Parallelizing Plant Health: A Comparative Study of Parallel Deep Learning for Plant Disease Classification

Team 13 – He Zhou, Weiyu Wang

1 Introduction

1.1 Background

Plant diseases pose a continuous threat to agricultural productivity, causing economic losses of approximately $220 billion annually, as reported by the Food and Agriculture Organization of the United Nations [1]. These losses disproportionately affect smallholder farmers in underdeveloped regions, sustaining food insecurity and economic hardships. Early disease detection is crucial, but it is often hindered by the shortage of expert opinions, resulting in delays and extensive crop damage.

In recent years, the integration of technologies, especially deep learning and computer vision, has revolutionized the field of plant disease detection. While conventional methods relied on hand-crafted features, deep learning can automatically extract discriminant features from images, rendering them highly efficient for this task.

1.2 Motivation & Goals

Deep learning has made remarkable strides in plant disease classification; however, the substantial challenge exists due to the extensive scale of plant datasets. These datasets, covering a wide variety of plant types and diseases, are often exceptionally large.

In response to this issue, the primary goal of our study is to accelerate the training phases of our model by leveraging the power of parallel techniques, facilitated by the centralized high-performance computing (HPC) Discovery cluster. The second goal is to assess and compare the performance of serial and parallel deep learning, analyzing the potential benefits of parallel approaches. As a result, we aim to provide agricultural stakeholders with timely and precise disease identification, ultimately minimizing crop losses and strengthening agricultural resilience.

2 Methodology

The following table outlines the key steps and corresponding tools, offering a structured overview of our project, as shown in Table 1.

Table 1
Project Roadmap

Step	Roadmap	Focuses	Tools	Parallelization Comparison
1	**Exploratory Data Analysis**	Number of classes	NumPy, Matplotlib	
2	**Data Preprocessing**	Mean, std calculation for data normalization	(1) Dask Delayed (2) PyTorch	√
3	**Model Definition**	Resnet18	Torchvision	
4	**Pre-training**	(1) Train with initial hyperparameters on a small number of epochs (2) Time and Speedup comparison	PyTorch (1) Serial (2) multi-process (workers) (3) multi-thread (DataParallel)	√
5	**Hyperparameter Tuning**	Bayesian optimization	Optuna (multi-thread)	√
6	**Final Training and Model Evaluation**	(1) Train with optimized hyperparameters on extended epochs (2) assess with classification report and confusion matrix	PyTorch Scikit-learn Seaborn	

In our study, we employed a set of key methodologies, including:

Exploratory data analysis: Conduct a thorough examination of datasets featuring diverse plant diseases. Utilize the pandas feature in Python to classify and count images in specific folders for different diseases. Leverage matplotlib to visually categorize and chart plant diseases based on various plant species.

Data preprocessing: Given the augmented dataset, our focus lies in the normalization process. Employ mean and standard deviation to ensure consistent baseline values across the dataset. Implement Dask and PyTorch, two parallel computing methods, in the data standardization process. Document and compare the uptime and efficiency gains by varying the number of workers or processes.

Model definition: Utilize Convolutional Neural Networks (CNNs) with the ResNet-18 architecture. Its depth allows it to capture intricate features and patterns in images, which helps to address the vanishing gradient problem and facilitate deep network training.

Model training: Conduct model training using both serial and parallel approaches on the Discover cluster. To begin with, implement baseline serial training with PyTorch on a single GPU with one main process. Next, employ multi-process training by leveraging DataLoader's multi workers. Lastly, explore parallel training with PyTorch DataParallel.

Hyperparameter training: Optimize further through hyperparameter tuning using Bayesian search with Optuna. Tune essential parameters including learning rate, step size, weight decay and momentum. Implement a multi-threading study, parallelizing the optimization procedure.

Parallelization Performance Comparison: Evaluate and compare serial and parallel approaches using elapsed time and GPU utilization. Calculate speedup and analyze the speedup curve. Visualize metrics using Matplotlib and seaborn for a comprehensive comparison.

Model Evaluation: Assess the trained model on the test dataset, presenting a comprehensive evaluation through a classification report and confusion matrix. Scikit-learn will be utilized for metric computation, and Seaborn will aid in visualization.

3 Description of Dataset

The New Plant Diseases Dataset (https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset) is a comprehensive collection of plant leaf images, inspired by the PlantVillage dataset and curated by Samir Bhattarai.

3.1 Dataset Introduction

The total size of this dataset amounts to 1.43 gigabytes. It comprises approximately 87.9k images in JPG format, encompassing both healthy and unhealthy leaves. The leaves were removed from the plant, placed against a gray or black background.

It has been partitioned into a ratio of 80% for training data and 20% for validation data. Additionally, there is a separate test set consisting of 33 images.

3.2 Dataset Splitting

The original dataset comprises 33 images designated for testing, representing only 0.37% of the entire dataset. To enhance the efficacy of our training, we have undertaken the creation of a new test subset by partitioning a segment from the original training set. This results in a

distribution of 61,494 images for training, 17,572 for validation, and 8,814 for testing, maintaining a balanced ratio of 70% for training, 20% for validation, and 10% for testing, as illustrated in Table 2.

