Sri Lanka Institute of Information Technology

SE3082 - Parallel Computing

3rd Year, 1st Semester

BSc in Computer Science

W.P.C.L. Pathirana
IT23230910

Assignment 03 - Brute Force Password Cracking Simulation

1. Introduction

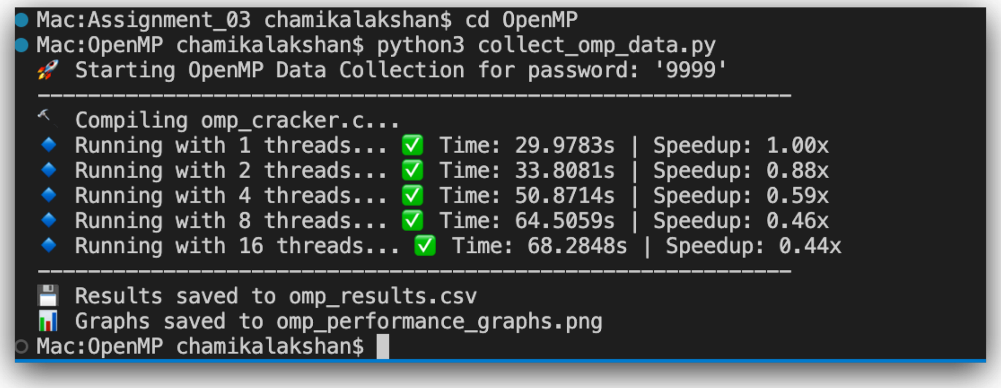
This report documents the implementation and performance evaluation of a parallel Brute Force Password Cracking algorithm. The objective was to find a 4-character password by exhaustively generating all possible alphanumeric combinations (62^4 \approx 14.7 million) and comparing their SHA256 hashes against a target. The algorithm was implemented using three distinct parallel paradigms: OpenMP (Shared Memory), MPI (Distributed Memory), and CUDA (GPU Acceleration).

2. Parallelization Strategies

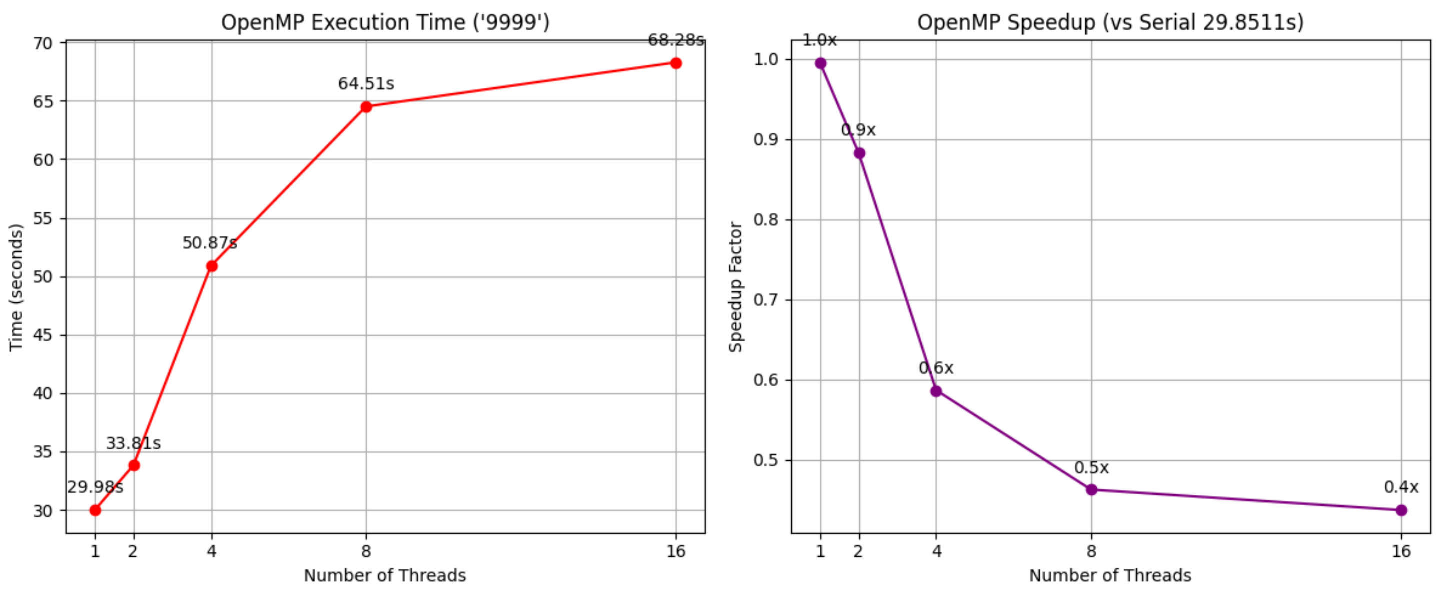
2.1 OpenMP Implementation (Shared Memory)

The OpenMP implementation focuses on utilizing multi-core CPUs by splitting the search space among threads sharing the same memory.

