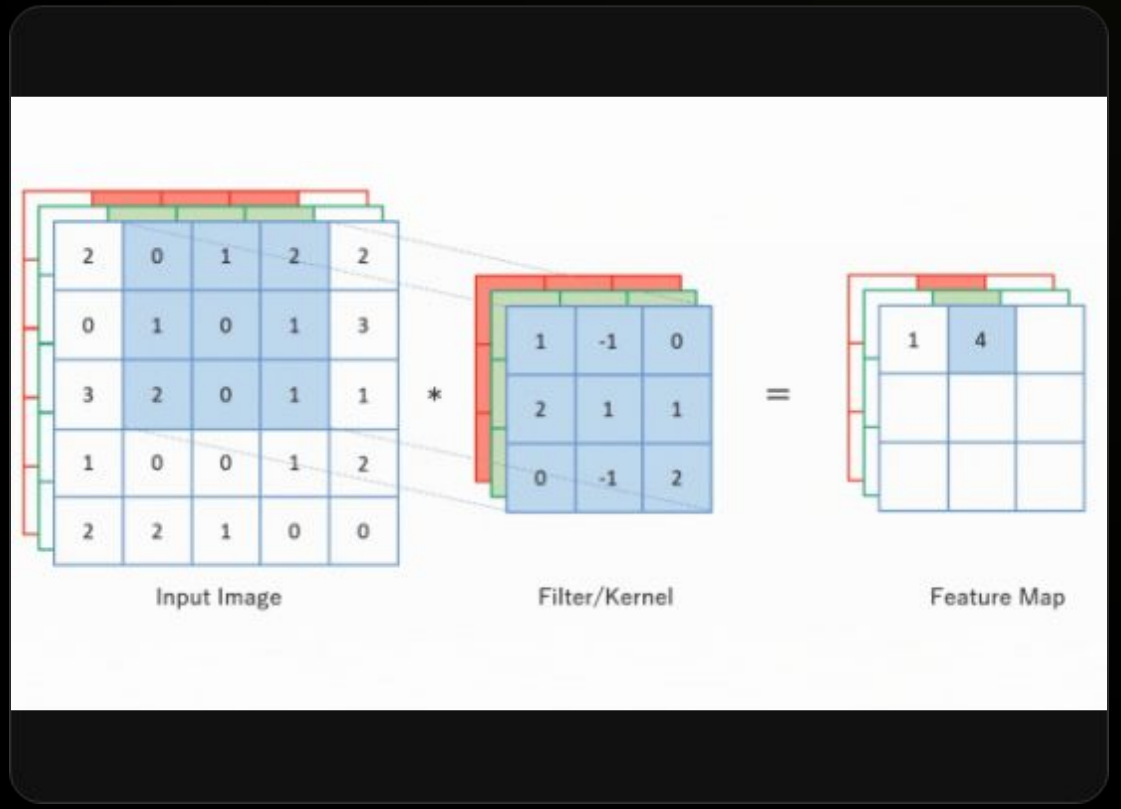
# Parallel 2D Convolution for Deep Learning

