Elian Garcia
Prof. Mallouh

Performance Comparison Report: Serial vs. Parallel Implementations Using MPI and CUDA

This report evaluates the performance improvements achieved through parallel computing techniques using both MPI and CUDA . Three projects were analyzed: a password-cracking program parallelized using MPI, a matrix multiplication program implemented in CUDA, and a Dijkstra shortest-path algorithm also parallelized in CUDA. Each project was tested using varied input sizes to measure how well parallelization improved runtime compared to serial execution. Results are summarized using tables and discussed in relation to algorithmic structure, hardware utilization, and scalability.

---

1. Password Cracker (Serial C++ vs. MPI with 4 Cores)

The password-cracking program attempts all possible combinations until the correct password is found. This makes the task highly parallelizable because independent processes can test separate portions of the keyspace.

Runtime Comparison

Implementation	Runtime avg (s)	Speedup
Serial C++	39 s	1.0×
MPI (4 cores)	11 s	**3.5×**

Discussion

Parallelizing with MPI significantly reduced execution time, achieving a **3.5× speedup**.. While ideal scaling with 4 cores would be 4×, achieving 3.5× suggests efficient load balancing and low synchronization cost.

The speedup is bounded mainly by:

- Uneven distribution of search space (if password found early by one core).

- MPI communication necessary at the end to broadcast termination.

Overall, MPI proved to be highly effective for brute-force search.

---

2. Matrix Multiplication (Serial C++ vs. CUDA)

Matrix multiplication requires roughly O(n^3) operations for square matrices. GPUs excel at this workload because thousands of threads can compute partial products in parallel.

The test case used was a **2000 × 2000 matrix multiplied by a 2000 × 2000 matrix**, producing a 2000 × 2000 output matrix.

Runtime Comparison

Implementation	Runtime avg (ms)	Speedup
Serial C++	100,000 ms	1.0×
CUDA	341 ms	**293×**

Discussion

The CUDA version achieved a **293× speedup**, meaning the GPU computed the same operation nearly 300 times faster than the CPU.

- Thousands of lightweight CUDA threads execute multiplications simultaneously.

- GPU cores are optimized for high-throughput floating-point operations.

---

3. Dijkstra’s Algorithm (Serial C++ vs. CUDA)

Dijkstra’s algorithm is more challenging to parallelize because it relies heavily on:

- Priority queue operations
- Repeated extraction of the next minimum-distance vertex
- Sequential dependency across iterations

Despite this, a parallel CUDA implementation was tested.

Runtime Comparison

Implementation	Runtime avg (ms)	Speedup
Serial C++	600 ms	1.0×
CUDA	400 ms	**1.5×**

Discussion

The CUDA version achieved a **1.5× speedup**, a meaningful 50% improvement. Unlike matrix multiplication, Dijkstra's algorithm is not strictly parallel:

- Each relaxation step depends on the result of the previous iteration.
- Updating the priority queue is difficult to parallelize efficiently.
- Some GPU threads remain idle due to irregular graph structure.

parallelizing the "edge relaxation" step allowed partial speedup.

Still, reducing runtime from 600 ms to 400 ms demonstrates that even partially parallelizable algorithms can benefit from CUDA acceleration.

---

4. Cross-Project Performance Comparison

The table below summarizes all three projects:

Project	Serial Time	Parallel Time	Parallel Method	Speedup

Password Cracker	39 s	11 s	MPI (4 cores)	**3.5×**
Matrix Multiplication	100,000 ms	341 ms	CUDA	**293×**
Dijkstra's Algorithm	600 ms	400 ms	CUDA	**1.5×**

Analysis Across Projects

- **Highest speedup:** Matrix multiplication (293×) — because GPUs excel at massively parallel, arithmetic-dense workloads.
- **Moderate speedup:** Password cracking (3.5×) — MPI performs well for parallel brute-force tasks.
- **Lowest speedup:** Dijkstra's algorithm (1.5×) — limited by dependencies.

---

5. Conclusion

Across all three projects, parallel computing methods significantly improved performance, but the degree of improvement varied based on workload characteristics:

- MPI offered strong speedup for tasks with independent work segments (password cracking).
- CUDA provided extraordinary acceleration for highly parallel mathematical workloads (matrix multiplication).
- For algorithms with inherent sequential constraints (Dijkstra's), only moderate improvement was possible.