

Parallel 2D Convolution for Deep Learning

Avanindra (Avi) Dubey
Iowa State University

December 2025

Course: CPRE 400/500 Level (High Performance Computing)

Abstract

This project addresses the critical computational bottleneck found in training Convolutional Neural Networks (CNNs), the architecture underpinning modern computer vision AI. The core operation of these networks, 2D convolution, exhibits $O(N^2)$ complexity when executed serially, making the training of large models on single nodes prohibitively slow. To resolve this, I implemented a parallelized convolution algorithm using the Message Passing Interface (MPI) on the Nova High-Performance Computing (HPC) cluster. By employing a “Filter Parallelism” strategy, the workload of 128 filters was distributed across varying processor counts. The implementation was benchmarked on core counts ranging from 1 to 32. The results demonstrated a successful reduction in runtime from 1.83 seconds (1 core) to 0.52 seconds (32 cores), achieving a 3.52x speedup and validating the scalability of distributed memory systems for deep learning tasks.

1 Introduction

1.1 Background

In the field of Artificial Intelligence, and specifically in computer vision, the Convolutional Neural Network (CNN) is the dominant architecture. CNNs are used for tasks ranging from analyzing medical scans to enabling autonomous vehicles to “see” the road. Unlike standard neural networks, which treat inputs as flat arrays of numbers, CNNs preserve the spatial relationship of pixels in an image.

The fundamental mathematical operation driving these networks is the 2D convolution. This process involves a small matrix of learnable weights, known as a “filter” or “kernel,” sliding over a larger input image. At every position, the filter performs element-wise multiplication with the underlying image pixels and sums the result (a dot product) to produce a single value in a new output matrix called a “feature map.”

1.2 The Problem: Computational Bottlenecks

While conceptually simple, the convolution operation is computationally expensive. A single layer in a modern deep learning model does not use just one filter; it often applies thousands of different filters to a high-resolution image to extract various features like edges, textures, or shapes.

When executed on a standard CPU, these operations are performed sequentially. The processor must iterate through every filter, and for each filter, iterate through every pixel of the input image. This results in a massive number of nested loops. Consequently, training modern AI models becomes a time-intensive task, often creating a bottleneck where the speed of innovation is limited by the speed of computation. This project aims to alleviate this bottleneck by moving from serial execution to a parallel computing paradigm using High-Performance Computing (HPC) resources.

2 Methodology

2.1 Strategic Approach: Filter Parallelism

To parallelize the convolution operation, I chose a strategy known as “Filter Parallelism.” In parallel image processing, a common approach is to split the image itself into smaller blocks. However, this method introduces complex boundary issues (halo exchange) where processors must communicate to share pixels at the edges of their assigned blocks.

Filter Parallelism avoids this overhead. Instead of splitting the image, I distributed the filters among the available processors. Since each filter operates independently of the others, the processors do not need to communicate during the actual computation phase. Each processor receives the full input image and a unique subset of filters to process. This “embarrassingly parallel” workload distribution maximizes the time the CPUs spend computing rather than communicating.

2.2 Implementation Environment

The solution was implemented in C++ to prioritize raw execution speed, as interpreted languages like Python often introduce latency unsuitable for low-level performance benchmarking. I utilized the MPI (Message Passing Interface) library for parallelization. MPI was selected over OpenMP (shared memory) because deep learning datasets are often too large to fit into the RAM of a single node; MPI allows the program to leverage the distributed memory and aggregate processing power of the entire Nova cluster.

2.3 Algorithm Details

The parallel algorithm proceeds in three distinct phases: Distribute, Compute, and Collect.

Phase 1: Initialization and Distribution The root process (Rank 0) initializes the data structures. It generates a random 1024×1024 input image and a set of 128 random 5×5 filters to simulate a real-world workload.

- **Broadcasting the Image:** The root process uses `MPI_Bcast` to send the complete image vector to all worker processes. This ensures every core has the necessary context to perform convolutions.
- **Scattering the Filters:** To divide the workload, I used `MPI_Scatterv` (Scatter Vector). This was necessary because the number of filters (128) might not always be perfectly divisible by the number of processors. The code calculates counts (how many filters each rank gets) and displacements to ensure an even load balance, handling remainders gracefully.

Phase 2: Computation Once the data is distributed, each worker process executes the convolution logic on its local subset of filters. I utilized flat 1D `std::vector` arrays to store the 2D data (image and filters). Accessing 2D data via 1D mapping (e.g., `index = y * width + x`) improves cache locality and memory throughput compared to pointer-heavy 2D arrays.