High-Performance Computing using MPI

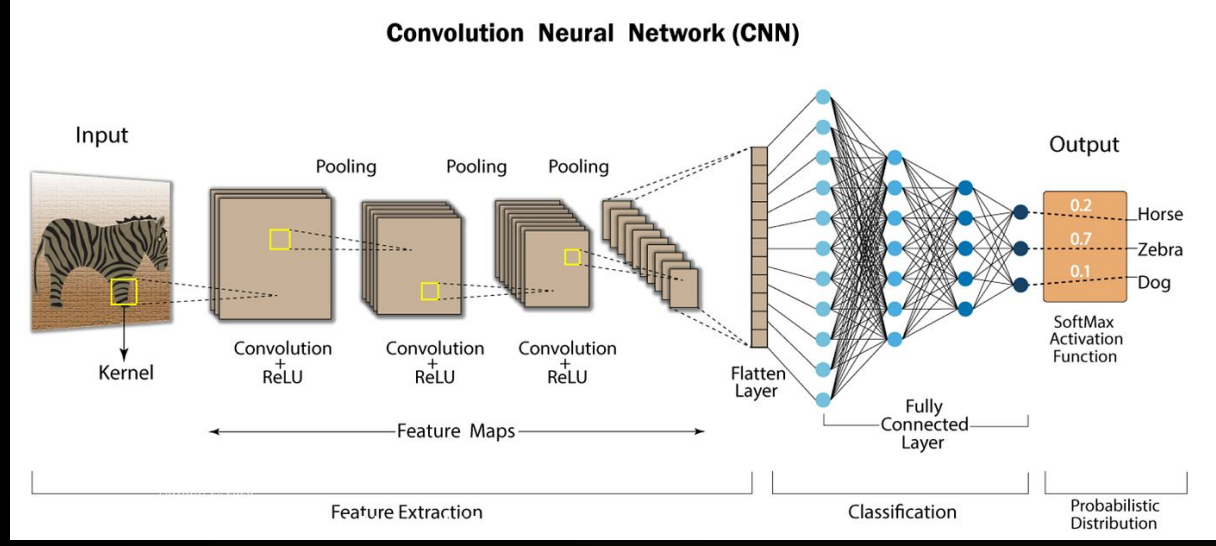
By: **Avi Dubey**

# The Computational Bottleneck

- **Massive Workload:** Convolutional Neural Networks (CNNs) rely on sliding thousands of filters over high-resolution images.
- **Serial Latency:** Standard CPU execution performs these operations sequentially, leading to $O(N^2)$ complexity.
- **Training Delay:** This bottleneck makes training large modern AI models prohibitively slow on single nodes.



# What I did :)




The Bottleneck: Training modern AI models using CNNs is slow because the math operation behind it is a 2D Convolution which takes massive computing power and runs often O(N^2) time, creating a major time bottleneck

The Solution: I implemented a Filter Parallelism using MPI message passing interface to divide the workload across multiple computers in the hpc cluster instead of relying on a single processor to do the task

The Strategy: Instead of splitting the image into parts, I distributed the thousands of *filters* so each processor works separately which eliminated more communication overhead during the computing task

The Result: By utilizing my strategy and solution I achieved a 3.5x speedup on the Nova cluster. THis proves that this method effectively scales deep learning tasks across distributed memory systems

# MPI Strategy: Filter Parallelism

### 1. Distribute

The root process uses **MPI Bcast** to send the full image and **MPI Scatterv** to divide the 128 filters among workers

### 2. Compute

Each worker than computes the convolution on its own subset of filters separately

### 3. Collect

Root process uses **MPI Gatherv** to assemble the resulting feature maps into the final output tensor

# Implementation Details

**Language:** C++ (Compiled with mpic++)

**Optimization:** I used flat 1D arrays to maximize the cache locality and memory throughout

**Verification:** I implemented a verification step that compares the parallel outputs that I run against a serial baseline to ensure that it runs accurately

**Environment: I** tested and benchmarked it on the nova HPC cluster like we have done in class.

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <cmath>
#include <random>
#include <mpi.h>

//differnt sizes to test with
const int IMAGE_HEIGHT = 1024;
const int IMAGE_WIDTH = 1024;
//the total number of filters
const int K_FILTERS = 128;
const int FILTER_HEIGHT = 5;
const int FILTER_WIDTH = 5;

//this calculates the sizes output sizes along with the image and filter size
const int OUTPUT_HEIGHT = IMAGE_HEIGHT - FILTER_HEIGHT + 1;
const int OUTPUT_WIDTH = IMAGE_WIDTH - FILTER_WIDTH + 1;
const int IMAGE_SIZE = IMAGE_HEIGHT * IMAGE_WIDTH;
const int FILTER_SIZE = FILTER_HEIGHT * FILTER_WIDTH;
const int OUTPUT_MAP_SIZE = OUTPUT_HEIGHT * OUTPUT_WIDTH;


//this performs a 2D convolution on one image with one filter. All data is in flat 1D vectors for the performance

std::vector<double> serial_convolution_single(const std::vector<double>& image,
                                              const std::vector<double>& filter)
{
    std::vector<double> output_map(OUTPUT_MAP_SIZE);

    //This loops over output map rows
    for (int y = 0; y < OUTPUT_HEIGHT; ++y) {
        //This loops over output map cols
        for (int x = 0; x < OUTPUT_WIDTH; ++x) {

            double sum = 0.0;
            //This loops over filter rows
            for (int j = 0; j < FILTER_HEIGHT; ++j) {
                //This loops over filter cols
                for (int i = 0; i < FILTER_WIDTH; ++i) {
```

```cpp
    //this stops the timer and verify
    MPI_Barrier(MPI_COMM_WORLD);
    double p_end = MPI_Wtime();

    if (world_rank == 0) {
        double parallel_time = p_end - p_start;
        double serial_time = std::chrono::duration<double>(
            std::chrono::high_resolution_clock::now() -
            std::chrono::high_resolution_clock::now()
        ).count();