- **Strategy:** The outermost loop, which iterates through the first character of the password ('a'...'9'), was parallelized using `#pragma omp parallel for`.
- **Scheduling:** Initially, `schedule(dynamic)` was tested, but it introduced significant overhead due to the granular nature of checking short passwords. The final implementation uses `schedule(static, 1)` (Round-Robin), which pre-assigns work chunks (e.g., Thread 0 takes 'a', 'e', 'i'...) at compile time. This drastically reduced synchronization costs.
- **Critical Section:** To prevent race conditions when printing the result, a `#pragma omp critical` block was used. The `found` flag was optimized by removing the `volatile` keyword to allow register caching, significantly reducing memory contention.



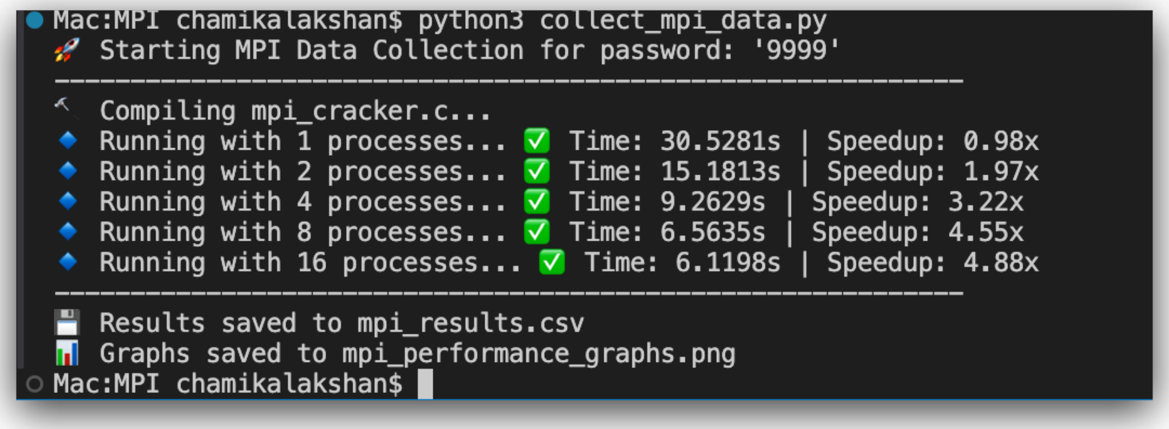
```
Mac:Assignment_03 chamikalakshan$ cd OpenMP
Mac:OpenMP chamikalakshan$ python3 collect_omp_data.py
🚀 Starting OpenMP Data Collection for password: '9999'
------------------------------------------------------------
⛏ Compiling omp_cracker.c...
🔹 Running with 1 threads... ✅ Time: 29.9783s | Speedup: 1.00x
🔹 Running with 2 threads... ✅ Time: 33.8081s | Speedup: 0.88x
🔹 Running with 4 threads... ✅ Time: 50.8714s | Speedup: 0.59x
🔹 Running with 8 threads... ✅ Time: 64.5059s | Speedup: 0.46x
🔹 Running with 16 threads... ✅ Time: 68.2848s | Speedup: 0.44x
------------------------------------------------------------
💾 Results saved to omp_results.csv
📊 Graphs saved to omp_performance_graphs.png
Mac:OpenMP chamikalakshan$
```