Mathematically, the discrete 2D convolution operation performed by the algorithm is defined as:

$$S(i, j) = (I * K)(i, j) = \sum_{m=0}^{H_f-1} \sum_{n=0}^{W_f-1} I(i+m, j+n) \cdot K(m, n) \quad (1)$$

Where:

- $S(i, j)$ is the value of the pixel at position (i, j) in the output feature map.
- I represents the input image matrix.
- K represents the filter (kernel) of dimensions $H_f \times W_f$.
- $(i+m, j+n)$ represents the spatial sliding of the window over the input.

Phase 3: Collection After computation, the partial results (feature maps) stored in each processor’s local memory must be aggregated. The root process uses `MPI_Gatherv` to collect the output buffers from all workers and reassemble them into the final output tensor. This is the only synchronization point required after the initial broadcast.

2.4 Verification Strategy

Speed is irrelevant without accuracy. To ensure the parallel implementation produced correct results, I implemented a verification system. The root process runs a serial version of the convolution (sliding every filter one by one) to establish a “ground truth” baseline. The final parallel output is compared against this serial baseline element by element. Due to the nature of floating-point arithmetic, I implemented an epsilon check (tolerance of $1e-6$) to account for minute precision differences.

3 Experimental Results

The implementation was tested on the Nova HPC cluster. I conducted benchmarks using varying numbers of cores ($p = 1, 4, 16, 32$) to analyze how performance scaled with additional hardware resources.

3.1 Performance Data

The following table summarizes the runtime results for processing the 1024×1024 image with 128 filters (5×5):

Number of Cores	Parallel Runtime (seconds)	Speedup (vs. 1 Core)
1	1.83	1.0x (Baseline)
4	1.03	1.77x
16	0.74	2.48x
32	0.52	3.52x

Table 1: Runtime Analysis on Nova Cluster

3.2 Accuracy Verification

For every test run, from 1 core up to 32 cores, the verification step returned “SUCCESS,” confirming that the parallel output matched the serial baseline exactly within the specified tolerance. This proves that the decomposition of the problem via Filter Parallelism did not compromise the mathematical integrity of the convolution.

4 Discussion and Analysis

4.1 Speedup and Scalability

The data shows a significant improvement in performance as the core count increases. The execution time dropped from 1.83 seconds on a single core to 0.52 seconds on 32 cores, achieving a 3.52x speedup. This reduction confirms that distributing the filters is an effective way to accelerate deep learning training tasks.

4.2 Amdahl’s Law and Diminishing Returns

While the speedup is substantial, the scaling is not linear. Theoretically, utilizing 32 cores should yield a speedup closer to 32x. The observed speedup of 3.52x illustrates Amdahl’s Law, which states that the maximum speedup of a program is limited by its sequential components. In this application, the “sequential” bottlenecks include:

- **Communication Overhead:** The time required to `MPI_Bcast` the large image and `MPI_Gatherv` the results increases slightly or remains constant as cores are added,

while the computation time per core decreases drastically. At 32 cores, the time spent communicating becomes a larger percentage of the total runtime compared to the time spent computing.

- **Memory Bandwidth:** Even with distributed memory, moving large amounts of data (the image and result tensors) across the cluster interconnect introduces latency that pure computation does not.

The graph of Runtime vs. Cores visually demonstrates this phenomenon: there is a steep drop in execution time when moving from 1 to 4 cores, but the curve flattens significantly between 16 and 32 cores. This suggests that for this specific image size and filter count, we are approaching the point of diminishing returns where adding more cores provides minimal benefit due to the fixed costs of MPI communication.

4.3 Overhead Analysis

Interestingly, the 1-core parallel run (1.83s) was slightly slower than the pure serial baseline (1.34s). This difference represents the initial overhead of initializing the MPI environment and setting up buffers. However, this initial “cost” is quickly outweighed by the parallel gains once the core count increases to 4 and beyond.

5 Conclusion

This project successfully designed, implemented, and verified a Parallel 2D Convolution algorithm using MPI. By leveraging Filter Parallelism, I effectively distributed the heavy computational workload of a CNN layer across the Nova HPC cluster.

The results validated the hypothesis: increasing the number of processors significantly reduces execution time. While communication overhead prevents perfect linear scaling, the reduction in runtime from 1.83 seconds to 0.52 seconds represents a major efficiency gain. This project demonstrates that distributed memory systems are essential for modern AI, allowing for the training of complex models that would otherwise be impossible on serial hardware. Future work could focus on overlapping communication with computation (using non-blocking MPI calls) to further push the limits of Amdahl’s Law.

References

Song Ho Ahn, “Example of 2D Convolution,” Song Ho Ahn’s Home.

Lawrence Livermore National Laboratory, “MPI Tutorial,” LLNL HPC Training.

GeeksforGeeks, “Implementation of 2D Convolution.”