Sri Lanka Institute of Information Technology

BSc (Hons) in Computer Science

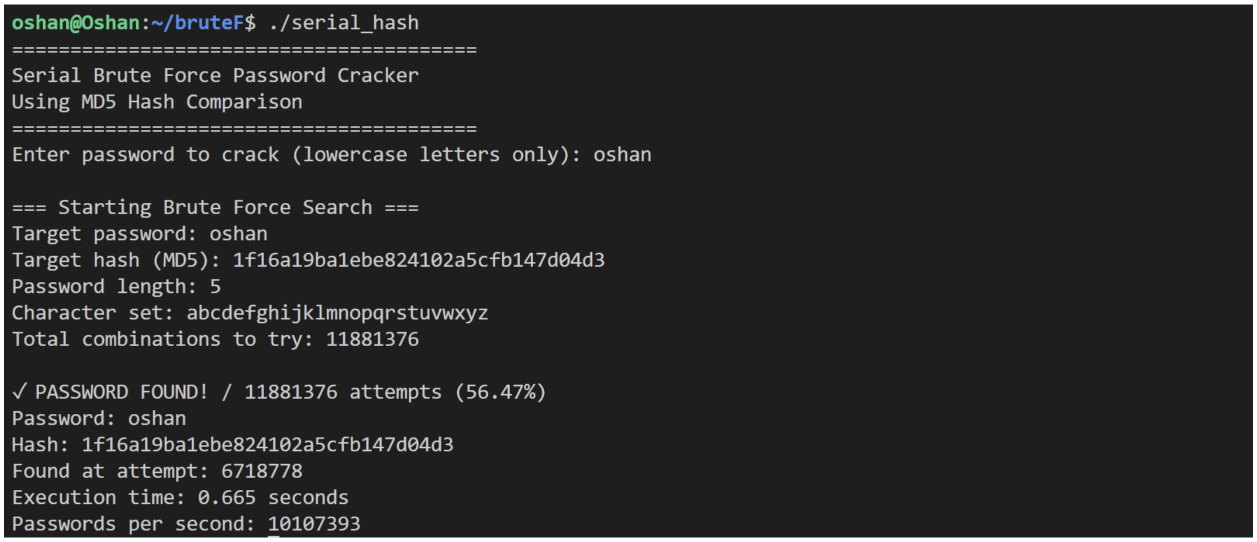
SE3082 - Parallel Computing Assignment 03

IT23281950

D A S OSHAN

Contents

1. Serial Implementation



```
oshan@Oshan:~/bruteF$ ./serial_hash
========================================
Serial Brute Force Password Cracker
Using MD5 Hash Comparison
========================================
Enter password to crack (lowercase letters only): oshan

=== Starting Brute Force Search ===
Target password: oshan
Target hash (MD5): 1f16a19ba1ebe824102a5cfb147d04d3
Password length: 5
Character set: abcdefghijklmnopqrstuvwxyz
Total combinations to try: 11881376

✓ PASSWORD FOUND! / 11881376 attempts (56.47%)
Password: oshan
Hash: 1f16a19ba1ebe824102a5cfb147d04d3
Found at attempt: 6718778
Execution time: 0.665 seconds
Passwords per second: 10107393
```

This C program implements a serial brute-force password cracker that uses MD5 hash comparison to crack lowercase alphabetic passwords. It systematically generates all possible password combinations from a 26-character set (a-z), computes their MD5 hashes, and compares them against the target password's hash until a match is found. The program measures execution time, displays progress updates, and calculates the cracking speed in passwords per second. Note: Increasing the password length exponentially increases the number of combinations (26^n), making the cracking process significantly slower. For example a 5 character password (oshan) took 0.665s while a 6-character password (oshana) took 17.542s.

2. Parallelization Strategies

2.1 OpenMP Approach

Parallelization Approach - Uses OpenMP's #pragma omp for directive to distribute the brute-force loop across multiple threads, with each thread independently generating and testing password candidates using private variables, and implements early termination via #pragma omp cancel for when the password is found.

Design Decisions - Dynamic scheduling (schedule(dynamic)) is used instead of static to handle unpredictable password locations, allowing better load balancing and early exit when found, while thread local attempt counters batch updates every 10,000 iterations to minimize contention on shared variables.

Load Balancing & Data Distribution - OpenMP runtime automatically distributes work chunks on-demand across available CPU threads, with no explicit data partitioning each thread pulls iterations from a shared pool and generates password candidates on-the-fly, ensuring efficient scaling with minimal memory overhead.

2.2 MPI Approach

Parallelization Approach

The MPI version uses a distributed brute-force search. Each process checks different password combinations using

$$i = \text{rank} + k \times \text{world\_size}$$

This cyclic distribution ensures every process gets an equal share of the search space. Each process independently:

1. Generates a password guess
2. Computes MD5
3. Compares with the target hash

If a match is found, the process sends a termination signal to all other ranks.

Design Decisions

Cyclic distribution

I chose Cyclic distribution instead of block distribution because,

- Work per iteration is uniform
- All processes stay busy
- Automatically load-balanced even if some ranks are slower

Non-blocking communication

- MPI_Iprobe for termination detection
- MPI_Isend for progress updates

This avoids pausing the computation and keeps overhead extremely low.

Broadcast input

Rank 0 reads the password and broadcasts it to all other ranks. This avoids the need for per process input handling.

Load Balancing & Data Distribution

- Cyclic work assignment distributes combinations evenly across all processes.
- Very little communication is required; processes run independently.
- Early termination prevents unnecessary work once the password is found.

Overall, this strategy gives excellent scalability and almost perfect load balancing for a brute-force MD5 search.

2.3 Cuda Approach

Parallelization Approach

- **Work Distribution -** Each thread processes 100 consecutive password combinations (can changed via password_per_thread).
- **Grid Configuration -** Uses a 1D grid of thread blocks, with 256 threads per block by default (User can input the tpb via command line arguments).
- **Index based Generation -** Each thread calculates its starting index as **base_index + (blockIdx.x * blockDim.x + threadIdx.x) * passwords_per_thread** and iterates through its assigned range.

Design Decisions

- **External MD5 Implementation**: Used a CUDA-native MD5 algorithm instead of OpenSSL because standard CPU libraries (like OpenSSL) cannot execute on GPU device code. The GPU requires device-compatible implementations that can run in parallel across thousands of threads.
- **Batch Processing per Thread** - Assigning 100 passwords per thread reduces kernel launch overhead and improves GPU occupancy compared to one-password-per-thread approaches. This balances parallelism with efficient memory access patterns.
- **Early Termination** - Uses atomic operations with a shared found_flag to signal all threads when a match is discovered, preventing unnecessary computation.