        //this gets serial time
        auto start = std::chrono::high_resolution_clock::now();
    }

    //printing out for the terminal for verfication
    if (world_rank == 0) {
        double parallel_time = p_end - p_start;
        std::cout << "\nParallel Run Time: " << parallel_time << " seconds." << std::endl;

        std::cout << "\nVerifying results..." << std::endl;
        if (verify(serial_output, parallel_output)) {
            std::cout << "Verification: SUCCESS! Serial and Parallel outputs match." << std::endl;
        } else {
            std::cout << "Verification: FAILED! Outputs do not match." << std::endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```

# Code Snippets

I implement the 2D convolution algorithm in which I use nested loops to slide a single filter over the image and calculate the dot product for each pixel

Than I establish a performance benchmark by running all convolutions sequentially on the root processor and timing the process to see how long it takes without parallelization

Than I sets up the parallel environment by broadcasting the image data to all cores ensuring every worker has the necessary input to perform its task

I also calculate how to divide the filters among the available cores load balancing determining exactly which chunk of filters each specific processor will be responsible for computing

```
133     //this is parallel execution
134     //this starts timer for parallel run
135     MPI_Barrier(MPI_COMM_WORLD);
136     double p_start = MPI_Wtime();
137
138     //this is the broadcast image and the non root process has to allocate the space before the broadcast
139     if (world_rank != 0) {
140         image.resize(IMAGE_SIZE);
141     }
142     MPI_Bcast(image.data(), IMAGE_SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
143
144     //this is how many filters each rank gets
145     std::vector<int> counts(world_size);
146     std::vector<int> displs(world_size);
147
148     int base_chunk = K_FILTERS / world_size;
149     int remainder = K_FILTERS % world_size;
150     int current_displ = 0;
151
152     for (int i = 0; i < world_size; ++i) {
153         counts[i] = base_chunk + (i < remainder ? 1 : 0);
154         displs[i] = current_displ;
155         current_displ += counts[i];
156     }
157
158     //each of the ranks needs to know their local number of filters
159     int local_k = counts[world_rank];
160
161     std::vector<int> counts_bytes(world_size);
162     std::vector<int> displs_bytes(world_size);
163
164     //for scattering the filters
165     for (int i = 0; i < world_size; ++i) {
166         counts_bytes[i] = counts[i] * FILTER_SIZE;
167         displs_bytes[i] = displs[i] * FILTER_SIZE;
168     }
169
```

```
106     //this is serial baseline so it runs on the root
107     if (world_rank == 0) {
108         std::cout << "\nRunning serial baseline on root..." << std::endl;
109         auto start = std::chrono::high_resolution_clock::now();
110
111         std::vector<double> current_filter(FILTER_SIZE);
112
113         for (int k = 0; k < K_FILTERS; ++k) {
114             //this copies the k th filter from the flat array
115             std::copy(all_filters.begin() + k * FILTER_SIZE,
116                       all_filters.begin() + (k + 1) * FILTER_SIZE,
117                       current_filter.begin());
118
119             //this runs convolution for this single filter
120             std::vector<double> one_map = serial_convolution_single(image, current_filter);
121
122             //this cpies the result into the full serial output tensor
123             std::copy(one_map.begin(), one_map.end(),
124                       serial_output.begin() + k * OUTPUT_MAP_SIZE);
125         }
126
127         auto end = std::chrono::high_resolution_clock::now();
128         std::chrono::duration<double> diff = end - start;
129         std::cout << "Serial Baseline Time: " << diff.count() << " seconds." << std::endl;
130     }
131
```

# Performance Results

```
--- Parallel 2D Convolution ---
Running with 16 processes.
Image: 1024x1024
Filters: 128 (5x5)

Running serial baseline on root...
Serial Baseline Time: 2.31388 seconds.

Parallel Run Time: 0.739325 seconds.

Verifying results...
Verification: SUCCESS! Serial and Parallel outputs match.
```

```
--- Parallel 2D Convolution ---
Running with 32 processes.
Image: 1024x1024
Filters: 128 (5x5)

Running serial baseline on root...
Serial Baseline Time: 1.57537 seconds.

Parallel Run Time: 0.520622 seconds.

Verifying results...
Verification: SUCCESS! Serial and Parallel outputs match.
```