Table 2

Distribution of Images in Dataset Subsets After Splitting

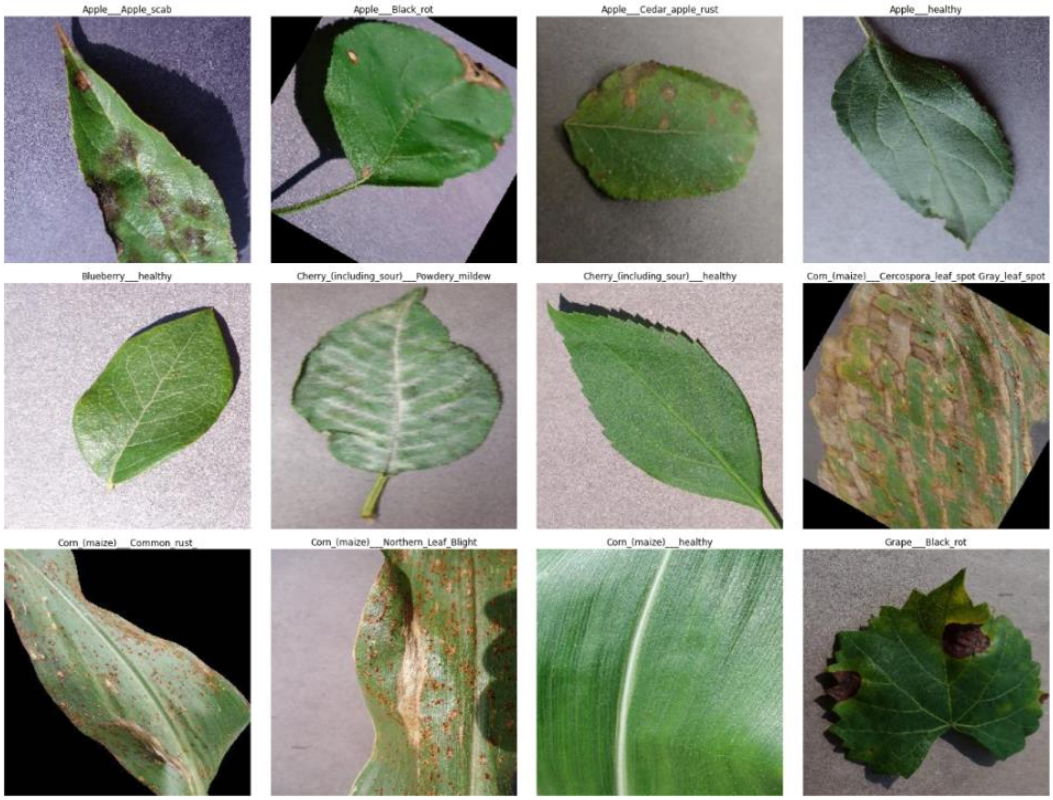
Subset	Image counts	Percent
train	61494	70%
valid	17572	20%
test	8814	10%
Total	**87880**	**100%**

3.3 Exploratory Data Analysis

To initially acquire an understanding and verify the applicability of the dataset, we strategically selected and presented twelve images, each representing a unique plant disease, as shown in Figure 1.

Figure 1

Data Samples



The dataset is organized into 38 distinct categories based on plant species and specific diseases, as illustrated in Figure 2.