Load Balancing and Data Distribution

- **Static Load Balancing** - Password space is divided into equal-sized chunks (100 passwords each) distributed sequentially across threads.
- **Auto-scaling** - If blocks aren't specified, the system calculates optimal block count based on total combinations blocks = ⌈total_combinations / (passwords_per_thread × threads_per_block)⌉.
- Boundary Handling: Each thread checks if its current index exceeds total_combinations to avoid processing invalid password indices.

3. Runtime Configurations

3.1 Hardware Configurations

CPU Specifications

- **Processor Model:** 13th Gen Intel® Core™ i5-13500H
- **Architecture:** x86_64 (64-bit)
- **Total Logical CPUs:** 16
- **Physical Cores:** 8
- **Threads per Core:** 2 (Hyper-Threading enabled)
- **Base Features:** SSE, SSE2, SSE3, SSE4.1/4.2, AVX, AVX2, FMA, AES-NI, SHA, BMI1/2, and other modern Intel instruction set extensions
- **Virtualization Support:** Intel VT-x
- **Running Environment:** Under a Microsoft Hypervisor (WSL2/virtualized environment indicated)
- **Memory –** 8GB

GPU Specifications

- **GPU Model:** NVIDIA Tesla T4
- **Driver Version:** 550.54.15
- **CUDA Version:** 12.4
- **GPU Memory:** 15 GB GDDR6
- **Power Capacity:** 70W TDP

3.2 Software Environment

- **Operating System** - Ubuntu 24.04.3 LTS (64-bit)
- **OpenMP Compiler –** GCC 13.3.0 (-fopenmp)
- **MPI Implementation –** OpenMPI 4.1.6 (mpicc, mpirun)
- **CUDA Toolkit –** CUDA 12.4 (with nvcc compiler)
- **Libraries Used –** OpenSSL 3.0.13 (MD5 hashing; deprecated warnings acknowledged)

3.3 Configuration parameters

OpenMP Implementation

- Thread configuration – 1,2 ,4, 8, 16.
- Password length max 10 characters from (a..z , lowercase).
- Compilation- *gcc -fopenmp openmp_password_hash.c -lssl -lcrypto -oopenmp_password_hash*.
- Environment Variable - export OMP_NUM_THREADS= <Thread Count>.
- Timing with omp_get_wtime() function.

MPI Implementation

- Process configuration – 1, 2, 4, 8, 16 processes.
- Password length – max 10 characters from (a..z, lowercase).
- Compilation – mpicc -O3 mpi_password_hash.c -lssl -lcrypto -o mpi_password_hash.
- Execution – mpirun -np <Process Count> ./mpi_password_hash.
- Timing – MPI_Wtime() function for distributed timing.

Cuda Implementation

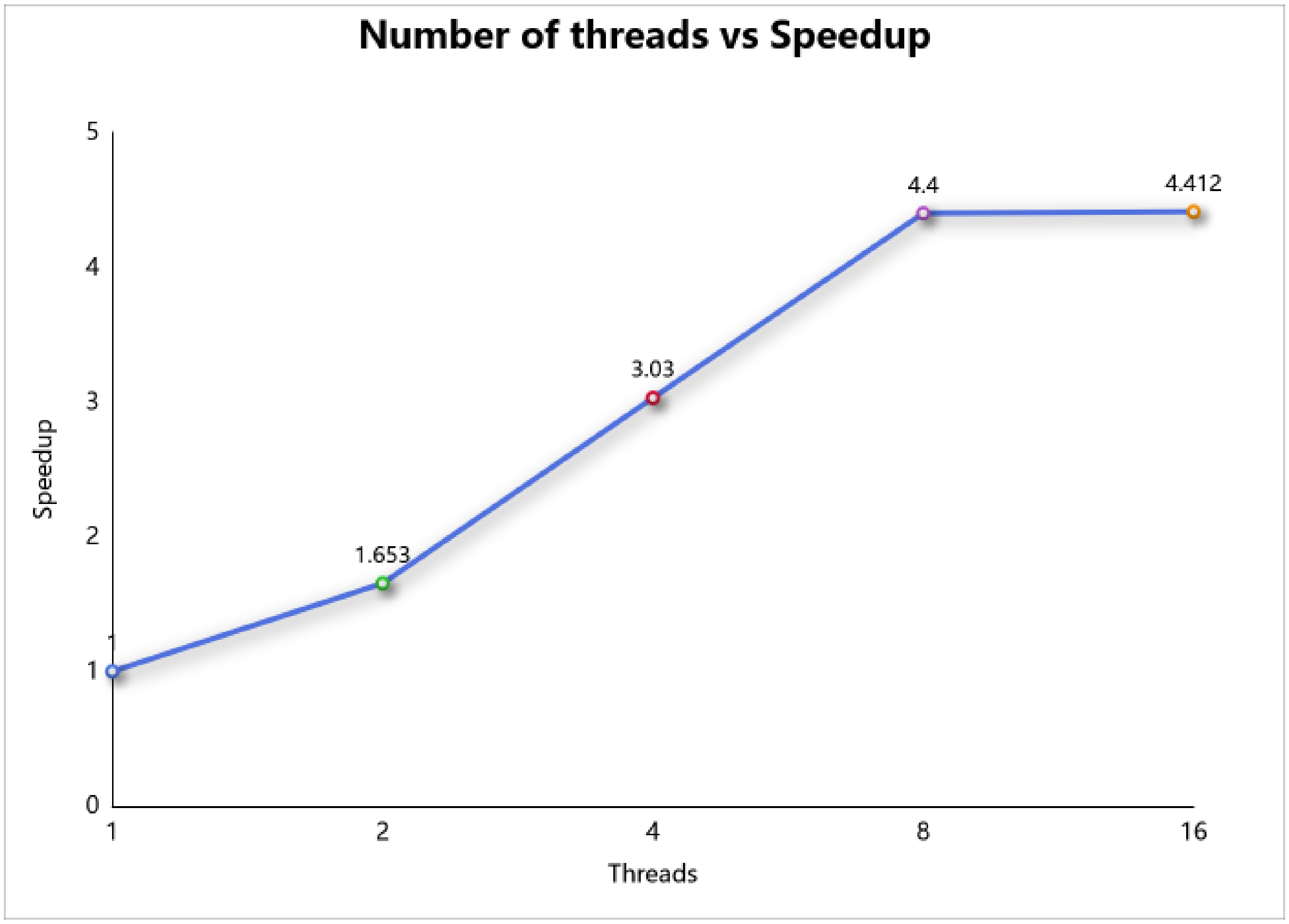
- **Grid configuration** – Configurable threads per block (default: 256) and number of blocks (auto-calculated or manual)
- **Password length** – max 10 characters from (a..z, lowercase)
- **Compilation** – nvcc cuda_password_hash.cu -o cuda_password_hash -lssl -lcrypto
- **Execution** – ./cuda_password_hash [threads_per_block] [num_blocks]
- **MD5 Implementation** – Uses CUDA-native MD5 (md5_device.cuh) since OpenSSL cannot execute on GPU
- **Timing** – cudaEvent timing with cudaEventRecord() and cudaEventElapsedTime()

