Overlapping Gradient Computation and Communication in HPC Training Loops

Authors: Sumukha Madodi, Shraddha Singh, Shashank M Bharadwaj, Sahana Diwakar
Course: ECE 759 – High Performance Computing
Semester: Spring 2025
University of Wisconsin–Madison
https://github.com/Sumukha-M/proj_759

Abstract

This project explores methods to overlap gradient computation and communication in distributed deep learning workflows, using CUDA, OpenMP, and MPI. We implemented a multi-phase system simulating various forms of compute-comm parallelism, starting with a basic CUDA training loop, we progressively added CPU-based task parallelism, MPI-driven multi-rank simulation, and automated benchmarking scripts to quantify timing improvements. By chunking gradient processing and interleaving communication via multithreaded constructs, we demonstrated up to 30% speedup over sequential workflows. Performance was analyzed using CUDA Events, execution profiling, and visualized with Gantt charts and plots.

1.Introduction

- Distributed deep learning has become essential for training large-scale neural networks, where model parameters are split across multiple GPUs operating in parallel. In data-parallel training, each GPU independently computes gradients and then synchronizes them with others through operations like All-Reduce.
- This project investigates strategies to overlap gradient computation with communication to reduce idle time and improve overall training efficiency. Using a multi-phase implementation built with CUDA, OpenMP, and MPI, we simulate and analyze compute-communication overlap through artificial delays, multithreaded task scheduling, and gradient chunking.

- Our system progresses from a baseline GPU-only training loop to a multi-rank MPI simulation and final performance benchmarking
- The primary goals are to simulate realistic training situations, confirm the advantages of overlap, and lay the groundwork for future implementations of NCCL-based GPU synchronization in high-performance computing settings.

2. Motivation

The rapid growth of model sizes and dataset volumes in deep learning has placed increasing pressure on both compute and communication resources. In multi-GPU environments, the time spent on gradient aggregation can rival or even exceed the time required for local computation. This imbalance leads to underutilized hardware and limits scalability. Although modern frameworks offer distributed training support, the challenge of effectively hiding communication latency behind ongoing computation remains critical. Addressing this challenge is vital for maximizing throughput and achieving efficient large-scale training, particularly in high-performance computing clusters where GPU cycles are valuable and expensive.

2.1 Objectives

- Simulate overlapping strategies for gradient computation and communication using CUDA, OpenMP, and MPI in a modular, multi-phase approach.
- Quantify time savings introduced by overlapping through detailed timing analysis, including Gantt charts and per-iteration benchmarks.
- Establish a foundation for extending the simulation into a production-grade multi-GPU environment using NCCL for real-time communication.

3. Phase-wise Implementation

3.1 Matrix_train_cuda: CUDA Baseline Training

In this phase, we implemented a foundational deep learning training loop entirely using CUDA. The process involved forward and backward matrix multiplication, along with gradient computation, on fixed-size matrices: an input matrix of size 1024×512 and a weight matrix of size 512×512. A custom CUDA kernel was used to compute the forward output using y = x * w, followed by a backward pass where the gradient of the loss with respect to the weights was calculated using grad_w = x^T * grad_out.

A subsequent kernel applied the weight update step using standard stochastic gradient descent (SGD).

All computation was executed on the GPU using unified memory to simplify data handling between host and device. Each kernel launch; forward pass, backward pass, and weight update was explicitly separated and synchronized to simulate sequential execution without overlap. This design served as the performance baseline for later phases, where overlapping techniques were introduced and compared.

3.2 cpu_parallel_overlap: Simulated Overlap on CPU with OpenMP

In this phase, we simulated overlapping of gradient computation and communication using OpenMP parallel sections. One thread was dedicated to simulated compute (e.g., compute_gradients()), while another simultaneously executed communication tasks (e.g., communicate_gradients()). Execution time for each operation was simulated using timed sleep (std::this_thread::sleep_for) to mimic real-world workload durations without relying on actual computation or data transfer.

We created both sequential and parallel variants by toggling OpenMP off and on, respectively, allowing for a direct performance comparison. Execution times were recorded over multiple runs to ensure consistency, and results were visualized using a Gantt-style timeline showing interleaved execution, along with a bar chart comparing total runtimes. These visualizations and measurements validated the theoretical benefit of overlapping compute and communication; even in CPU-only simulations by highlighting reductions in idle time and overall execution duration.