Figure 2

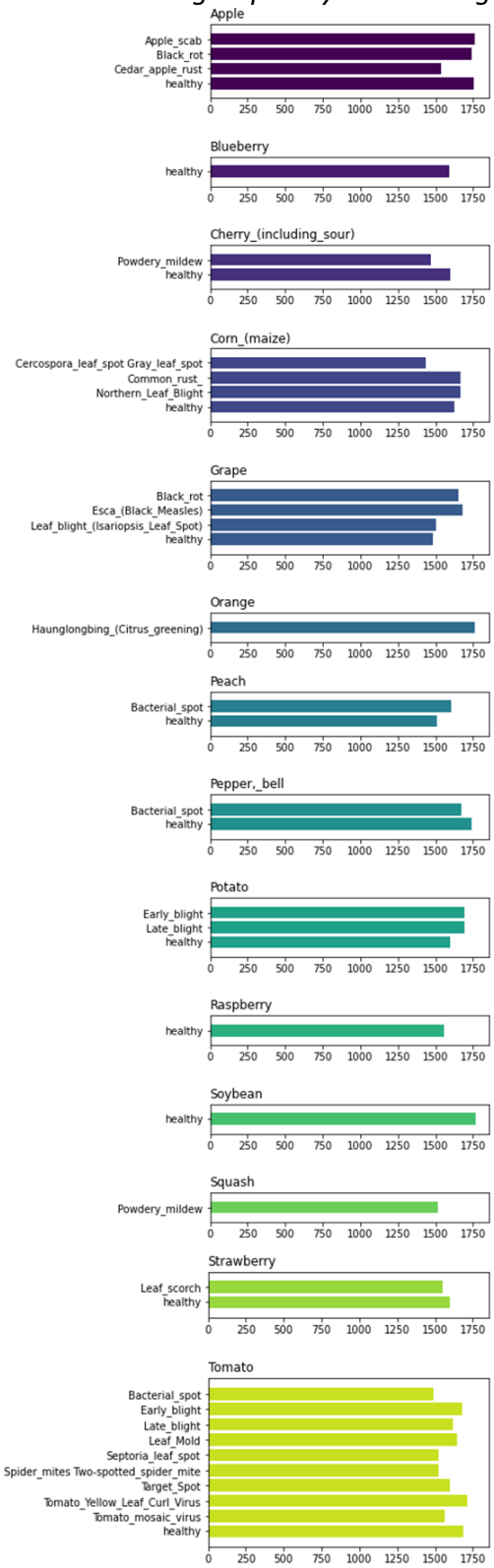
Distinct Categories of the Dataset

	File Name	Image Count
0	Apple___Apple_scab	1764
1	Apple___Black_rot	1738
2	Apple___Cedar_apple_rust	1540
3	Apple___healthy	1757
4	Blueberry___healthy	1589
5	Cherry_(including_sour)___Powdery_mildew	1472
6	Cherry_(including_sour)___healthy	1597
7	Corn_(maize)___Cercospora_leaf_spot Gray_leaf_...	1436
8	Corn_(maize)___Common_rust_	1668
9	Corn_(maize)___Northern_Leaf_Blight	1669
10	Corn_(maize)___healthy	1626
11	Grape___Black_rot	1652
12	Grape___Esca_(Black_Measles)	1680
13	Grape___Leaf_blight_(Isariopsis_Leaf_Spot)	1506
14	Grape___healthy	1480
15	Orange___Haunglongbing_(Citrus_greening)	1758
16	Peach___Bacterial_spot	1608
17	Peach___healthy	1512
18	Pepper,_bell___Bacterial_spot	1673
19	Pepper,_bell___healthy	1739
20	Potato___Early_blight	1696
21	Potato___Late_blight	1696
22	Potato___healthy	1596
23	Raspberry___healthy	1558
24	Soybean___healthy	1769
25	Squash___Powdery_mildew	1519
26	Strawberry___Leaf_scorch	1552
27	Strawberry___healthy	1596
28	Tomato___Bacterial_spot	1489
29	Tomato___Early_blight	1680
30	Tomato___Late_blight	1619
31	Tomato___Leaf_Mold	1646
32	Tomato___Septoria_leaf_spot	1526
33	Tomato___Spider_mites Two-spotted_spider_mite	1523
34	Tomato___Target_Spot	1598
35	Tomato___Tomato_Yellow_Leaf_Curl_Virus	1715
36	Tomato___Tomato_mosaic_virus	1566
37	Tomato___healthy	1685

There are 14 distinct plant categories, including Apple, Blueberry, Cherry, Corn, Grape, Orange, Peach, Pepper, Potato, Raspberry, Soybean, Squash, Strawberry. We classify plant diseases according to different plant species, which helps to quickly identify patterns and potential anomalies in the data [2], as depicted in Figure 3.

Figure 3

Plant Diseases grouped by Plant Categories



4 Results and Analysis

4.1 Data Preprocessing

The dataset comprises augmented and pre-cropped images, each with dimensions of 256x256 pixels. Our primary focus is on the normalization process, where the key objective is to standardize the range of pixel values across the dataset. This standardization is particularly crucial for deep learning models, ensuring that input features (pixel values) maintain a consistent scale. This practice promotes stable and efficient training, contributing to the robust performance of the models on the given data [3].

For dataset normalization, we employed Dask and PyTorch, utilizing parallelization to calculate mean and standard deviation respectively. To assess the impact of parallelization, we conducted experiments with 1, 2, 4, and 8 workers.

4.1.1 Using Dask

(1) Mechanism

We leveraged Dask's process-based task scheduler and dask.delayed function to optimize our workflow involving large-scale data processing. The dask.delayed function defers the execution of tasks, allowing for lazy evaluation. The process-based task scheduler distributes computations across multiple processes, enhancing parallelism and overall efficiency [4]. Subsequently, each deferred object, representing a computation task, is transformed into a Dask Array.

(2) Hardware

We allocated 4 CPU cores for 1, 2, 4, and 8 processes, as detailed in Table 3.

Table 3
Hardware Information using Dask

Hardware	Details
CPU	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz Architecture: X86_64 Count: 8

(3) Elapsed time

The elapsed time and speedup are showcased in Table 4, along with visual representations in Figure 4 and Figure 5.

Table 4

Calculation Elapsed time and Speedup Comparison Using Dask

Number of Workers	Elapsed Time(s)	Speedup
1	1321.3	1
2	840.0	1.57
4	526.0	2.47
8	459.9	2.87

Figure 4

Elapsed time Comparison Plot Using Dask

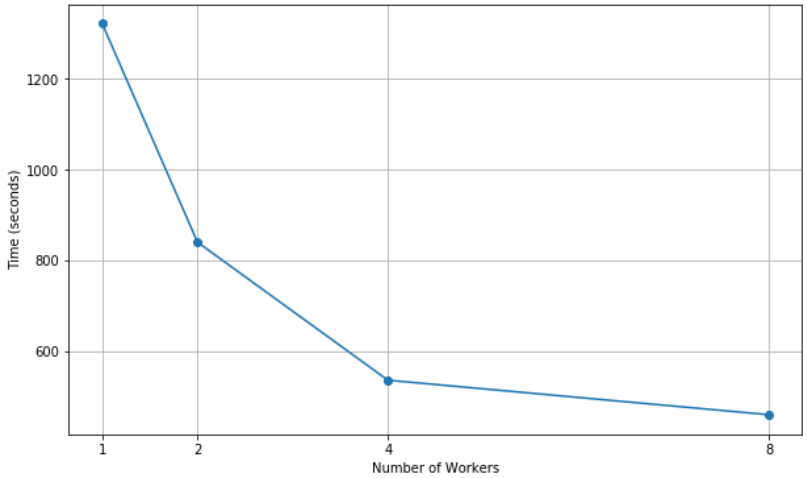
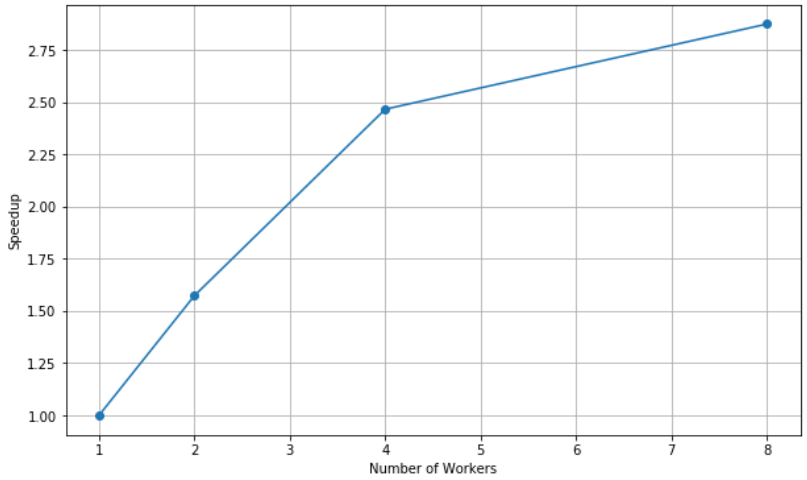