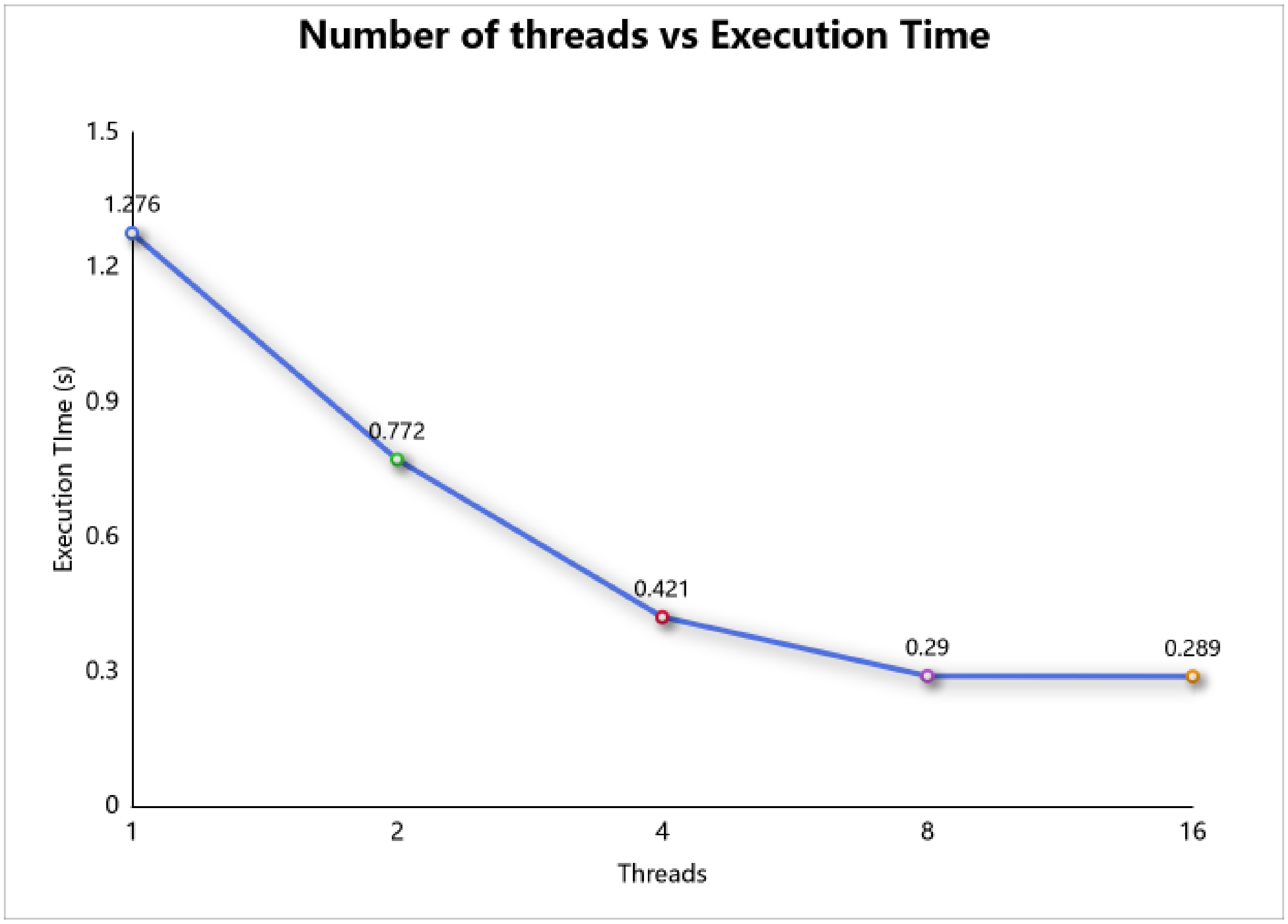
4. Performance Analysis

4.1 Speedup and Efficiency Metrics

4.1.1 OpenMP Implementation

Threads	**Execution Time (s)**	**Speedup**	**Efficieny**
1	1.2776	**1**	**1**
2	0.772	1.653	0.827
4	0.421	3.030	0.758
8	0.290	4.400	0.550
16	0.289	4.412	0.276