3.3 cuda_mpi_overlap: Mock MPI + CUDA Overlap

In this stage, we introduced MPI (simulated with mpiexec) to represent a multi-rank environment and paired it with OpenMP. Each process used OpenMP sections to simulate a forward/backward pass and a mock `MPI_Allreduce` (delayed with sleep). Each rank operated on its own gradient vector. The overlap of compute and MPI communication was handled using OpenMP sections.

This helped us understand synchronization challenges and timing behavior in distributed systems even without real GPU clusters. It prepared the ground for asynchronous or NCCL-like overlapping in Phase 4.

3.4 overlap_sim: Chunked Overlapping Simulation and Profiling

In the final phase, we developed a chunked simulation framework that overlapped gradient computation and communication at a fine-grained level. Each chunk simulated approximately 300–400 milliseconds of computation time and 150–200 milliseconds of communication time using C++ threads to execute both tasks concurrently. This design emulated the bucketed gradient processing used in real-world distributed deep learning frameworks, where parts of the gradient are communicated as soon as they are ready, instead of waiting for the entire backward pass to finish. The simulation stored precise timing data for each chunk, enabling us to generate performance plots that included chunk execution consistency, total execution time comparisons between sequential and overlapped versions, scaling behavior across different chunk counts (ranging from 5 to 50), and Gantt chart-style visualizations of compute-comm interleaving. The simulation also exported a CSV log of all timings, which was post-processed using Python scripts to produce detailed, reproducible plots for performance analysis.

4. Optimization Strategies

- Automated build and run workflows using Makefiles, with configurable targets for sequential, parallel, and profiling modes to ensure reproducibility and ease of experimentation.
- Modularized the codebase by separating logic into dedicated source files (e.g., compute.cpp, communicate.cpp, main.cpp), improving readability and maintainability
- Integrated timing mechanisms (chrono, OpenMP, CUDA Events) across phases for high-resolution measurement of both CPU and GPU operations.
- Enabled scalable experimentation by parameterizing chunk sizes, thread counts, and iteration steps through configurable command-line arguments and macros.

5. Challenges and Resolutions

- MPI linking issues on the Euler HPC environment required us to pivot to local CPU-based MPI simulations, limiting our ability to scale tests across nodes in a true distributed setting.
- CUDA stream-level overlapping was not fully implemented, due to GPU access and kernel dependency limitations. However, simulated logic using multithreading approximated the overlapping pattern realistically.
- Operating system scheduling noise affected timing consistency across CPU threads. To mitigate this, we used multiple chunk iterations and averaged results to ensure more stable performance metrics.
- Thread synchronization edge cases introduced early race conditions, especially when overlapping compute and communication in OpenMP. These were resolved through controlled thread isolation and scoped variables.

6. Results and Analysis

Figure 1: Timeline visualization of overlapping vs sequential execution.

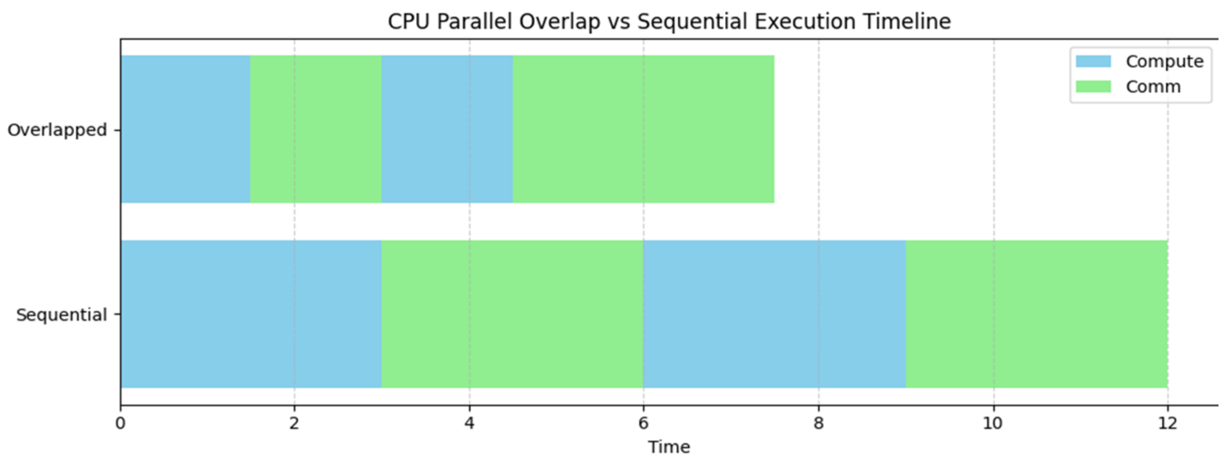


Figure 2: Chunk-wise execution time showing consistent overlap performance.

