Implementation Report on Distributed CNN using OpenMP and MPI

Nahom Senay GSR/4848/17

Submitted to: Beakal Gizachew Assefa (PhD)
Github Link: https://github.com/Nahom32/Distributed-CNN

Addis Ababa University
January 2026

1 Introduction

Deep learning architectures and algorithms are known to be resource intensive. The resource in this context implies both data and compute. It has been estimated that a deep learning model operates optimally if one has 5000 data instances for every feature [1] .

This has created a need for leveraging distribution and parallelization for training these models [2]. For instance, Alexnet [3] used parallel programming to distribute the convolution and training computation on separate GPUs. There capacity to use parallelized GPU programming enabled them to beat the ILSVRC benchmark of 2012. Furthermore, it showed that neural networks are scalable as long as data and compute resources are provided.

Since the early 2010s, state-of-the-art hardware has been ripe enough to support the compute and memory requirements of deep learning models. This has made them the go-to approach to Artificial Intelligence.

2 Model Selection and Serial Baseline

2.1 Dataset Selection

The implemented model is trained on the MNIST dataset [4]. This dataset contains 60,000 handwritten digits images from 0-9.

2.2 Model Design

The model selected in this parallelization experiment is the Convolutional Neural Network (CNN). The input layer expects a grayscale image of 28×28 pixels.

The convolution layer has a kernel size of 33. The convolution layer contains eight filters. The output layer of the convolution layer is $28 \times 28 \times 8$ with 8 separate feature maps.

The activation function used in the internal layers is the ReLU (Rectified Linear Unit).

$$f(x) = max(0, x)$$

The softmax function has been used in the output layer to identify which number has the highest probability.

$$f(x) = \frac{exp(z_i)}{\sum_i^n exp(z_i)}$$

Max Pooling was used for the pooling layer. The window size of the max pool layer was 2×2. The pooling layer had an output dimension of $13 \times 13 \times 8$. Before entering the fully-connected layer, A flattening layer was implemented that converts the tensor to a 1D matrix.

The fully connected layer is a multi-layered perceptron with a softmax output that can give a probabilistic rank for [0-9] with values $0 \leq x \leq 1$. The following is a table summarizing the overall cnn architecture.

The loss calculation used for this experiment is the **cross-entropy loss** given by the following equation:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{1}$$

Table 1: Summary of Tensor Transformations in the CNN Architecture

Layer Type	Input Dimension	Operation Details	Output Dimens
Input	$28 \times 28 \times 1$	Raw Grayscale Image	-
Convolution (Conv2D)	$28 \times 28 \times 1$	8 Filters, 3×3 Kernel, Stride 1	$26 \times 26 \times 8$
Activation (ReLU)	$26 \times 26 \times 8$	Element-wise $\max(0, x)$	$26 \times 26 \times 8$
Max Pooling	$26 \times 26 \times 8$	2×2 Kernel, Stride 2	$13 \times 13 \times 8$
Flatten	$13 \times 13 \times 8$	Reshape 3D Volume to 1D Vector	1352
Fully Connected (FC)	1352	Matrix Mult. (10×1352) + Bias	10

3 Parallelization Strategies

3.1 Data Parallelism

In this experiment, the parallelization strategies followed the hybrid parallelism model. It combines both Distributed Data Parallelism via MPI with Shared-Memory Parallelism of OpenMP. The MPI acts as a manager of different independent processes with sharded data. The data set at hand is divided into four separate processes in equal measure. This means that each process will get 15,000 shards of the dataset. The rationale of using this approach is to utilize the available cores and have a better load balancing between the cores of the system.

Every MPI rank initializes the parameters of the Neural Network (weights W and Biases b) on its memory space. Then each rank processes the batch it sharded and computes the local weight separately. Before updating the weights, all the processes are paused via a barrier. Then the gradients are summed from all the ranks on a master core and sent to the processes. The algorithm used is for the gradient optimization is the stochastic gradient descent algorithm distributed over the nodes.

3.2 Task Parallelism

Inside every computing element (node), OpenMP has been used to parallelize the heavy computation involved for the convolution operation and the max pooling operation. This is made to leverage the shared memory of a single process. This helps to utilize the Arthimetic and Logic Unit of the CPU cores.

```
#pragma omp parallel for collapse(2)
for (int b = 0; b < batch_size; ++b) {       // Split Batch
    for (int o = 0; o < out_channels; ++o) { // Split Filters
        // ... Thread does the heavy lifting here ...
    }
```

```
}
```

Listing 1: OpenMP Parallelization Strategy using Loop Collapsing

4 Experimental Setup

The experiments were conducted on a single node equipped with an **m4 macbook air** device. The compiler used for the experiment was Apple Clang 16.0.0, with OpenMPI 5.0. The visualization experiments where performed using python matplotlib and the subprocess library to run the cnn training binary. In addition to these, scaling experiments have been implemented as the number cores involved increases.