Figure 5

Speedup Comparison Plot Using Dask



Analyzing the running schedule and line charts above, it's evident that with a single Worker, the process takes the longest time, recorded at 1321.3 seconds, serving as the baseline for parallelization. Doubling the number of workers to two significantly reduces the time to 840.0

seconds, marking a reduction of about 36.5%. Scaling up to four workers further reduces the time to 535.9 seconds, showcasing a reduction of about 59.5% compared to a single Worker and about 36.2% compared to two workers. With eight workers, the runtime decreases to 459.9 seconds, reflecting a 65.2% reduction compared to a single Worker and a 13.2% reduction compared to four workers. However, it's notable that the reduction in runtime diminishes as more workers are added.

4.1.2 Using PyTorch

(1) Mechanism

We chose PyTorch DataLoader to optimize data loading during model training, enhancing memory efficiency by creating batches and supporting parallel loading with multiple child processes.

PyTorch DataLoader and its num_workers parameter enables multi-threaded data loading, allowing the code to process multiple batches of images in parallel. This accelerates the preparation of analysis or training data, taking advantage of PyTorch's batch processing capabilities and efficient tensor operations for quick and resource-efficient computation of mean and standard deviation.

(2) Hardware

We employ the same hardware configuration as when using Dask, as indicated in Table 3, which is previously referenced in the discussion on serial training.

(3) Elapsed time

The elapsed time and speedup are presented in Table 5, accompanied by visual representations in Figure 6 and Figure 7.

Table 5
Elapsed time and speedup Comparison Using PyTorch

Number of Workers	Time(s)	Speedup
1	378.3	1
2	207.8	1.82
4	122.3	3.09
8	83.7	4.52

Figure 6

Elapsed time Comparison Plot Using PyTorch

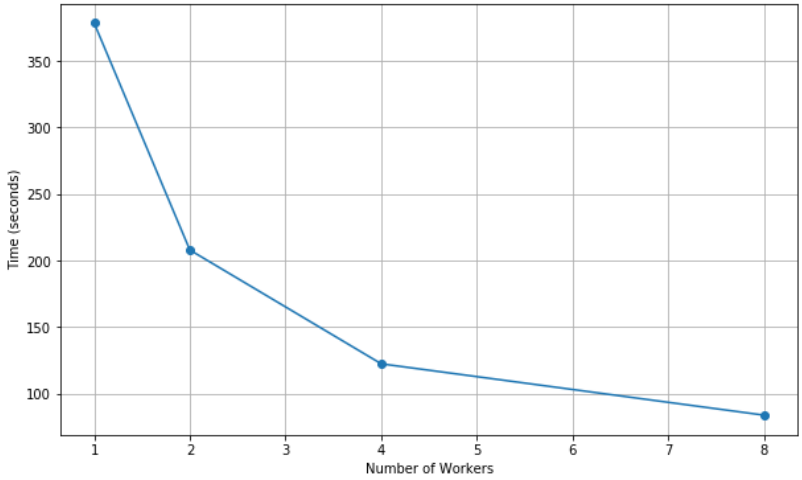
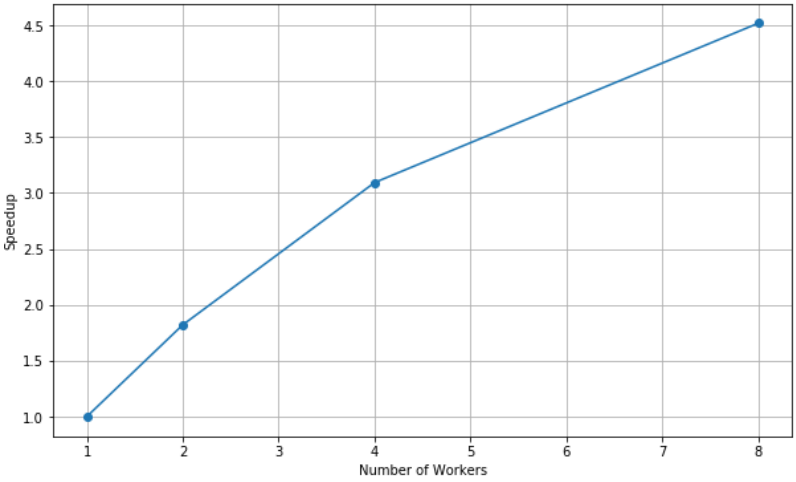


Figure 7

Speedup Comparison Plot Using PyTorch



In the case of serial running (using a single worker), the elapsed time is 378.3 seconds. Introducing 2 workers reduces the time to 207.8 seconds, marking a 45% improvement over a single worker. While this reduction is significant, it falls slightly short of the desired 50%, likely due to the overhead of managing parallel tasks. With 4 workers, the time is further reduced to 122.3 seconds, indicating a 68% improvement compared to the single-worker scenario and showcasing higher utilization for parallel processing. Moving to 8 workers results in a runtime of 83.7 seconds, representing a 78% improvement over the single-worker scenario. However, diminishing returns become more pronounced, with the improvement not scaling proportionally to the number of workers.