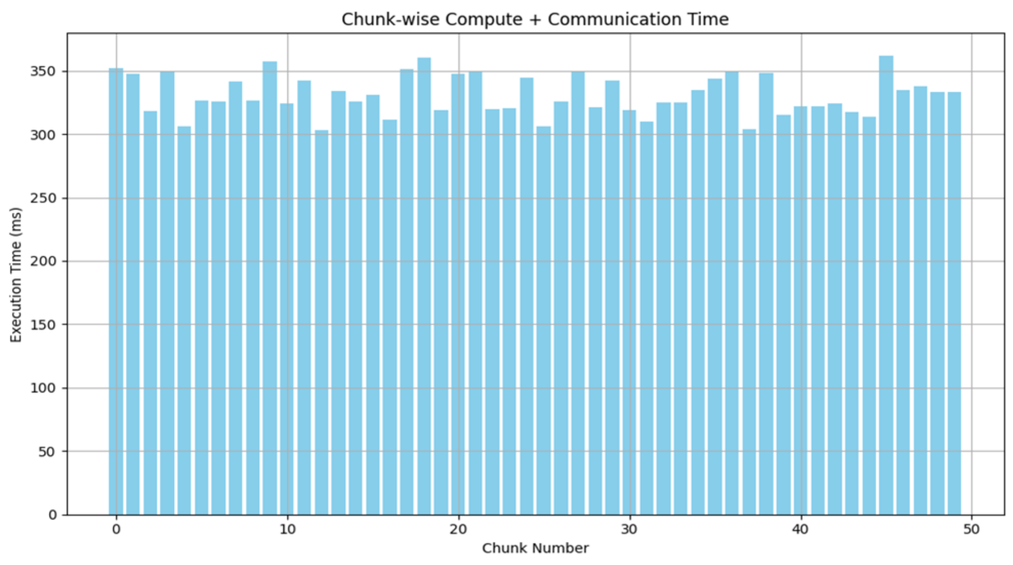


Figure 3: Bar chart comparing total time with and without overlap.