The hyperparameters used for the experiment were described as follows:

Table 2: Experimental Setup and System Specifications

Component	**Specification**
Hardware	Apple M4 Processor (SoC)
Memory	Unified Memory Architecture (UMA)
OS	macOS Sequoia
Compiler	Apple Clang 16.0.0 (-O3 optimization)
MPI	OpenMPI v5.0
OpenMP	LLVM libomp
Dataset	MNIST (60k Train / 10k Test)
Batch Size	32 (Per Rank)
Learning Rate	0.01

5 Performance Evaluation and Comparison

The performance of the parallelized with 4 ranks were trained on 5 epochs had the following results: The following is the implementation of the serial cnn training with

Table 3: Training Metrics over 5 Epochs (Distributed CNN)

Epoch	**Training Loss**	**Test Accuracy (%)**
1	1.0758	88.19
2	0.3996	90.30
3	0.3483	90.90
4	0.3262	91.29
5	0.3118	91.55
Total Training Time: 45.97 seconds		

no MPI sharding. The following is a comparison is between the serial and distributed training. It has been shown that the serial code has better performed in terms of accuracy.

Table 4: Performance Comparison: Serial vs. Distributed (4 Ranks)

Execution Mode	**Processors**	**Total Time (s)**	**Speedup Factor**
Serial Baseline	1	163.60	1.00x
Distributed (MPI+OpenMP)	4	45.97	**3.56x**

Table 5: Training Convergence Comparison (Serial vs. Distributed)

	Serial Training		**Distributed Training**	
Epoch	**Loss**	**Accuracy (%)**	**Loss**	**Accuracy (%)**
1	0.5432	90.52	1.0758	88.19
2	0.3061	91.50	0.3996	90.30
3	0.2737	92.46	0.3483	90.90
4	0.2452	93.21	0.3262	91.29
5	0.2174	93.97	0.3118	91.55