4.1.3 Comparison Summary

The initial times with a single worker differ significantly between Dask and PyTorch (1321.3 vs. 378.3 seconds), suggesting variations in the tasks performed or the efficiency of the frameworks. Both frameworks exhibit diminishing returns as the number of workers increases, a common characteristic of parallel processing. However, the point at which returns diminish differs, possibly due to distinct overheads or task parallelizability. The disparities in time reduction and speedup hint at PyTorch's potential suitability for the specific task or better optimization for parallel processing in this context.

While both Dask and PyTorch effectively employ parallel processing to reduce computation time, PyTorch demonstrates a more substantial reduction in time as the number of workers increases. This analysis underscores that, despite both frameworks being capable of parallel processing, their performance can vary considerably based on the task nature and available computing resources.

4.2 Pre-training

In this context, "pre-training" refers to the initial training phase before hyperparameter tuning. During this stage, a small number of epochs are executed with an initial set of hyperparameters to obtain an initial evaluation and baseline performance. This baseline serves as a reference point for comparison with performance after hyperparameter tuning.

We assessed both serial and parallel training techniques to identify the approach that achieves optimal training elapsed time.

4.2.1 Serial Training

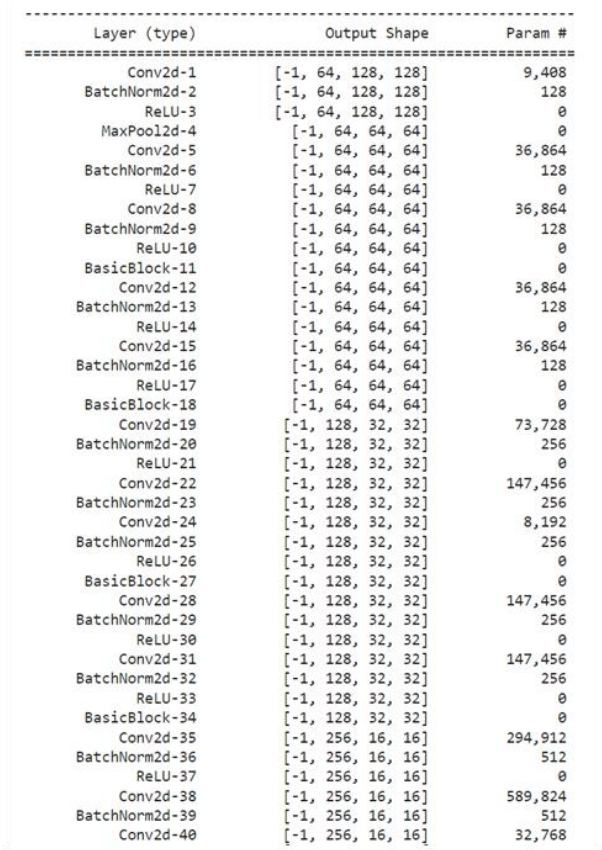
(1) Model Architecture

We utilized the ResNet18 implementation from Torchvision without pre-trained weights. The model architecture information depicted in Figure 8 is generated using the torchsummary package.

The architecture begins with a 7x7 convolutional layer followed by batch normalization and ReLU activation, incorporating max pooling for spatial dimension reduction. ResNet18 consists of four residual blocks, progressively increasing the number of filters. Global average pooling is applied after the residual blocks, leading to a 1x1 feature map. The architecture concludes with fully connected layers, including a softmax activation layer for classification [5].