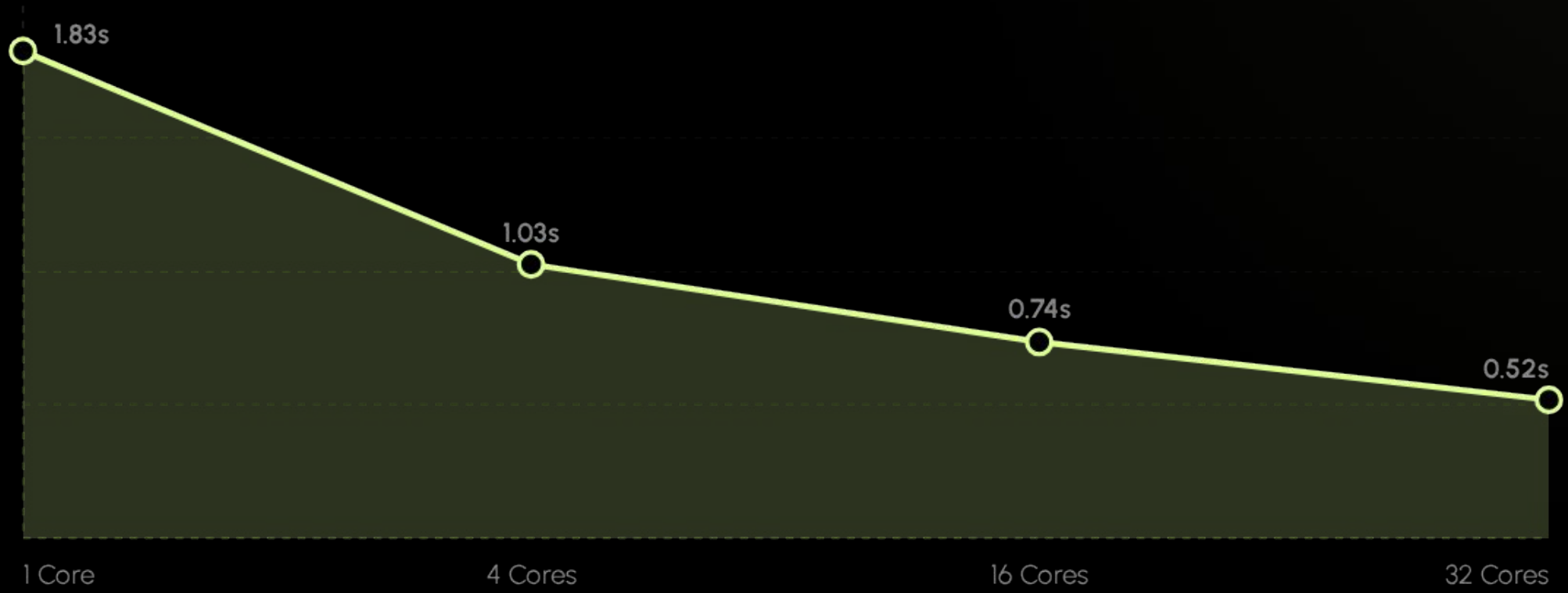
```
--- Parallel 2D Convolution ---
Running with 4 processes.
Image: 1024x1024
Filters: 128 (5x5)

Running serial baseline on root...
Serial Baseline Time: 1.93069 seconds.

Parallel Run Time: 1.03223 seconds.
```

```
--- Parallel 2D Convolution ---
Running with 1 processes.
Image: 1024x1024
Filters: 128 (5x5)

Running serial baseline on root...
Serial Baseline Time: 1.34274 seconds.

Parallel Run Time: 1.83391 seconds.
```

- **Speedup:** The parallel implementation achieved a **3.5x speedup**, reducing the runtime from 1.83 seconds on a single core to just 0.52 seconds on 32 cores.
- **Scalability:** The performance graph shows a clear downward trend in execution time as the number of cores increases from 1 → 4 → 16 → 32, proving that the Filter Parallelism strategy effectively scales on the Nova cluster.
- **Overhead Analysis:** While the 1-core parallel run of 1.83 s was slightly slower than the serial baseline of 1.34s due to initial MPI communication overhead, this cost was quickly outweighed by the speed gains at 4 cores and beyond.
- **Accuracy:** Every parallel run that i did from 1 to 32 cores passed the verification step, producing results identical to the serial baseline within a certain tolerance to account for real world discrepancies.