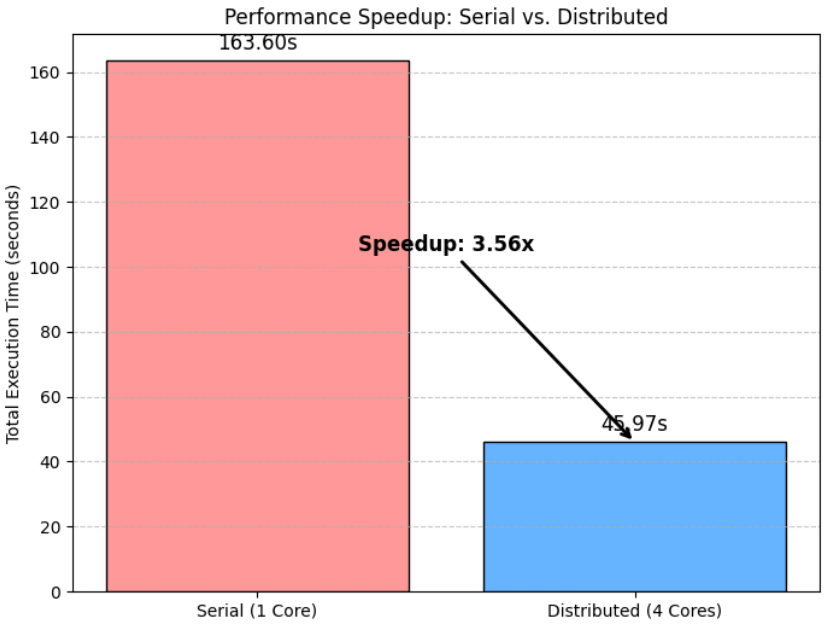

Figure 1: Speedup of Distributed Training

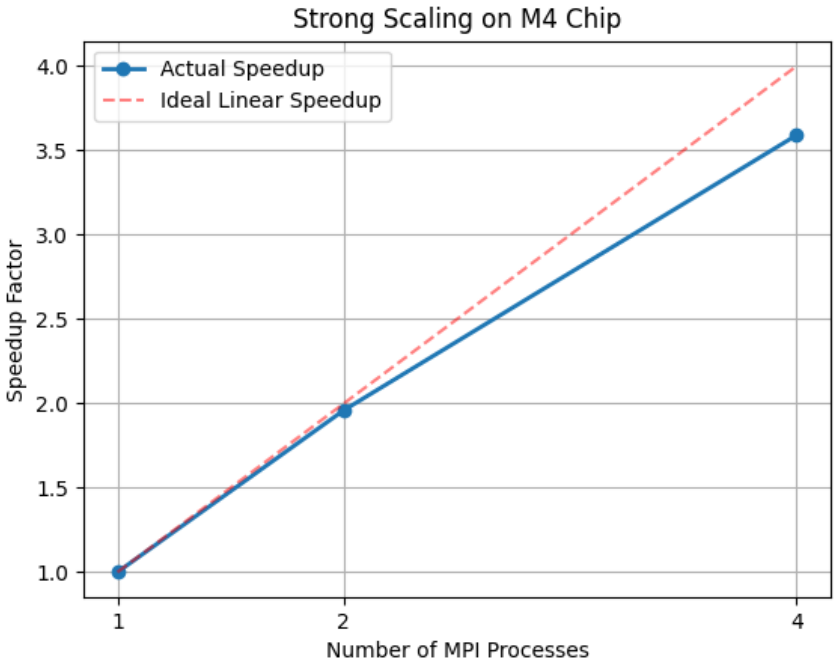

Figure 2: fig: Analysis of the number of cores

The following result is the speed up as the number of cores increases The following is the training loss and accuracy curve for the distributed processing CNN algorithm.

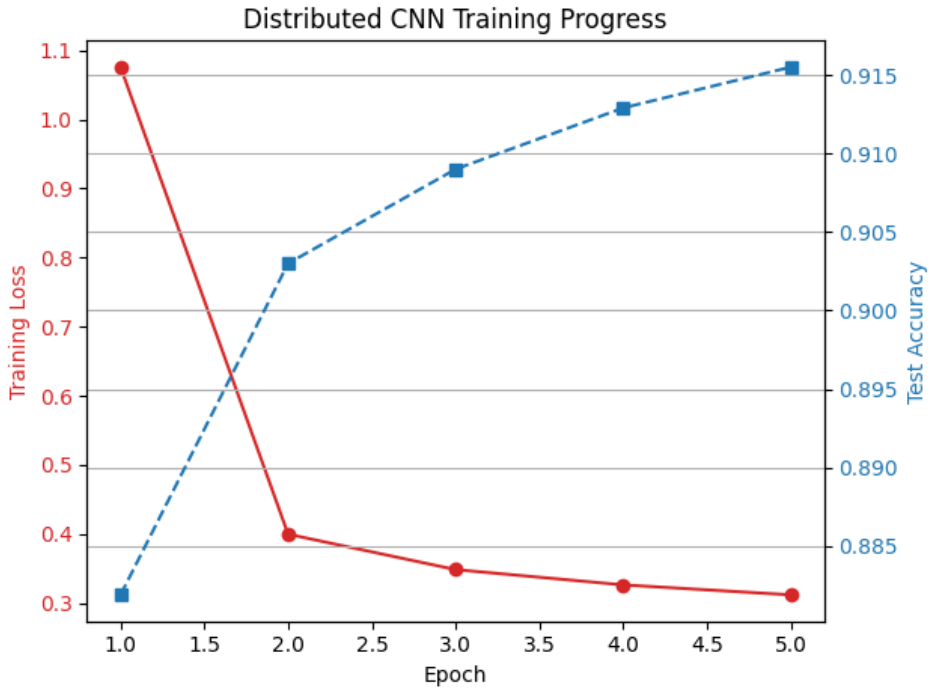

Figure 3: fig: Distributed CNN Training

6 Discussion

It has been shown that the distributed code has 3.56 times faster than the serial code. This is because of the load has been distributed to the different cores available. Furthermore it has been shown that as the number of mpi cores increase, the speedup will scale faster. But it has been shown that it didn't reach the perfect speed up line. This can be attributed to the overhead of spawning different threads has additional overhead on the system.

The serial code has shown to be slightly better in accuracy this can be attributed to the less updates done on each core which will slightly affect the performance of the algorithm. This is because we are using the **stochastic gradient descent** algorithm.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[2] M. Langer, Z. He, W. Rahayu, and Y. Xue, "Distributed training of deep learning models: A taxonomic perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2802–2818, 2020.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, 2012. [Online]. Available: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.