Figure 8

ResNet18 Model Architecture



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 128, 128]	9,408
BatchNorm2d-2	[-1, 64, 128, 128]	128
ReLU-3	[-1, 64, 128, 128]	0
MaxPool2d-4	[-1, 64, 64, 64]	0
Conv2d-5	[-1, 64, 64, 64]	36,864
BatchNorm2d-6	[-1, 64, 64, 64]	128
ReLU-7	[-1, 64, 64, 64]	0
Conv2d-8	[-1, 64, 64, 64]	36,864
BatchNorm2d-9	[-1, 64, 64, 64]	128
ReLU-10	[-1, 64, 64, 64]	0
BasicBlock-11	[-1, 64, 64, 64]	0
Conv2d-12	[-1, 64, 64, 64]	36,864
BatchNorm2d-13	[-1, 64, 64, 64]	128
ReLU-14	[-1, 64, 64, 64]	0
Conv2d-15	[-1, 64, 64, 64]	36,864
BatchNorm2d-16	[-1, 64, 64, 64]	128
ReLU-17	[-1, 64, 64, 64]	0
BasicBlock-18	[-1, 64, 64, 64]	0
Conv2d-19	[-1, 128, 32, 32]	73,728
BatchNorm2d-20	[-1, 128, 32, 32]	256
ReLU-21	[-1, 128, 32, 32]	0
Conv2d-22	[-1, 128, 32, 32]	147,456
BatchNorm2d-23	[-1, 128, 32, 32]	256
Conv2d-24	[-1, 128, 32, 32]	8,192
BatchNorm2d-25	[-1, 128, 32, 32]	256
ReLU-26	[-1, 128, 32, 32]	0
BasicBlock-27	[-1, 128, 32, 32]	0
Conv2d-28	[-1, 128, 32, 32]	147,456
BatchNorm2d-29	[-1, 128, 32, 32]	256
ReLU-30	[-1, 128, 32, 32]	0
Conv2d-31	[-1, 128, 32, 32]	147,456
BatchNorm2d-32	[-1, 128, 32, 32]	256
ReLU-33	[-1, 128, 32, 32]	0
BasicBlock-34	[-1, 128, 32, 32]	0
Conv2d-35	[-1, 256, 16, 16]	294,912
BatchNorm2d-36	[-1, 256, 16, 16]	512
ReLU-37	[-1, 256, 16, 16]	0
Conv2d-38	[-1, 256, 16, 16]	589,824
BatchNorm2d-39	[-1, 256, 16, 16]	512
Conv2d-40	[-1, 256, 16, 16]	32,768
BatchNorm2d-41	[-1, 256, 16, 16]	512
ReLU-42	[-1, 256, 16, 16]	0
BasicBlock-43	[-1, 256, 16, 16]	0
Conv2d-44	[-1, 256, 16, 16]	589,824
BatchNorm2d-45	[-1, 256, 16, 16]	512
ReLU-46	[-1, 256, 16, 16]	0
Conv2d-47	[-1, 256, 16, 16]	589,824
BatchNorm2d-48	[-1, 256, 16, 16]	512
ReLU-49	[-1, 256, 16, 16]	0
BasicBlock-50	[-1, 256, 16, 16]	0
Conv2d-51	[-1, 512, 8, 8]	1,179,648
BatchNorm2d-52	[-1, 512, 8, 8]	1,024
ReLU-53	[-1, 512, 8, 8]	0
Conv2d-54	[-1, 512, 8, 8]	2,359,296
BatchNorm2d-55	[-1, 512, 8, 8]	1,024
Conv2d-56	[-1, 512, 8, 8]	131,072
BatchNorm2d-57	[-1, 512, 8, 8]	1,024
ReLU-58	[-1, 512, 8, 8]	0
BasicBlock-59	[-1, 512, 8, 8]	0
Conv2d-60	[-1, 512, 8, 8]	2,359,296
BatchNorm2d-61	[-1, 512, 8, 8]	1,024
ReLU-62	[-1, 512, 8, 8]	0
Conv2d-63	[-1, 512, 8, 8]	2,359,296
BatchNorm2d-64	[-1, 512, 8, 8]	1,024
ReLU-65	[-1, 512, 8, 8]	0
BasicBlock-66	[-1, 512, 8, 8]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Linear-68	[-1, 38]	19,494
Dropout-69	[-1, 38]	0

Total params: 11,196,006
Trainable params: 19,494
Non-trainable params: 11,176,512

Input size (MB): 0.75
Forward/backward pass size (MB): 82.00
Params size (MB): 42.71
Estimated Total Size (MB): 125.46

(2) Hardware

We utilized 4 CPU cores and 1 T4 GPU for the serial training process. Table 6 provides detailed information about the hardware devices used.

Table 6

Hardware Information for Serial Training

Hardware	Details
CPU	Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz Architecture: X86_64 Count: 4
GPU	Tesla T4 Count: 1

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:3B:00.0 Off |                    0 |
| N/A   52C    P0              29W /  70W |      2MiB / 15360MiB |      6%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

(3) Elapsed Time

We trained the model for 5 epochs and examined the elapsed time (see Appendix A). The elapsed times per epoch ranged from 427.79 to 445.89 seconds. Across all five epochs, the cumulative elapsed time totaled **2153.49** seconds. This serves as the reference serial time for comparative analysis.

4.2.2 Using Multi-process (workers)

(1) Mechanism

PyTorch DataLoader optimizes data loading by leveraging its num_workers parameter, which orchestrates the use of multiple processes to efficiently load and process batch data from disk into memory. This parallel processing capability is particularly beneficial when dealing with large datasets. The designated number of worker processes, often corresponding to the

number of available CPU cores, allows concurrent loading and preprocessing of batches, significantly improving data feeding speed [6].

Furthermore, the efficiency extends to GPU utilization during the training process. While the data loading occurs in parallel, the main training process can simultaneously consume batches from the queue, minimizing idle time and enhancing overall training iteration speed. This concurrent processing mechanism ensures that both CPU and GPU resources are effectively utilized, leading to improved training efficiency and reduced waiting times during model training. In our experimentation with 5 epochs, we systematically varied the num_workers parameter, evaluating its impact on time efficiency when set to values such as 0, 2, 4, and 8.

(2) Hardware

We utilized four CPUs and one GPU for the multi-processing training process, maintaining the same hardware configuration as in the serial training, as shown in Table 6, which is previously referenced in the discussion on serial training.

(3) Elapsed Time

We trained the model using 2, 4, and 8 processes (see Appendix A).

It is noteworthy that when running on 8 processes, a warning was triggered: "This DataLoader will create 8 worker processes in total. Our suggested max number of workers in the current system is 4, which is smaller than what this DataLoader is going to create. Please be aware that excessive worker creation might cause DataLoader to run slowly or even freeze. It's advised to lower the worker number to avoid potential slowness or freeze if necessary."

This warning was issued due to potential over-subscription, indicating that the number of workers exceeds the number of CPU cores. Adjusting the worker number to align with system capabilities is advised by PyTorch to ensure optimal performance and prevent potential slowdowns or freezes.

The elapsed time and speedup are presented in Table 7 and Figure 9.

