FINAL REPORT – Parallel Trapezoidal Rule (OpenMP, MPI, CUDA)

Author: Kanchan Anbalagan
IT23279452
Course: SE3082 – Parallel Computing

1. Introduction

This project implements the trapezoidal numerical integration method using four execution models: Serial, OpenMP, MPI, and CUDA. The goal is to evaluate and compare how each parallel programming paradigm accelerates a workload consisting of millions of independent function evaluations. Because each trapezoid calculation is independent, the algorithm is highly suitable for parallelization on CPUs and GPUs. The work includes developing the four implementations, running performance experiments, and analysing speedup, efficiency, and scalability.

---

2. Parallelization Strategies

2.1 Serial Baseline

The serial program iterates through all subdivisions from $i = 1$ to $n-1$, evaluates $f(x) = \sin(x)$, accumulates the sum, and applies the trapezoidal formula. This version provides the correctness reference and baseline time for speedup calculations.

2.2 OpenMP

The loop is parallelized using `#pragma omp parallel for reduction(+:sum)` to distribute iterations among threads. Because each iteration has the same cost, static scheduling ensures even distribution. Only one reduction occurs at the end, minimizing synchronization overhead. This model benefits from shared memory and minimal communication.

2.3 MPI

The interval is divided among MPI ranks. Each process computes its local segment and sends its partial sum to rank 0 using `MPI_Reduce`. A remainder-handling strategy ensures balanced workloads even when n is not divisible by the number of processes. MPI works

well up to the number of available CPU cores and demonstrates process-level parallelism.

2.4 CUDA

CUDA launches thousands of threads, each handling a strided set of subdivisions. Each thread accumulates a local sum and performs a single `atomicAdd(double)` at the end. GPU timing is measured using CUDA events. The Tesla T4 GPU enables massive parallelism and extremely fast execution for large values of `n`.

---

3. Runtime Configurations

CPU System

- Apple M-series CPU (8 logical threads)
- GCC / Clang + OpenMP support
- OpenMPI for MPI execution

GPU System (Google Colab)

- NVIDIA Tesla T4
- Compute Capability: 7.5
- CUDA Toolkit: 12.x
- Memory: 16 GB

Common Parameters

- Input size: up to **n = 10,000,000**
- OpenMP threads tested: 1–8
- MPI processes tested: 1, 2, 4, 8
- CUDA threads/block: 64, 128, 256, 512

4. Performance Analysis

4.1 OpenMP

- Strong scaling up to 8 threads
- Best time: **0.015739 s**

- Best speedup: **3.10×**
- Bottlenecks: thread scheduling, reduction cost, shared memory bandwidth

4.2 MPI

- Best performance at 4 processes: **0.012809 s**, **3.58× speedup**
- Efficiency drops at 8 processes due to oversubscription and communication overhead
- MPI is effective but limited by physical CPU core count on single-node hardware

4.3 CUDA

- For n = 10M, runtime ~ **0.0353 s** across all thread configurations
- GPU achieves extremely high parallel throughput
- Performance dominated by memory bandwidth, not thread configuration
- For much larger n, CUDA outperforms CPU methods significantly

4.4 Comparative Graph Interpretation (Optional for your PDF)

- Execution time: CUDA < MPI < OpenMP < Serial
- Speedup: CUDA highest for sufficiently large n; MPI > OpenMP for CPU-only
- OpenMP has lowest overhead but is limited by CPU core count
- MPI shows strong scaling until cores saturate
- CUDA saturates GPU resources quickly and gives the best large-n performance

---

5. Comparative Analysis

Using the best configuration of each model:

Model	Best Time (s)	Best Speedup	Notes
Serial	0.048822	1.0×	Baseline
OpenMP	0.015739	3.10×	Good shared-memory performance
MPI	0.012809	3.58×	Best CPU-based performance
CUDA	~0.0353	Varies with large-n	Fastest raw throughput for large datasets

Summary:

- **CUDA** provides the highest raw parallelism and is the best choice for large-scale integration when GPU resources are available.
- **MPI** gives the best CPU performance on multi-core systems up to core count.
- **OpenMP** is simple, efficient, and well suited for single-node shared-memory execution but limited by thread count.

6. Critical Reflection

Implementing four versions of the same algorithm highlighted the differences between shared-memory, distributed-memory, and GPU-based execution. OpenMP was the easiest to implement, while MPI required careful load balancing and understanding of communication cost. CUDA provided the most performance potential but required more attention to memory access patterns and kernel configuration. Hardware limitations, especially CPU core count and Colab GPU constraints, influenced speedup results. Overall, the project demonstrated how the same algorithm behaves differently across parallel models and how hardware shapes performance.

7. Conclusion

The trapezoidal rule parallelizes cleanly across all three models due to independent iteration workloads. OpenMP and MPI provide solid CPU-side acceleration, with MPI scaling better until physical cores saturate. CUDA excels for large problem sizes, delivering extremely fast runtime by leveraging thousands of parallel GPU threads. Across all experiments, the implementations were correct, scalable to their hardware limits, and demonstrated the core principles of parallel programming.

8. References

- OpenAI, “ChatGPT,” 2025 — assistance for explanation, structuring, and code guidance.
- CUDA Toolkit Documentation