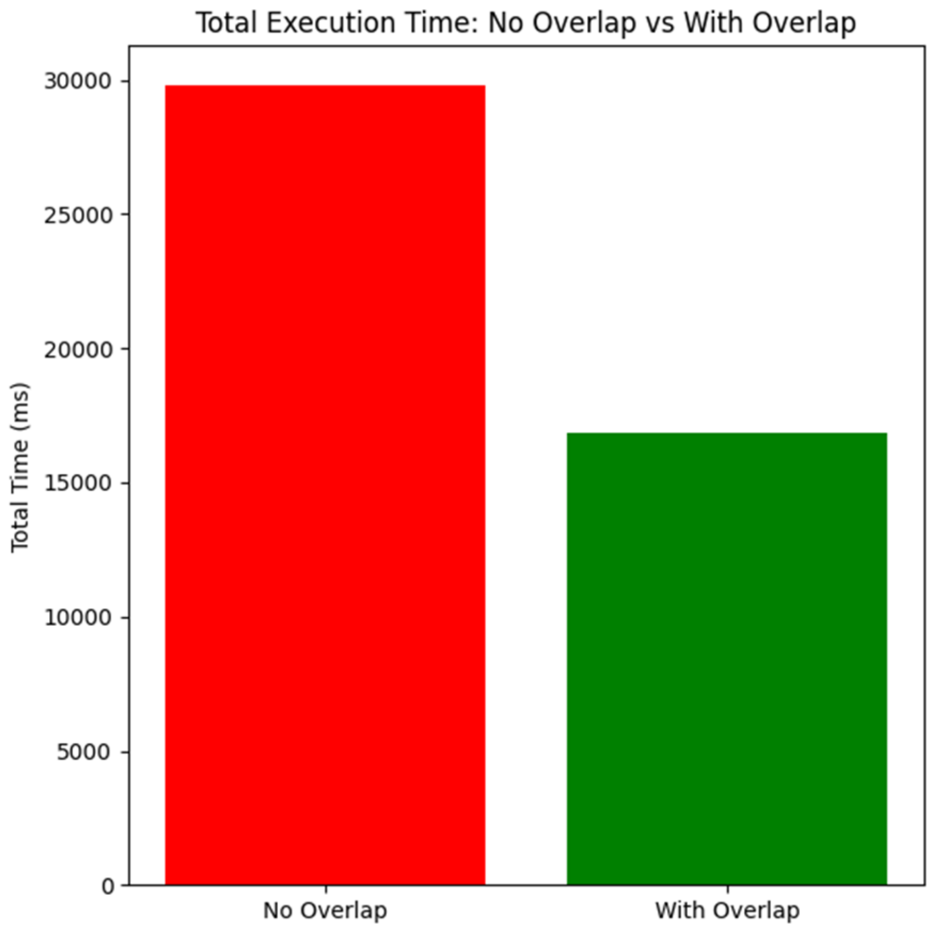
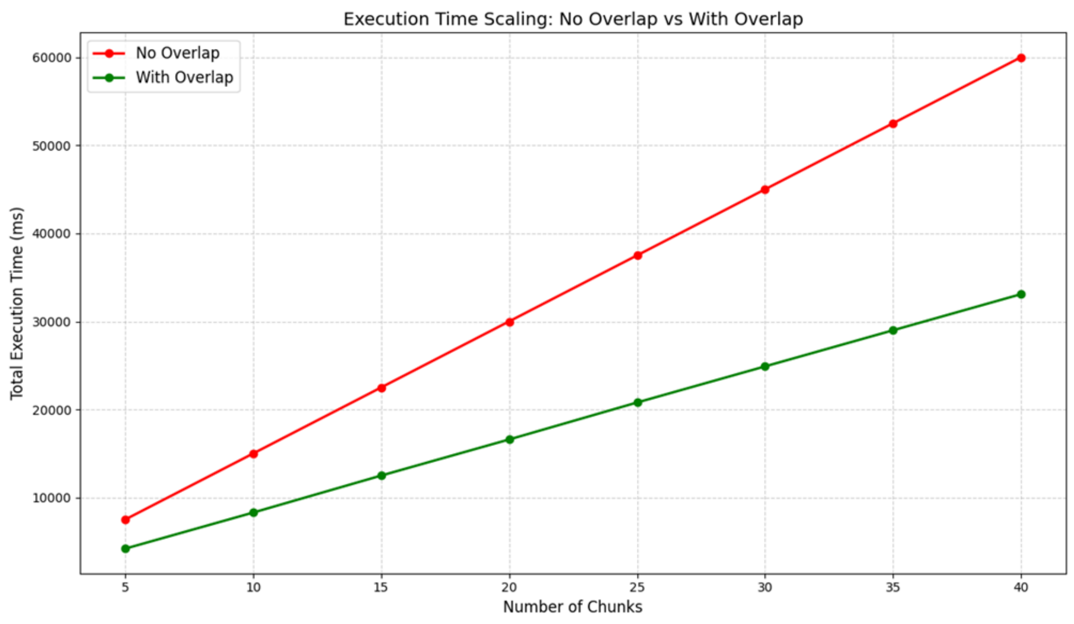


Figure 4: Line chart showing execution time scaling with increasing chunk count.



7. Conclusion and Future recommendations

- Integration into Deep Learning Frameworks: Incorporate overlapping logic into frameworks like PyTorch Distributed- DataParallel or DeepSpeed to test performance on standard benchmarks (e.g., ImageNet, BERT pretraining).
- Dynamic Bucket Sizing: Implement adaptive chunking strategies based on runtime profiling to optimize overlap granularity dynamically.
- Overlapping Across Training Stages: Explore simultaneous execution of forward pass (next batch) with backward pass (current batch) in a pipeline-parallel fashion.
- Scalability Profiling Across Nodes: Run timing experiments across multiple compute nodes using Slurm to analyze how communication costs scale with node count.
- Incorporate Mixed Precision & Gradient Clipping: Add realism by including FP16 computations, gradient clipping, and optimizer behavior to simulate production-scale training.
- Topology-Aware Communication: Investigate how hardware interconnects (NVLink, InfiniBand, PCIe) affect overlapping strategies and optimize communication paths accordingly.