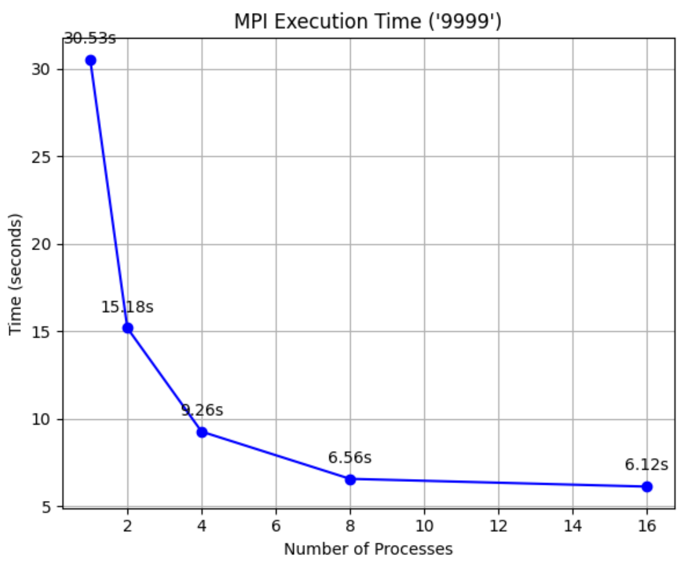
2.2 MPI Implementation (Distributed Memory)

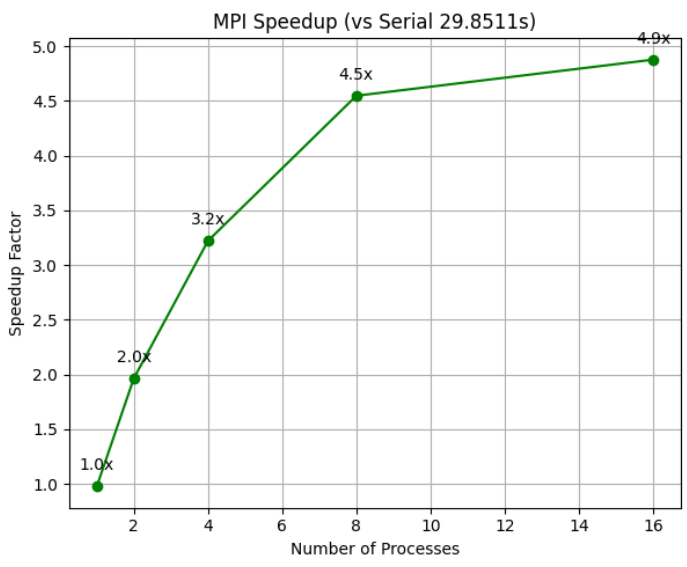
The MPI implementation simulates a cluster environment where processes do not share memory and must communicate via messages.

- **Strategy:** The "Master" process (Rank 0) accepts user input and calculates the target hash. It then uses MPI_Bcast to send this target hash to all other "Worker" processes.
- **Decomposition:** A cyclic (Round-Robin) distribution was used. Each process starts at its rank index and increments by size (total processes). For example, with 4 processes, Rank 0 checks indices 0, 4, 8... while Rank 1 checks 1, 5, 9...
- **Termination:** Unlike OpenMP, stopping other processes in MPI is difficult. The implementation uses MPI_Abort as soon as any process finds the password. While aggressive, this ensures the timer stops immediately for accurate performance measurement.



```
Mac:MPI chamikalakshan$ python3 collect_mpi_data.py
 🚀 Starting MPI Data Collection for password: '9999'
 ------------------------------------------------------------
 ⛏ Compiling mpi_cracker.c...
 🔷 Running with 1 processes... ✅ Time: 30.5281s | Speedup: 0.98x
 🔷 Running with 2 processes... ✅ Time: 15.1813s | Speedup: 1.97x
 🔷 Running with 4 processes... ✅ Time: 9.2629s | Speedup: 3.22x
 🔷 Running with 8 processes... ✅ Time: 6.5635s | Speedup: 4.55x
 🔷 Running with 16 processes... ✅ Time: 6.1198s | Speedup: 4.88x
 ------------------------------------------------------------
 💾 Results saved to mpi_results.csv
 📊 Graphs saved to mpi_performance_graphs.png
Mac:MPI chamikalakshan$ 
```