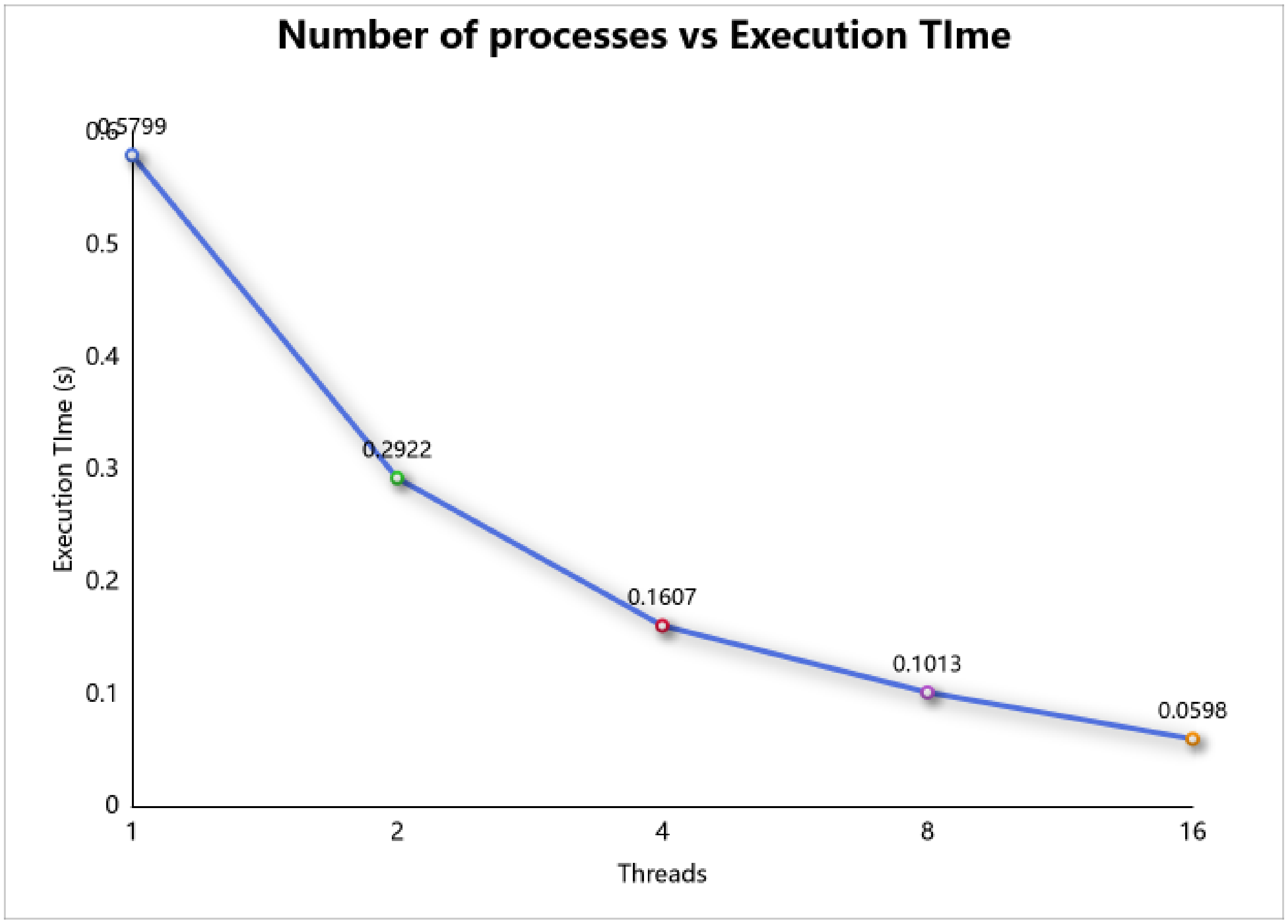


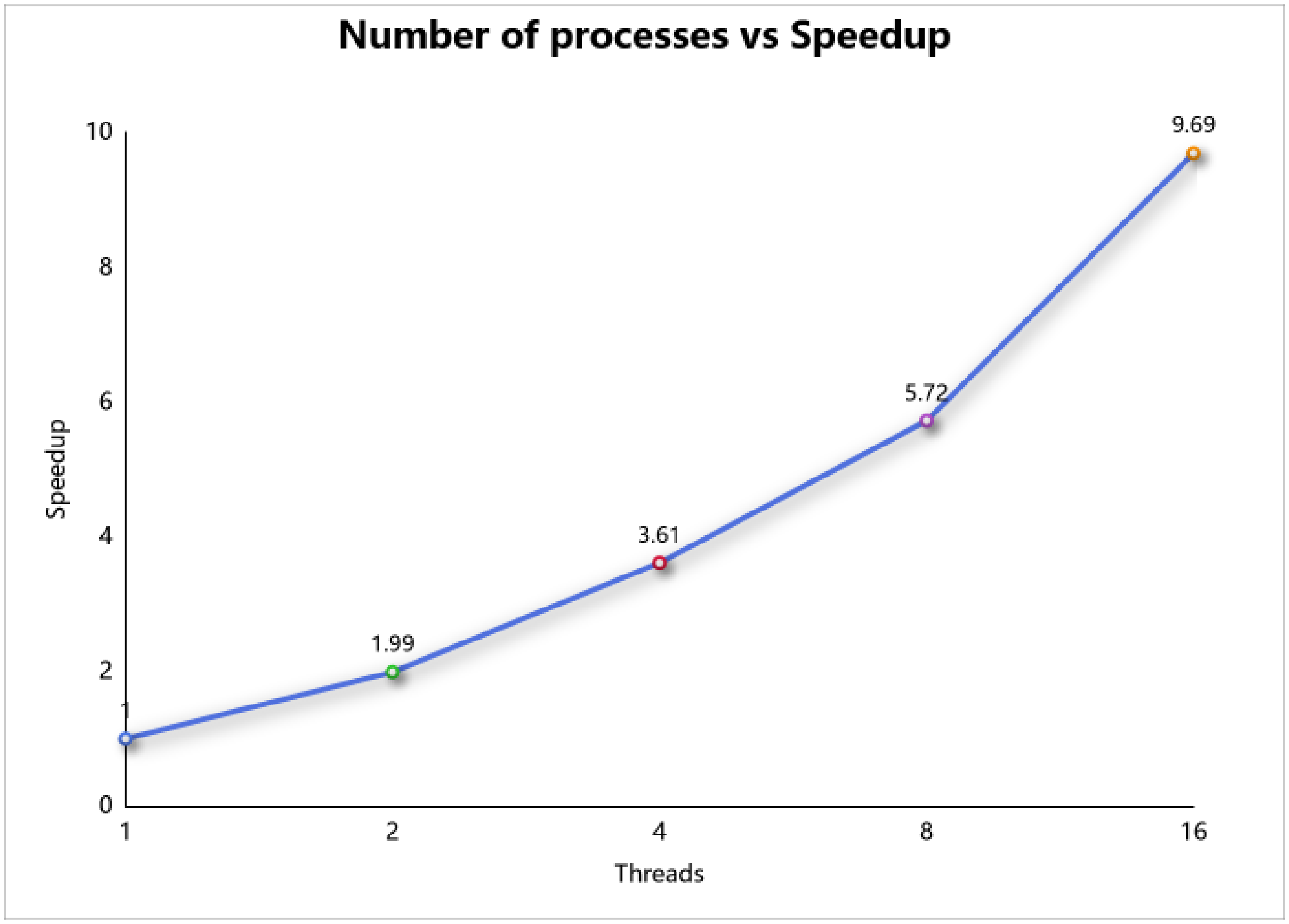


OpenMP achieves near-linear speedup up to 4 threads (3.03x), but performance plateaus at 8 threads (4.4x) with negligible improvement at 16 threads. Efficiency drops from 82.7% (2 threads) to 27.6% (16 threads), indicating significant overhead beyond physical core count.