# Performance Results

As we can see the performance does a plateau which illustrates Amdahl's Law. In which the maximum speedup is constricted by the application's serial components and communication overhead. As core counts increase the non parallel bottlenecks prevent a linear scaling in performance which results in diminishing returns despite the additional hardware




*Achieved a **3.5x speedup**, reducing runtime from 1.83s to 0.52s by utilizing 32 cores.*

# Final Understanding

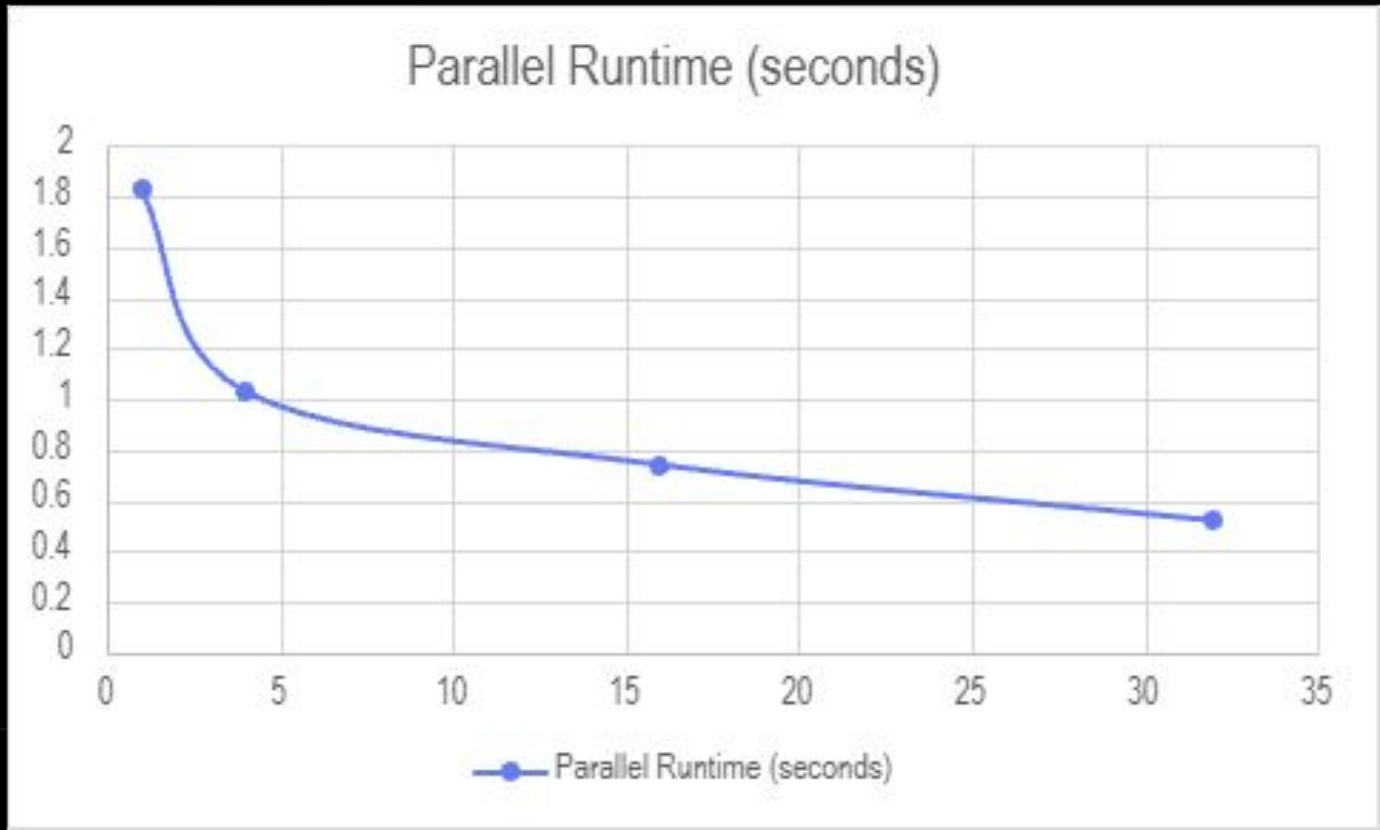
The data shows that as you increase the number of processor cores from 1 up to 32 the time it takes to complete the 2D Convolution task decreases significantly