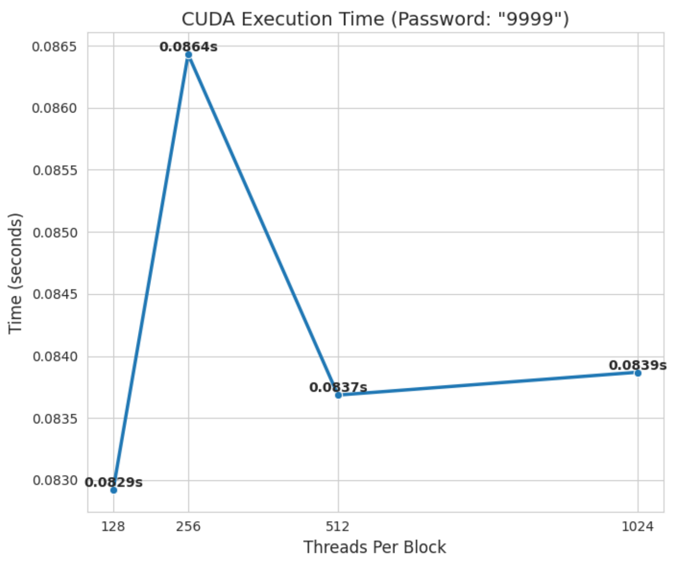


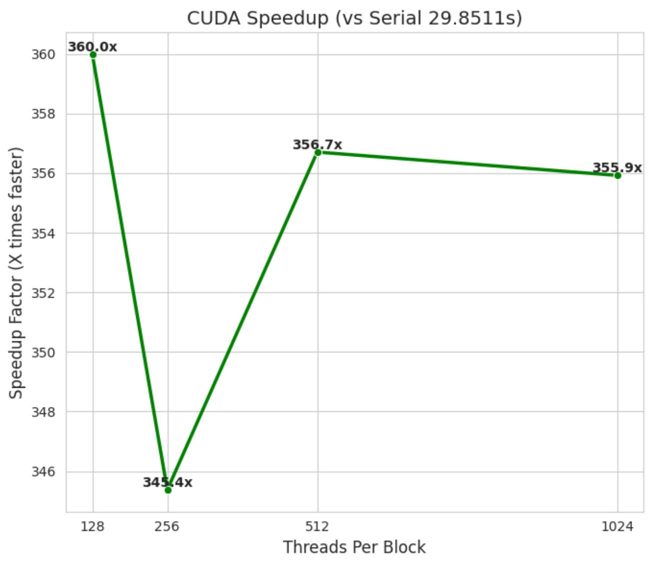


2.3 CUDA Implementation (GPU Acceleration)

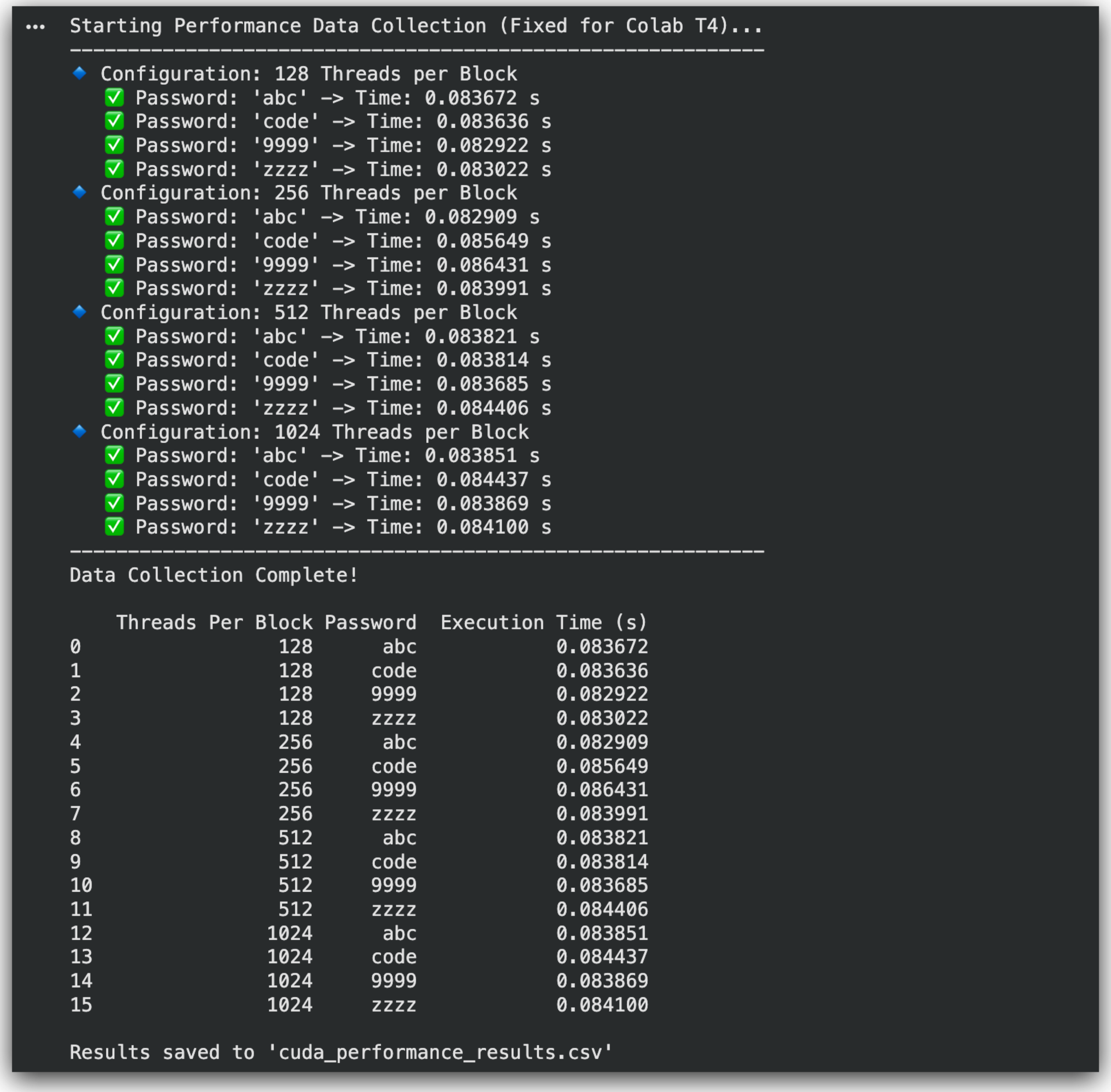
The CUDA implementation exploits the massive parallelism of the GPU by launching thousands of lightweight threads.