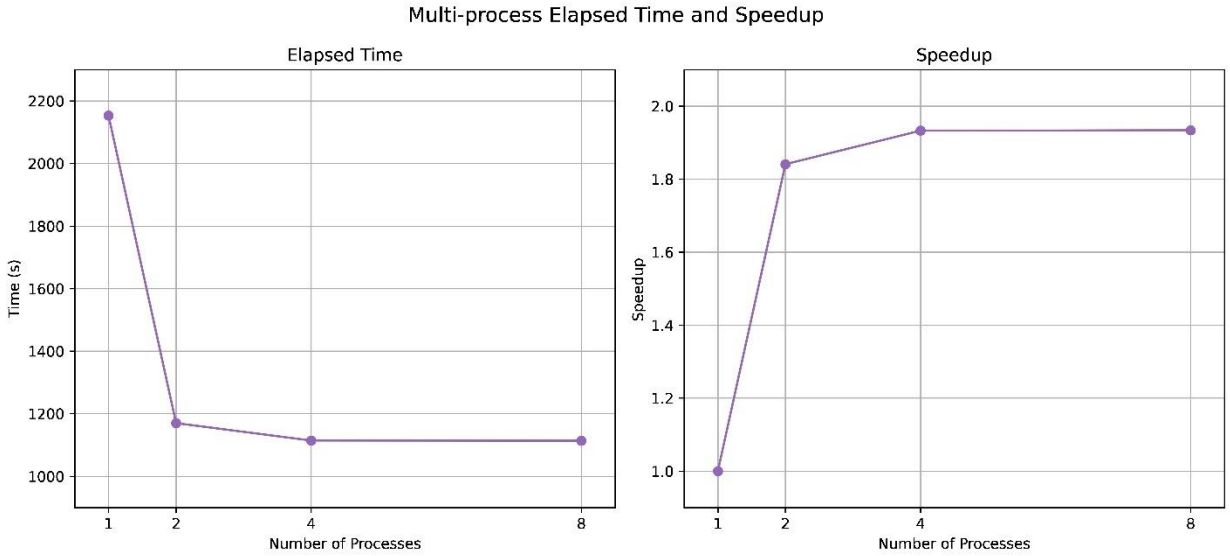
Table 7

Elapsed Time and Speedup Comparison Using Multi-process

Count of Processes	Elapsed time (s)	Speedup
Serial (1 process)	2153.4935	1
2 processes	1169.8231	1.8409
4 processes	1114.2052	1.9328
8 processes	1113.5024	1.9340

Figure 9

Elapsed Time and Speedup Comparison Plot Using Multi-process



By employing 2 processes, the elapsed time is substantially reduced to 1169.82 seconds, resulting in a commendable speedup of 1.84. Scaling up the parallelism to 4 processes further enhances efficiency, yielding a reduced elapsed time of 1114.21 seconds and an improved speedup of 1.93. However, increasing the number of processes to 8 under an oversubscription scenario shows minimal gains in speedup, maintaining a nearly consistent performance with an elapsed time of 1113.50 seconds and a corresponding speedup of 1.93.

4.2.3 Using Multi-thread (DataParallel)

(1) Mechanism

PyTorch's DataParallel is a container designed for a single process, multi-threaded execution on a single machine. This approach efficiently distributes both input data and the model across multiple GPUs, proving particularly advantageous when dealing with large datasets and complex models [7].

Throughout the training process, the input data is partitioned across available GPUs, enabling the simultaneous computation of forward and backward passes on each GPU. Once the forward and backward passes conclude on each GPU, the results are aggregated to collectively update the model parameters.

The DataParallel module simplifies the parallelization process, automatically managing the distribution and aggregation of data and model parameters. This facilitates parallel computation across multiple GPUs, streamlining the training of deep learning models.

(2) Hardware

We utilized four CPUs and 2 T4 GPUs for the DataParallel training, as detailed in Table 8.

Table 8

Hardware Information for DataParallel

Hardware	Details
CPU	Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz Architecture: X86_64 Count: 4 (for 1 processes)
GPU	Tesla T4 Count: 2

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.161.03   Driver Version: 470.161.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   69C    P8    12W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            Off  | 00000000:00:05.0 Off |                    0 |
| N/A   69C    P8    11W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

(3) Elapsed Time

We recorded the elapsed time during the DataParallel training progress (see Appendix A). The elapsed time and speedup are presented in Table 9 and Figure 10.

