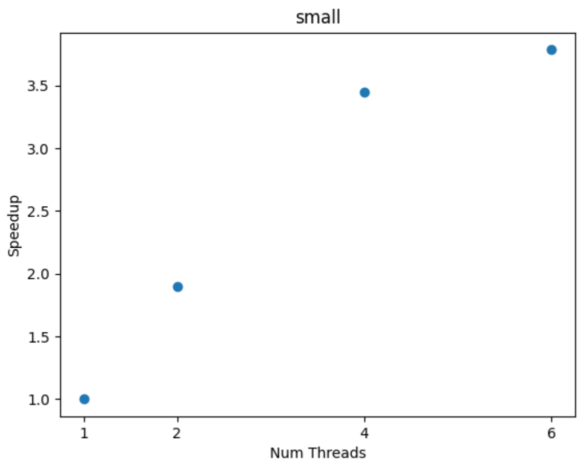
Performance Analysis:

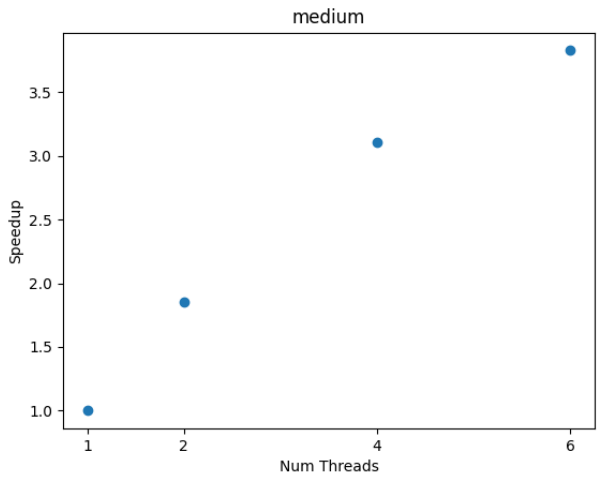
The difficulty of a password (assuming char set shuffled to avoid pathologically good / bad inputs) is proportional to the number of guesses the system must make before exhausting all possibilities. My valid char set has 96 characters in it, so difficulty is ~= 96^(max length)

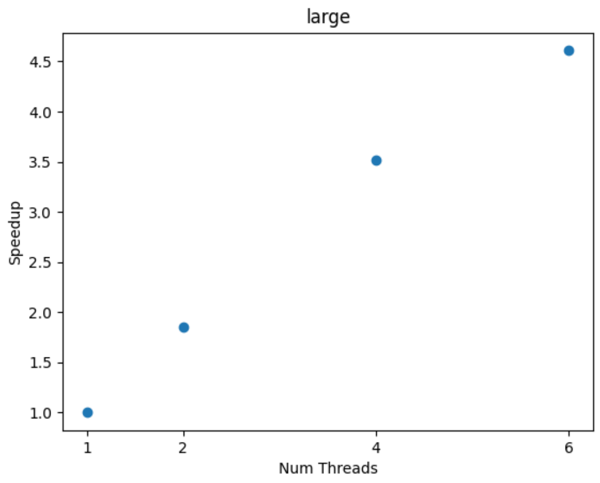
As a result of this, difficulty grows exponentially as a factor of password length. A 5 digit password is almost 100 times as challenging as a 4 digit password. I used this fact to craft different degrees of difficulty for my input files. Each file has only 1-3 passwords, but the length of the passwords increases in each file. My small test file has 3, length 3 passwords and should be handled in a few seconds. The medium file mixes length 3 and 4 passwords and runs from 30 seconds to 1 minute. My large file has a single 5 digit password, which takes the sequential cracker over 20 minutes to break. My system can still handle even longer passwords, but the sequential crack time of these long passwords made it impracticable to include them.

My system performed very well, exhibiting nearly linear speedups as the number of threads increased. I tested my system on my 6-core macbook air so my testing unfortunately only went up to 6 threads, and I'm curious to see whether my performance would continue improving beyond 6 threads with more powerful hardware. The size of the problem seemed to have some impact on performance, for all three problem sizes I saw a roughly linear increase in performance as the number of threads rose, but performance grew fastest on the largest dataset (6 threads were 4.5x as fast as sequential). On the harder password in the large input file, the cost of cracking the password dwarfs the cost of reading / writing, so the parallelized part of the algorithm especially shines. The act of dividing up the prefix search space and searching, is almost perfectly parallel, which is why I chose to design my system the way I did. If my inputs were of a different nature (i.e. dealing with very long lists of short passwords) I would likely see some degradation in performance gain, as the cost of reading / writing data grew. As described above, to deal with this type of input I would likely change my system to read / write concurrently as well as crack.

Graphs:


small
Speedup
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
Num Threads


medium
Speedup
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
Num Threads


large
Speedup
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
Num Threads