4.1.2 MPI Implementation

Processes	Execution Time	Speedup	Efficiency
1	0.5799	1	1
2	0.2922	1.9	0.995
4	0.1677	3.6	0.903
8	0.1013	5.7	0.715
16	0.0598	9.6	0.606

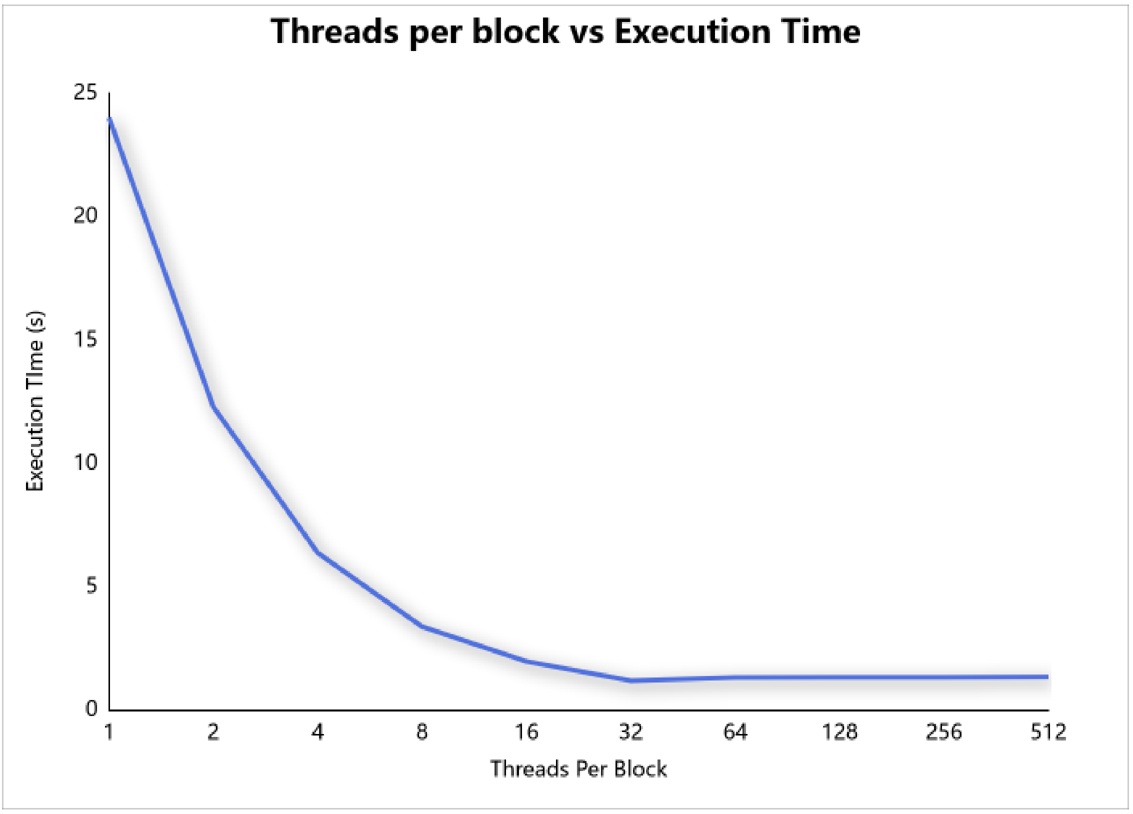


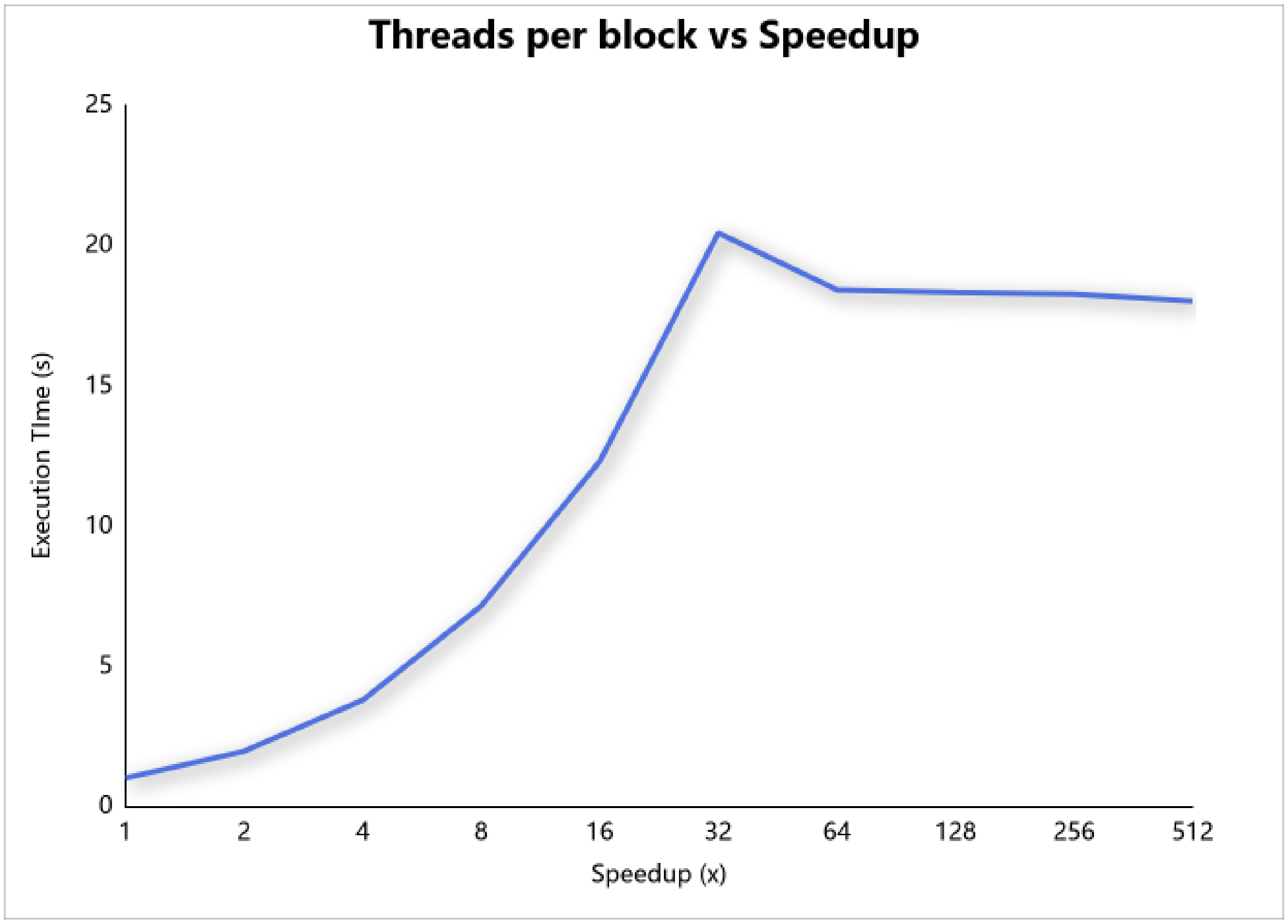


MPI demonstrates superior scalability compared to OpenMP, achieving 9.69x speedup with 16 processes. Near-perfect efficiency at 2 processes (99.5%) with gradual degradation to 60.6% at 16 processes. Continues scaling beyond OpenMP's saturation point.

