Outline

- Summary of problem
- Overview of Sequential Implementation
  - Demonstration
- Overview of Parallel Implementation
  - Demonstration
- Conclusion
Problem Overview

- Takes in any length string text
  - Breaks message into 512-bit blocks
- Hash function outputs 128-bit value
- As of Dec. 2008 the US Government discourages further use of MD5 for cryptographic hashing

Source: [https://www.k-ict.org/v4/online-security/md5-hash/](https://www.k-ict.org/v4/online-security/md5-hash/)

Computational Problem

Investigate & quantify collision time of attack using sequential and parallelized approaches
Memory Management

- The MD5 algorithm always generates a 128 bit (16 bytes) output.
- Storing each MD5 input & output would be 32 bytes (Assuming all inputs are stored as 16 bytes).
- To decrease memory usage and increase the probability to find a collision we decided to only store the first 24 LSBs of the MD5 output, and store all values as integers (4 bytes).
- In that case there is only $2^{24} = 16,777,216$ unique MD5 outputs.
  - Maximum memory usage = $2^{24} * 4 \text{ bytes} * 2 = 134 \text{ MB} * 2 = 268 \text{ MB}$
- We feel as Hashmaps can handle this amount of storage very comfortably.
  - Hash Maps lend a constant, $O(1)$ lookup time as well.

- Rainbow tables provide a more advanced solution and would allow us to increase the number of bits we store in regards to the MD5 output.
  - We reviewed a few papers in terms of rainbow tables and attempted to implement a basic test solution but ultimately gave up after we found that Hashmap provided us a reasonable solution.
Sequential Algorithm

Int start → Hex Input from user
Int stop → Hex Input from user
Int count = start

Create empty HashMap(Hash, Hash Input)

While(count <= stop)
    String Hash = Generate MD5 Hash(count.toString)
    if(Hashmap Contains Hash)
        Print both Hashes that cause collision
        Print Hash output
        System.exit()
    else
        Add Hash & count to hashmap
        count++

Print (“No collision found based on input”)

Sequential Program - Flowchart

Start

Read user input to determine search space

Create empty hash map

Generate input string

Generate 24-bit MD5 hash

Check if output hash exists in hash map

Add output hash & input string to hash map

If no

Print resulting input & hash

End
Sequential Demo
Parallel Algorithm

Int start → Hex Input from user
Int stop → Hex Input from user
Int count = start
Boolean collision = false

Create empty HashMap(Hash, Hash Input)

While(count <= stop)
  ParallelFor(start_idx, end_idx)
    String Hash = Generate MD5 Hash(count.toString)
    if(Hashmap Contains Hash)
      Print both Hashes that cause collision
      Print Hash output
      collision = true
      Stop all threads
    else
      Add Hash & count to hashmap
      count++

If no collision
  Print ("No collision found based on input")
Parallel Program - Flowchart

- Start
- Read user input to determine search space
- Create hash map
- Split work amongst multiple workers with leapfrog schedule
- Generate input string
- Generate 24-bit MD5 hash
- Check if hash exists in hash map
  - No: Add hash to hash map
  - Yes: Stop
- Print resulting input & hash
- End
Parallel Demo
Conclusion

● Our initial testing shows that our hypothesis was accurate
  ○ The more cores searching for a collision, the faster we can find one
  ○ Managing memory is a serious concern, being considered at all stages
  ○ Since our input space is small our run times are generally pretty quick
● Rainbow tables can provide a more advanced solution but Hashmaps allow us to show speed-ups in a simpler implementation
● Additionally, test on multiple nodes
THANK YOU

QUESTIONS ?