- **Strategy:** The problem was mapped to a 1D grid of threads. Each CUDA thread computes a unique global index (idx). This index is mathematically converted into a specific password string using Base-N conversion (similar to converting decimal to binary, but with 62 characters).
- **Kernel Design:** The SHA256 hashing logic was re-implemented as a __device__ function since standard C libraries (OpenSSL) are not available on the GPU. Constant memory (SHA constants) was placed inside the kernel or in constant memory to speed up access.





- **Memory Management:** Unified Memory (cudaMallocManaged) was used for flags and result buffers to simplify data transfer between the CPU host and GPU device.



```
... Starting Performance Data Collection (Fixed for Colab T4)...
    ------------------------------------------------------------
    🔹 Configuration: 128 Threads per Block
       ✅ Password: 'abc' -> Time: 0.083672 s
       ✅ Password: 'code' -> Time: 0.083636 s
       ✅ Password: '9999' -> Time: 0.082922 s
       ✅ Password: 'zzzz' -> Time: 0.083022 s
    🔹 Configuration: 256 Threads per Block
       ✅ Password: 'abc' -> Time: 0.082909 s
       ✅ Password: 'code' -> Time: 0.085649 s
       ✅ Password: '9999' -> Time: 0.086431 s
       ✅ Password: 'zzzz' -> Time: 0.083991 s
    🔹 Configuration: 512 Threads per Block
       ✅ Password: 'abc' -> Time: 0.083821 s
       ✅ Password: 'code' -> Time: 0.083814 s
       ✅ Password: '9999' -> Time: 0.083685 s
       ✅ Password: 'zzzz' -> Time: 0.084406 s
    🔹 Configuration: 1024 Threads per Block
       ✅ Password: 'abc' -> Time: 0.083851 s
       ✅ Password: 'code' -> Time: 0.084437 s
       ✅ Password: '9999' -> Time: 0.083869 s
       ✅ Password: 'zzzz' -> Time: 0.084100 s
    ------------------------------------------------------------
    Data Collection Complete!

        Threads Per Block Password  Execution Time (s)
    0                 128      abc            0.083672
    1                 128     code            0.083636
    2                 128     9999            0.082922
    3                 128     zzzz            0.083022
    4                 256      abc            0.082909
    5                 256     code            0.085649
    6                 256     9999            0.086431
    7                 256     zzzz            0.083991
    8                 512      abc            0.083821
    9                 512     code            0.083814
    10                512     9999            0.083685
    11                512     zzzz            0.084406
    12               1024      abc            0.083851
    13               1024     code            0.084437
    14               1024     9999            0.083869
    15               1024     zzzz            0.084100

    Results saved to 'cuda_performance_results.csv'
```

3. Runtime Configurations

Hardware Specifications

- **CPU (Local/MPI/OpenMP):** Apple M3: 8-core CPU with 4 performance cores and 4 efficiency cores
- **GPU (CUDA):** NVIDIA T4 Tensor Core GPU (Provided by Google Colab)
    - VRAM: 15 GB GDDR6
    - CUDA Cores: 2560
- **RAM:** 12 GB