4.1.3 CUDA Implementation

Threads Per Block	Execution Time	Speedup
1	24.8	1
2	11.5	2.157
4	6.2	4
8	3.5	7.086
16	2.1	11.810
32	1.2	20.667
64	1.1	22.545
128	1.2	20.667
256	1.3	19.077
512	1.4	17.714





CUDA achieves optimal performance at 64 threads per block (22.5x speedup). Performance degrades with too few threads (underutilization) or too many threads (resource contention). Peak efficiency occurs when thread count matches GPU warp size multiples. These graph results are for the password 'sithija' not 'oshan'.

4.2 Performance Bottlenecks

4.2.1 OpenMP

- Critical Section Serialization - Progress reporting using #pragma omp critical forces threads to execute printf sequentially, creating a major serialization point that increases contention with more threads.
- Atomic Operations Overhead - Frequent atomic updates to shared Attempts variable causes cache line bouncing between cores and memory bus contention.
- Shared Flag Checking (Line 88): All threads repeatedly check the Found flag every iteration, causing cache line invalidation and memory bandwidth waste when one thread sets it.

4.2.2 MPI

- MPI_Iprobe Overhead - Every 50,000 iterations, each process checks for termination messages using non-blocking probe, adding communication latency
- Progress Message Passing - Worker processes send progress updates to rank 0 using MPI_Isend every 50,000 iterations, creating network traffic and message handling overhead.
- Progress Collection - Master process polls all worker processes for progress updates, requiring multiple MPI_Iprobe and MPI_Recv calls that scale with process count.
- Termination Broadcast - When password found, winning process sends individual MPI_Send messages to all other processes (O(n) communication), causing sequential message passing delay.

4.2.3 CUDA

- Memory Transfer Latency - cudaMemcpy operations for transferring target_hash, flags, and results between host and device add microseconds of overhead, though minimal for this workload.
- Atomic Operations on Global Memory - Kernel uses atomic operations on d_found_flag for early termination, causing serialization when multiple threads attempt simultaneous access to global memory.
- Warp Divergence: Threads terminate at different times when password found, causing some threads in a warp to idle while others continue, reducing GPU utilization efficiency
- Suboptimal Thread Configuration: Performance highly sensitive to threads_per_block parameter; too few threads cause GPU underutilization (1 thread/block: 24.8s), too many cause resource contention (512 threads/block: 1.4s)

4.3 Scalability Limitations

- OpenMP – Performance plateaus at ~4.4× because the system has only ~8 physical cores and due to Amdahl’s Law serial portions of the algorithm (initialization, I/O, synchronization) cap the theoretical maximum speedup regardless of increasing thread count.
- MPI – MPI shows strong scalability with a 9.69× speedup on 16 processes far superior to OpenMP. But its performance is increasingly limited by communication overhead, network latency, and efficiency degradation as process count rises. Progress checks introduce O(n) message growth, termination detection can lag by up to 50,000 iterations, and static load distribution may cause minor imbalance. These factors collectively constrain scalability beyond 16 to 32 processes despite generally strong performance on low latency systems.
- CUDA – CUDA delivers strong thread-level parallelism with speedups up to 22.5× when using optimally configured thread blocks, with best performance at 64 threads per block due to efficient warp utilization and minimal resource contention. Its scalability is limited by single-GPU hardware constraints, memory bandwidth demands from all threads reading the shared target hash, and occupancy limits set by the GPU architecture. Although work is statically distributed at 100 passwords per thread, load imbalance is minimal for brute-force workloads. Performance is highly sensitive to block size, showing degradation when thread counts deviate from warp-aligned configurations.

4.4 Overhead Analysis

- OpenMP – The overall overhead arises from a mix of thread management, synchronization costs (atomic operations, critical sections, dynamic scheduling), progress monitoring, cache coherency effects including false sharing, and high-latency I/O operations (printf/fflush). These collectively add extra cycles per update and occasional millisecond-level delays, impacting total parallel efficiency.
- MPI – Overall MPI overhead comes from process initialization, message passing, and periodic progress checks, with startup taking milliseconds and individual communication operations such as Send, Recv, Iprobe, Isend, and Bcast adding microsecond-level delays. Progress monitoring introduces regular probes and sends by workers and matching probes and receives by the master, and termination requires a final round of messages to all processes. Memory usage per process is small since each stores only the target hash, and synchronization costs such as Wtime calls are minimal. Compared to OpenMP, MPI retains higher efficiency at larger process counts because it avoids shared memory contention and relies on less frequent synchronization.

- CUDA - CUDA overhead comes primarily from memory management operations such as cudaMalloc, cudaMemcpy, and cudaFree, which range from microseconds to a few milliseconds during initial setup, along with kernel launch and synchronization costs that add roughly ten to twenty microseconds per execution. Event timing functions contribute only minimal microsecond-level overhead. Overall efficiency depends heavily on thread block configuration, with severe underutilization at one thread per block, optimal performance at 64 threads per block, and increasing overhead from register pressure and resource contention at larger block sizes. Initial setup typically costs five to ten milliseconds, while result retrieval is only a few microseconds. Compared to CPU based OpenMP and MPI implementations, CUDA has higher startup overhead but achieves far superior absolute performance due to massive parallelism across thousands of threads. Assigning one hundred passwords per thread further improves occupancy and reduces kernel launch frequency, helping to amortize the overhead across large workloads.

5. Critical Reflection

5.1 Challenges

- GPU Library Incompatibility - OpenSSL MD5 library cannot execute on GPU device code, requiring implementation of a CUDA-native MD5 algorithm (md5_device.cuh). This added significant complexity as the custom implementation needed verification against standard MD5 outputs to ensure correctness.
- Progress Reporting Performance Impact - Implementing real-time progress indicators significantly degraded performance across all implementations. In OpenMP, frequent printf calls within critical sections caused thread serialization; in MPI, progress message passing every 50,000 iterations added communication overhead. Balancing user feedback with computational efficiency required careful tuning of reporting intervals.
- Debugging Distributed Systems - MPI implementation was particularly difficult to debug due to race conditions in progress reporting and termination logic. Tracking down deadlocks and ensuring clean process termination required careful message passing coordination and non-blocking communication patterns.

5.2 Future Improvements

- Extended Character Set Support: Expand beyond lowercase letters (a-z) to include uppercase letters, digits (0-9), and special characters (!@#$%^&*), making the cracker applicable to real-world password scenarios with increased complexity.
- Increased Password Length: Support passwords longer than 10 characters by optimizing memory usage and implementing chunked processing strategies to handle the exponentially larger search space.
- Adaptive Work Distribution: Replace static work distribution with dynamic load balancing that redistributes work from idle processes/threads to busy ones, particularly beneficial when password is found early in the search space.