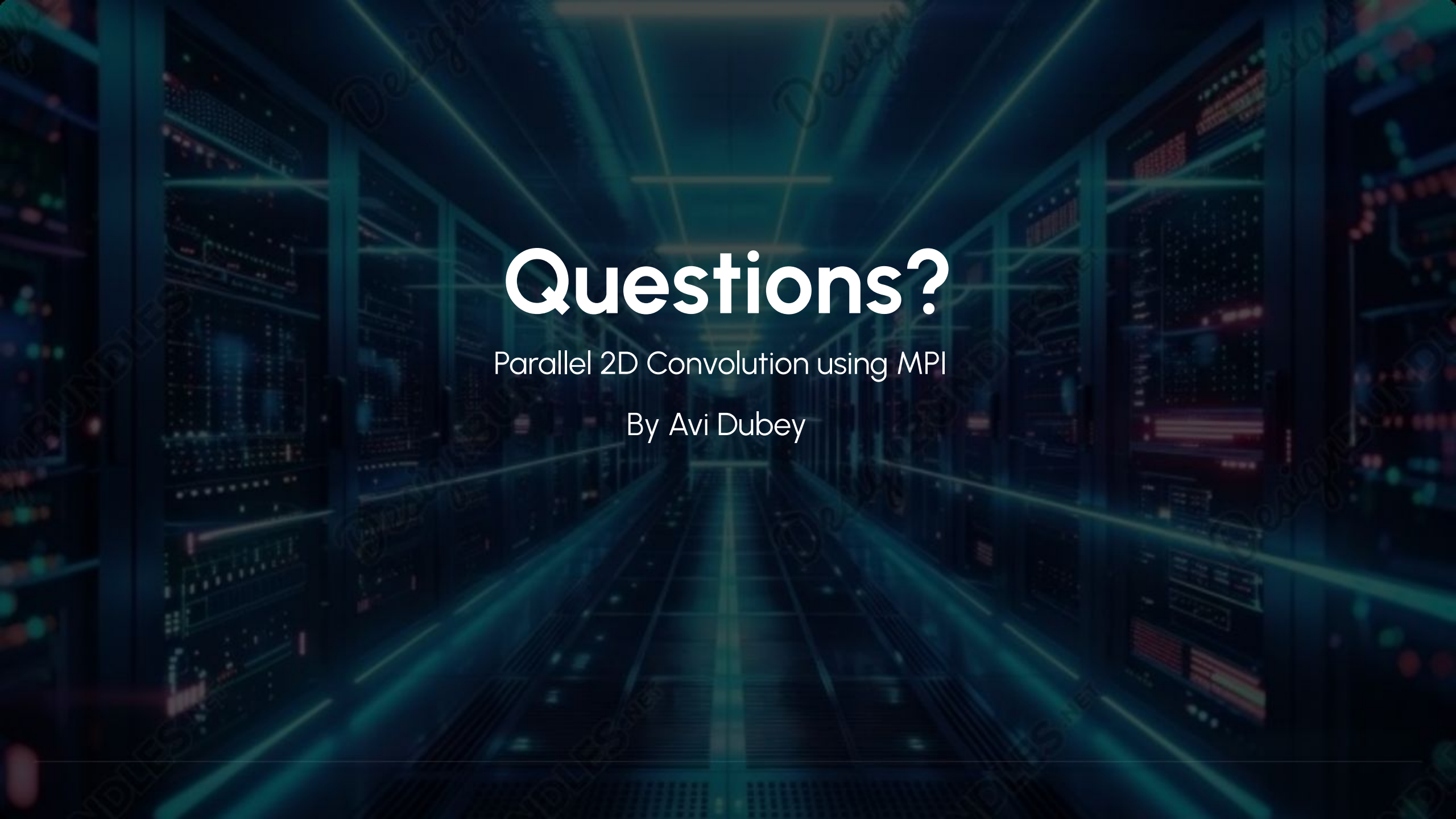
The runtime drops from **1.83 seconds** using a single core down to **0.52 seconds** when using 32 cores.

While the process gets faster, it doesn't scale perfectly. Using 32 times the hardware (32 cores) results in a **3.52x speedup**, rather than being 32 times faster, likely due to the overhead of managing parallel tasks.

**Diminishing Returns:** The graph shows a steep drop in time initially between 1 and 4 cores, but the curve flattens out as you add more cores, meaning the efficiency gains become smaller at higher

| Number of Cores | Parallel Runtime (seconds) | Speedup from Core 1 |
|---|---|---|
| 1 | 1.83 | 1x |
| 4 | 1.03 | 1.77x |
| 16 | 0.74 | 2.48x |
| 32 | 0.52 | 3.52x |




Questions?
Parallel 2D Convolution using MPI
By Avi Dubey