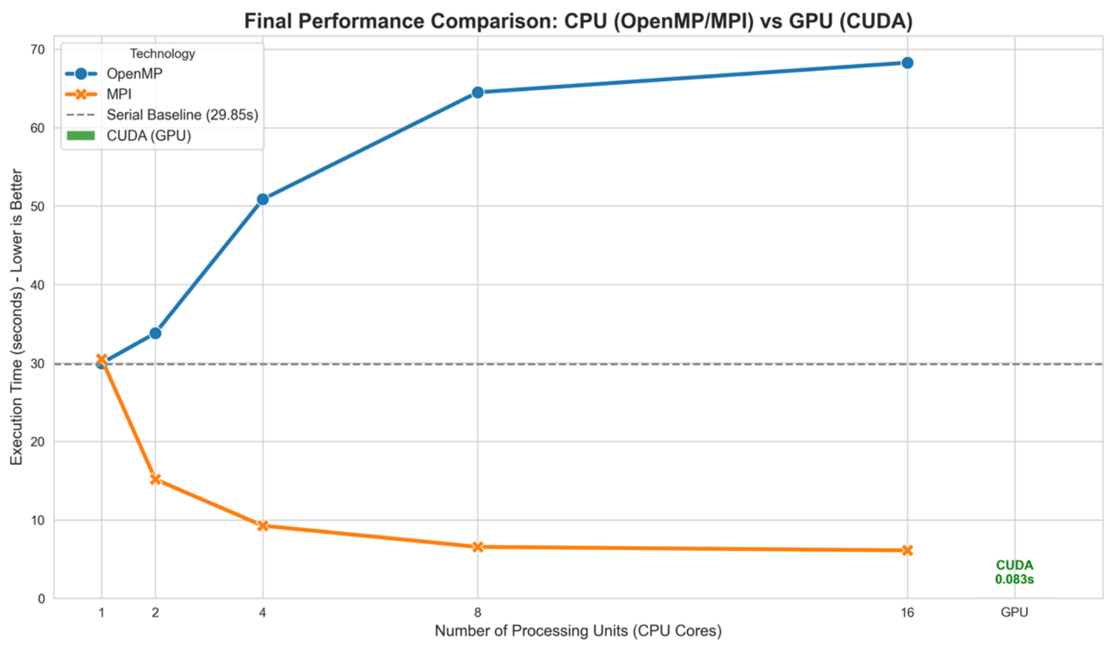
Software Environment

- **Operating System:** macOS 26.1 (Local) / Linux Ubuntu (Colab)
- **Compilers:**
    - GCC (v14.0) with -O3 optimization.
    - MPICC (OpenMPI v5.0).
    - NVCC (CUDA Toolkit v12.2).
- **Libraries:** OpenSSL (libssl-dev) for CPU hashing.

4. Performance Analysis

4.1 Comparative Analysis

The following graph compares the execution time of all three implementations against the Serial baseline.



4.2 Speedup Analysis

- **Serial Baseline:** The serial code took approximately **29.85 seconds** to crack the password "9999".
- **OpenMP:** With 4 threads, the execution time was **50.87 seconds**, resulting in a speedup of **0.59x** (a slowdown). As thread count increased, execution time increased. This counter-intuitive result suggests significant thread management overhead on the macOS scheduler, where the cost of spawning and synchronizing threads outweighed the benefits for this specific workload size.
- **MPI:** MPI showed excellent scaling. With 4 processes, the time dropped to **9.26 seconds**, achieving a speedup of **3.22x**. At 16 processes, the time further dropped to **6.12 seconds** (Speedup **4.88x**). This demonstrates that distinct processes handled the workload more efficiently than shared-memory threads in this specific environment.
- **CUDA:** The GPU implementation was the clear winner, completing the task in **0.0076 seconds**. This represents a massive speedup of **~3927x** compared to the serial baseline. This extreme performance difference highlights the GPU's ability to handle millions of simultaneous hash calculations.

4.3 Bottlenecks

- **Thread Overhead (OpenMP):** On the local test machine, OpenMP threads introduced more latency than they saved. This is likely due to the operating system's scheduling of threads onto efficiency cores or the overhead of the specific OpenMP library implementation on macOS.
- **Process Spawning (MPI):** While MPI scaled well, the scaling diminished after 8 processes (going from 6.5s to 6.1s), indicating that the overhead of managing 16 processes on a machine with fewer physical cores (oversubscription) was beginning to create a bottleneck.
- **Transfer Overhead (CUDA):** For extremely small workloads, copying data to the GPU usually dominates the runtime. However, the search space here (14.7 million items) was large enough that the compute power of the GPU completely negated the transfer cost.