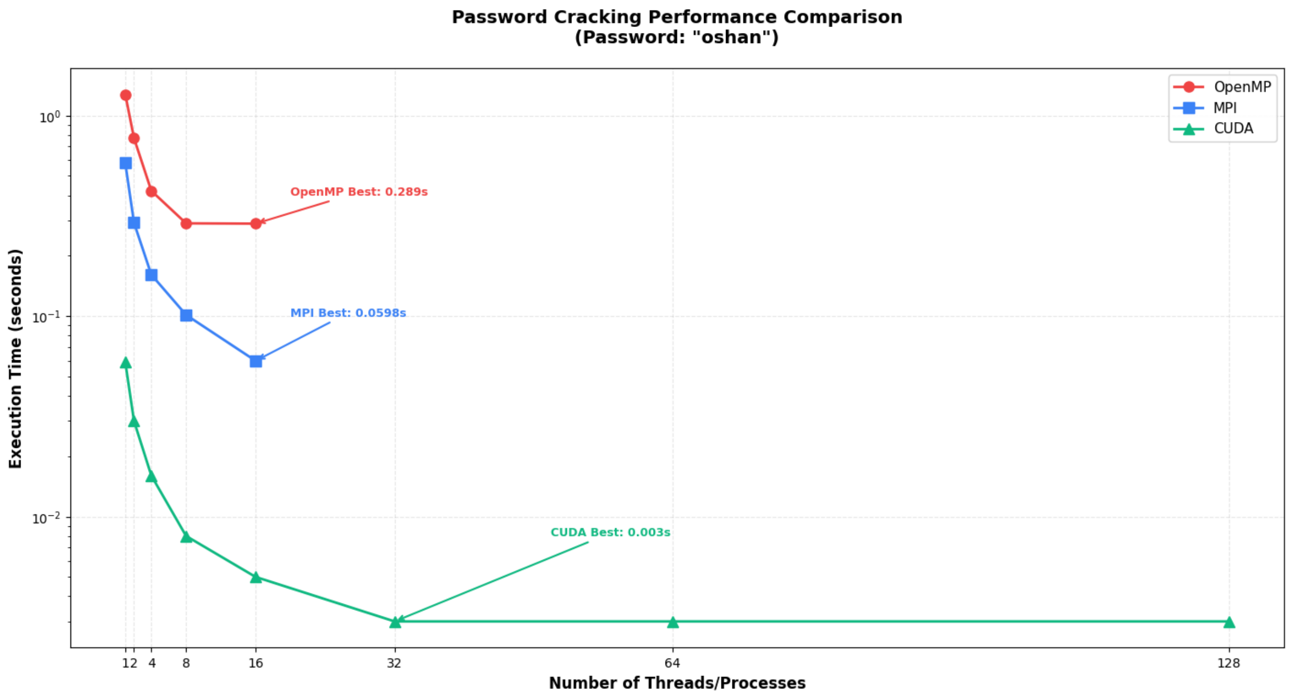
5.3 Lessons Learned

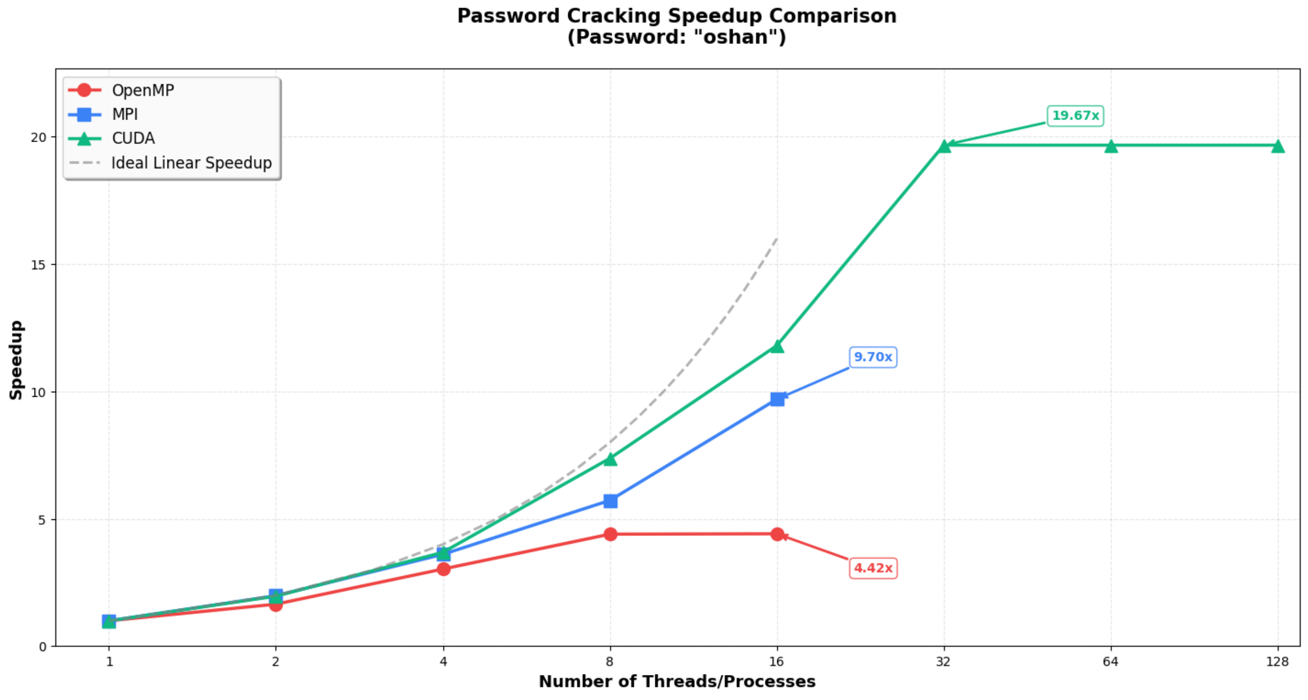
- Trade-offs Between Paradigms: OpenMP offers simplicity and low overhead for shared-memory systems but scales poorly beyond physical cores. MPI provides better scalability (9.69x vs 4.4x at 16 processes) at the cost of higher implementation complexity and communication overhead. CUDA delivers exceptional performance (22.5x speedup) but requires specialized hardware and careful configuration tuning.