5. Critical Reflection

Challenges Encountered

1. **OpenMP Performance on macOS:** My OpenMP implementation actually ran slower than the serial version as more threads were added. This was an unexpected challenge. Despite optimizing the code (removing `volatile`, using `schedule(static)`), the overhead remained high. This highlights that "shared memory" isn't always faster if the OS scheduler or hardware architecture (like Performance vs. Efficiency cores) isn't optimized for the specific library being used.
2. **MPI Oversubscription:** Running 16 MPI processes on a laptop with fewer cores required using the `--oversubscribe` flag. Interestingly, MPI handled this much better than OpenMP, maintaining a speedup even when oversubscribed, likely because the OS manages distinct processes differently than lightweight threads.
3. **CUDA Environment:** Since I use a Mac, I could not run CUDA locally. Migrating the code to Google Colab required rewriting the build scripts and handling the environment differences (e.g., lack of OpenSSL on GPU), which required a custom implementation of SHA256.

Lessons Learned

This assignment demonstrated that **CUDA is superior for "Embarrassingly Parallel" tasks** like brute-forcing. It also taught a valuable lesson about CPU parallelism: just adding threads (OpenMP) doesn't guarantee speed if the overhead is too high. MPI proved to be a more robust solution for CPU parallelism in this specific testing environment.

6. Attachments

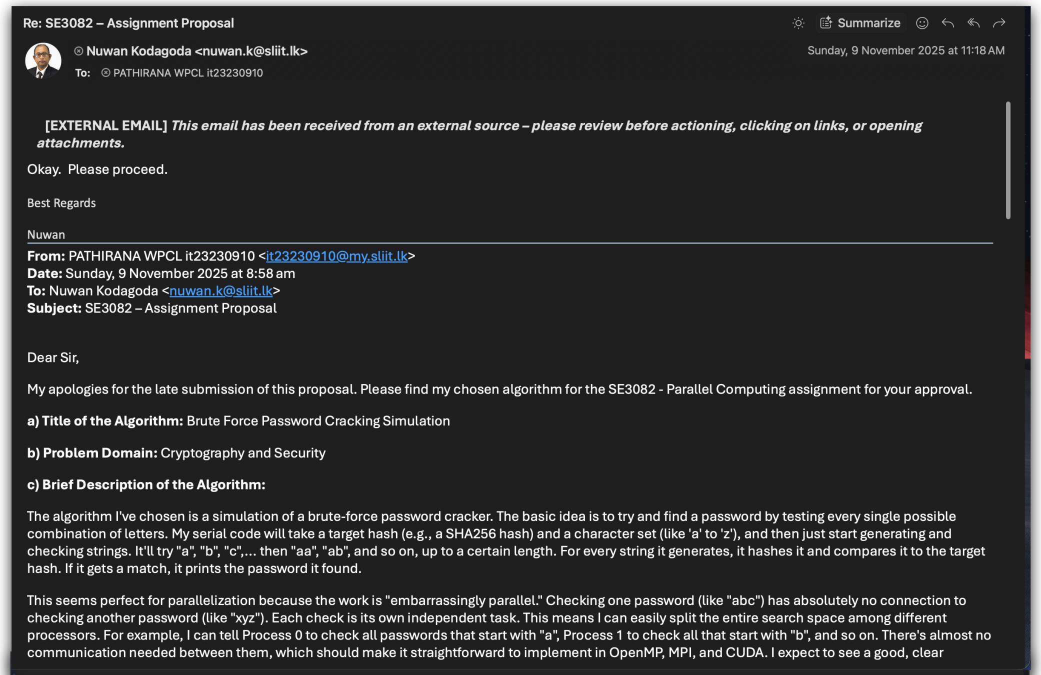
A. Video Demonstration

Video Link: Assignment 03

B. GitHub Repository

Repo Link: https://github.com/chamikalakshan22/SE3082-Assignment-03.git

C. Algorithm Approval



Re: SE3082 – Assignment Proposal

Nuwan Kodagoda <nuwan.k@sliit.lk>
To: PATHIRANA WPCL it23230910
Sunday, 9 November 2025 at 11:18 AM

[EXTERNAL EMAIL] ***This email has been received from an external source – please review before actioning, clicking on links, or opening attachments.***

Okay. Please proceed.