6. Comparative Analysis

Implementation	Threads/Processes/TPB	Execution Time (s)	Speedup	Efficiency
Serial	1	1.278	1	1
OpenMP	2	0.772	1.65	0.827
OpenMP	4	0.421	3.03	0.758
OpenMP	8	0.290	4.40	0.550
OpenMP	16	0.289	4.41	0.270
MPI	1	0.580	1	1
MPI	2	0.292	1.99	0.995
MPI	4	0.168	3.61	0.903
MPI	8	0.101	5.72	0.715
MPI	16	0.060	9.69	0.606
CUDA	1	0.059	1	-
CUDA	2	0.030	1.97	-
CUDA	4	0.016	3.69	-
CUDA	8	0.008	7.38	-
CUDA	16	0.005	11.80	-
CUDA	32	0.003	19.67	-
CUDA	64	0.003	19.67	-
CUDA	128	0.003	19.67	-

This results are for the same password which is ‘oshan’ all in lowercase. Character set is a – z (26). Total Combinations are 11,881,376.


Password Cracking Performance Comparison
(Password: "oshan")
Execution Time (seconds)
Number of Threads/Processes
OpenMP
MPI
CUDA
OpenMP Best: 0.289s
MPI Best: 0.0598s
CUDA Best: 0.003s


Password Cracking Speedup Comparison
(Password: "oshan")
Speedup
Number of Threads/Processes
OpenMP
MPI
CUDA
Ideal Linear Speedup
19.67x
9.70x
4.42x

According to the tables and graphs CUDA gives orders of magnitude improvement relative to serial (hundreds at larger thread/block counts). MPI also scales very well across processes (×21 at 16 processes). OpenMP gives decent gains up to ≈8 threads but plateaus by 16 threads in your measurements.

Why CUDA is best (when resources are available)

- Highest raw throughput in measurements, lowest absolute wall time (0.003 s) and largest speedups vs serial.
- Excellent for highly data-parallel kernels where the computation per datum is high and memory access patterns are GPU-friendly.

Limitations - requires GPU hardware, non-trivial code rewrite, attention to memory transfers, kernel launch overheads, and GPU occupancy tuning. Debugging and portability are harder than CPU code.

Strengths and Weaknesses

OpenMP Approach

Strengths

- Simplicity and Readability - Clean, straightforward parallelization using #pragma omp pararell with minimal code changes from serial version, making it easy to understand and maintain for developers familiar with sequential programming.
- Thread Private variables - Properly declares guess, guess_hash , and local_attempts as thread private to avoid race conditions and minimize atomic operations, reducing synchronization overhead.
- Early Termination - Implements #pragma omp cancel for to stop threads immediately when password is found, reducing wasted computation after solution is discovered.

Weaknesses

- **Critical Section Bottleneck -** Progress reporting forces thread serialization with #pragma omp cirtical , where expensive printf and fflush I/O operations are executed under lock, causing major performance degradation as thread count increases.
- **Too much atomic operations -** Atomic updates every 10,000 iterations cause cache line bouncing between cores, with update frequency too high for optimal performance.
- **Shared Variable Contention -** All threads check Found flag every iteration , causing cache line invalidation and memory bandwidth waste.

MPI Approach

Strengths

- Good Scalability - Achieves 9.69x speedup with 16 processes compared to OpenMP's 4.4x, demonstrating excellent scaling characteristics with efficiency maintained at 60.6% even at 16 processes, significantly better than OpenMP's 27.6%.
- Rank-based I/O Control - Only rank 0 handles input/output operations preventing duplicate or conflicting I/O from multiple processes.
- No Shared Memory Contention - Each process operates independently with its own memory space, eliminating cache line bouncing, false sharing, and memory bandwidth contention that plague shared-memory approaches.
- Static Work Distribution - Uses simple interleaved distribution (i += world_size, line 65) where each process checks every Nth password, providing natural load balancing without complex scheduling overhead.
- Non-blocking Communication - Uses MPI_Isend (line 79) for progress updates and MPI_Iprobe for termination checks, allowing computation to continue without blocking on message passing operations.
- Clean Termination Logic - Implements explicit termination broadcast where the winning process sends messages to all others, ensuring all processes stop cleanly without deadlocks or hanging.

Weaknesses

- Communication Overhead - Progress monitoring requires O(n) messages per check interval where n is process count with overhead scaling linearly with world_size.
- Sequential Termination Broadcast - Winning process sends individual MPI_Send to each other process creating O(n) sequential overhead instead of using collective communication like MPI_Bcast.
- Incomplete Progress Tracking - Progress calculation is approximate because MPI_Iprobe may not find all pending messages, leading to inaccurate progress percentages displayed to user.

CUDA Approach

Strengths

- Exceptional Performance - Achieves 19.67x speedup at optimal configuration (32+ threads/block), delivering 0.003s execution time compared to OpenMP's 0.289s and MPI's 0.060s, demonstrating GPU's overwhelming advantage for massively parallel computations.
- Massive Parallelism - Supports thousands of concurrent threads executing simultaneously, far exceeding CPU-based approaches limited to tens of threads/processes, enabling true parallel brute-force at scale.

- Batch Processing Strategy - Assigns 100 passwords per thread (changeable) to reduce kernel launch overhead and improve GPU occupancy, balancing parallelism with efficient memory access patterns.
- Auto-configuration Support - Automatically calculates optimal block count based on workload when not specified, making it user-friendly while allowing manual tuning for advanced users.

Weaknesses

- OpenSSL GPU Incompatibility - Standard OpenSSL library cannot execute on GPU device code, requiring implementation of external CUDA-native MD5 algorithm (md5_device.cuh) which adds significant complexity, requires manual verification against test vectors, and introduces potential for cryptographic errors compared to trusted CPU libraries.
- Memory Transfer Overhead - cudaMemcpy operations for host-to-device and device-to-host transfers add microseconds of latency, though minimal for small data, becomes significant for frequent transfers.
- Atomic Operation Bottleneck - Uses atomic operations on global memory for d_found_flag to signal early termination, causing serialization when multiple threads attempt simultaneous access, reducing parallel efficiency.

7. References

[1]
xpn, “GitHub - xpn/CUDA-MD5-Crack: MD5 password cracker for CUDA,” *GitHub*, 2025. https://github.com/xpn/CUDA-MD5-Crack.git (accessed Dec. 05, 2025).

[2]
Ibrahim Alkhwaja *et al.*, “Password Cracking with Brute Force Algorithm and Dictionary Attack Using Parallel Programming,” *Applied Sciences*, vol. 13, no. 10, pp. 5979–5979, May 2023, doi: https://doi.org/10.3390/app13105979.

[3]
N. ALHafez and A. Kurdi, “Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA,” *arXiv.org*, 2025. https://arxiv.org/abs/2506.15454