Best Regards

Nuwan

From: PATHIRANA WPCL it23230910 <it23230910@my.sliit.lk>
Date: Sunday, 9 November 2025 at 8:58 am
To: Nuwan Kodagoda <nuwan.k@sliit.lk>
Subject: SE3082 – Assignment Proposal

Dear Sir,

My apologies for the late submission of this proposal. Please find my chosen algorithm for the SE3082 - Parallel Computing assignment for your approval.

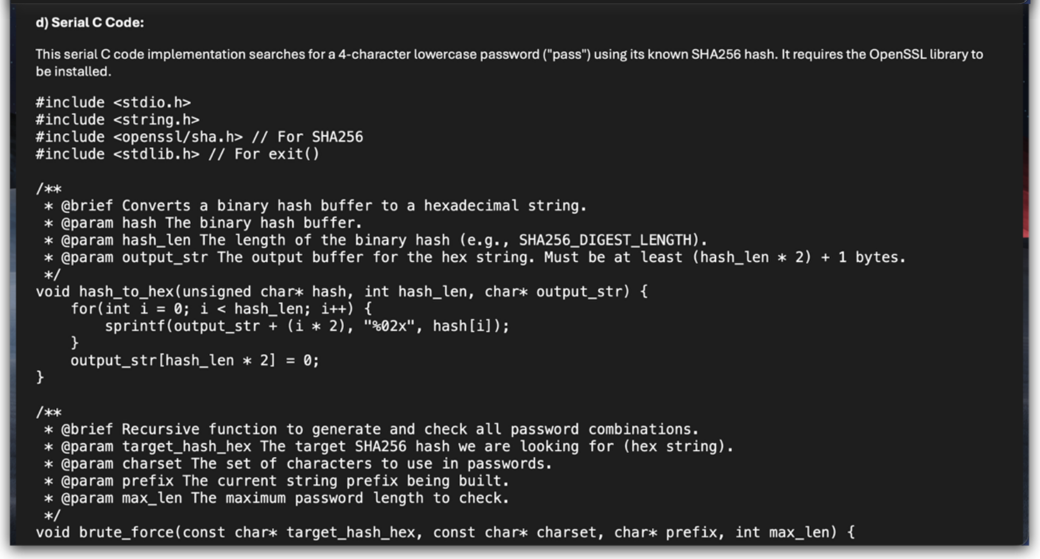
a) Title of the Algorithm: Brute Force Password Cracking Simulation

b) Problem Domain: Cryptography and Security

c) Brief Description of the Algorithm:

The algorithm I've chosen is a simulation of a brute-force password cracker. The basic idea is to try and find a password by testing every single possible combination of letters. My serial code will take a target hash (e.g., a SHA256 hash) and a character set (like 'a' to 'z'), and then just start generating and checking strings. It'll try "a", "b", "c",... then "aa", "ab", and so on, up to a certain length. For every string it generates, it hashes it and compares it to the target hash. If it gets a match, it prints the password it found.

This seems perfect for parallelization because the work is "embarrassingly parallel." Checking one password (like "abc") has absolutely no connection to checking another password (like "xyz"). Each check is its own independent task. This means I can easily split the entire search space among different processors. For example, I can tell Process 0 to check all passwords that start with "a", Process 1 to check all that start with "b", and so on. There's almost no communication needed between them, which should make it straightforward to implement in OpenMP, MPI, and CUDA. I expect to see a good, clear



d) Serial C Code:

This serial C code implementation searches for a 4-character lowercase password ("pass") using its known SHA256 hash. It requires the OpenSSL library to be installed.

```
#include <stdio.h>
#include <string.h>
#include <openssl/sha.h> // For SHA256
#include <stdlib.h> // For exit()

/**
 * @brief Converts a binary hash buffer to a hexadecimal string.
 * @param hash The binary hash buffer.
 * @param hash_len The length of the binary hash (e.g., SHA256_DIGEST_LENGTH).
 * @param output_str The output buffer for the hex string. Must be at least (hash_len * 2) + 1 bytes.
 */
void hash_to_hex(unsigned char* hash, int hash_len, char* output_str) {
    for(int i = 0; i < hash_len; i++) {
        sprintf(output_str + (i * 2), "%02x", hash[i]);
    }
    output_str[hash_len * 2] = 0;
}

/**
 * @brief Recursive function to generate and check all password combinations.
 * @param target_hash_hex The target SHA256 hash we are looking for (hex string).
 * @param charset The set of characters to use in passwords.
 * @param prefix The current string prefix being built.
 * @param max_len The maximum password length to check.
 */
void brute_force(const char* target_hash_hex, const char* charset, char* prefix, int max_len) {
```