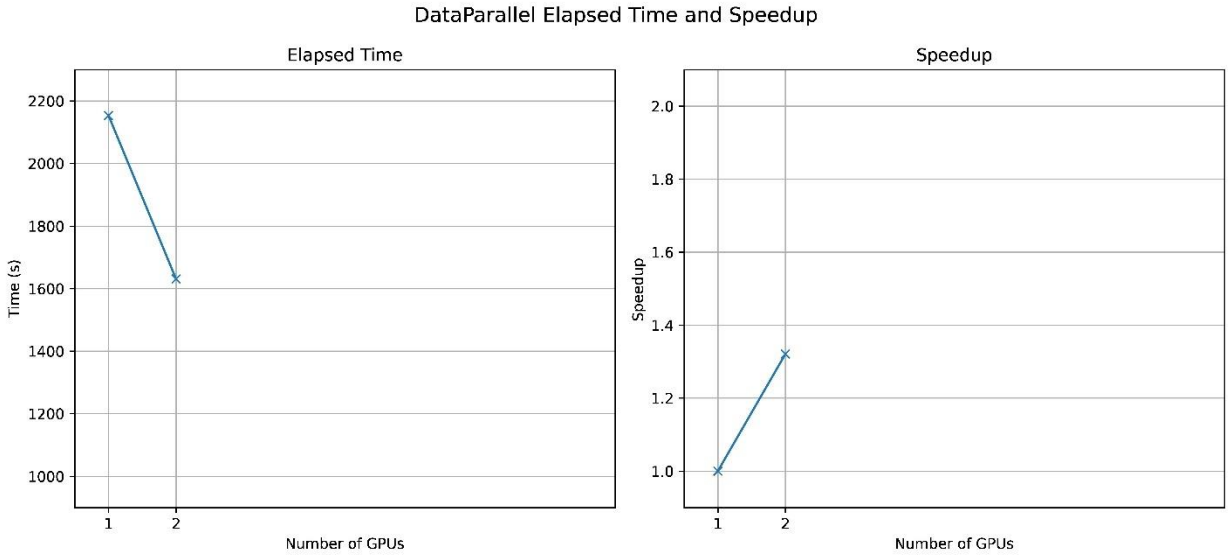
Table 9

Elapsed Time and Speedup Comparison using DataParallel

Count of GPU (T4)	Elapsed time (s)	Speedup
Serial (1 GPU)	2153.4935	1
2 GPU	1630.4431	1.3208

Figure 10

Elapsed Time and Speedup Comparison Plot using DataParallel



Introducing parallel processing with two GPUs reduced the elapsed time to 1630.4431 seconds, yielding a speedup of 1.3208.

4.2.4 Comparison Summary

Table 10, along with Figures 11 and 12, encapsulates a comprehensive summary detailing the comparison of training elapsed time and speedup between serial and parallel techniques.

Table 10

Training Elapsed Time and Speedup Comparison Summary

Training	Resources	Elapsed Time (s)	Speedup
Serial	Main process	2153.4935	1
Multi-process (workers)	2 processes	1169.8231	1.8409
	4 processes	1114.2052	1.9328
	8 processes	1113.5024	1.9340
Multi-thread (DataParallel)	2 GPUs	1630.4431	1.3208

Figure 11

Training Elapsed Time Comparison Plot

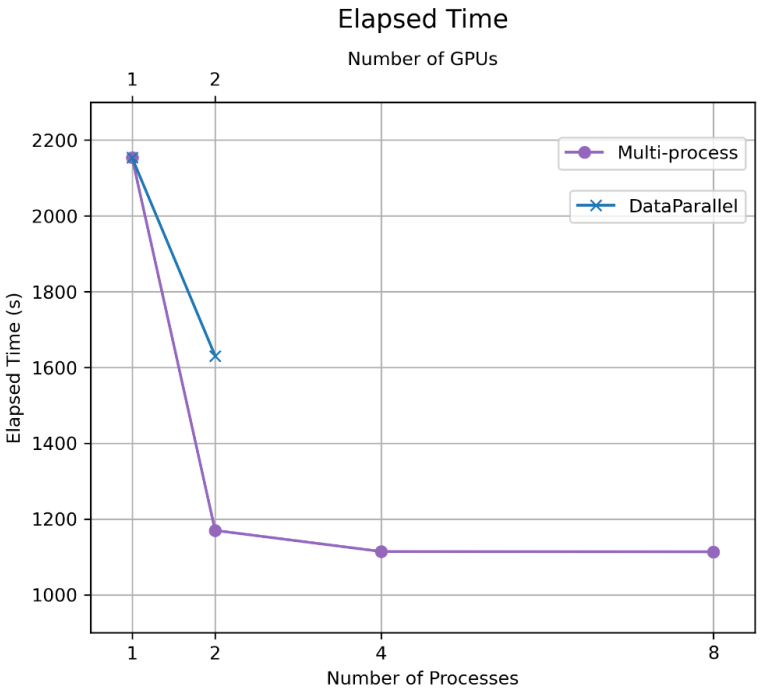
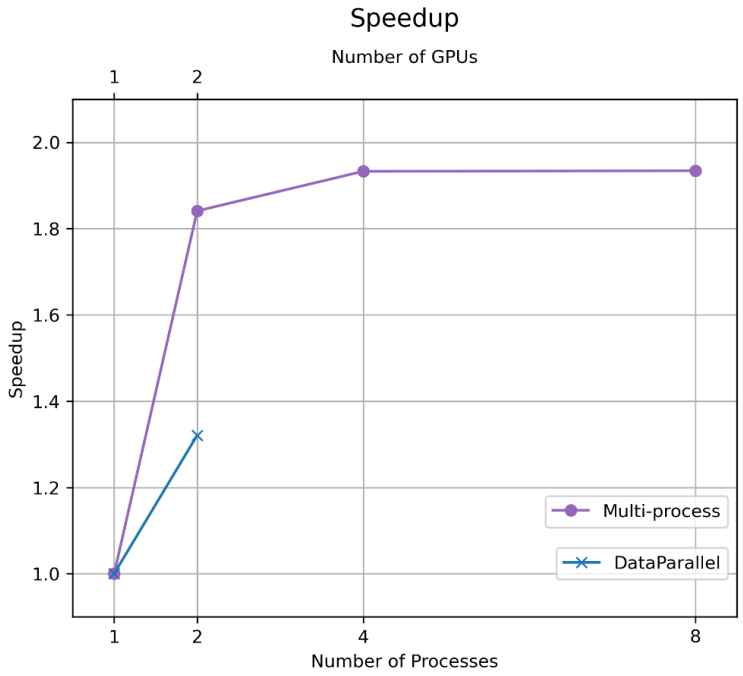


Figure 12

Training Speedup Comparison Plot



In the sequential execution with a single main process, the computational task took 2153.4935 seconds to complete.

Transitioning to multi-processing with 2, 4, and 8 processes resulted in notable reductions in elapsed time, achieving speedups of 1.8409, 1.9328, and 1.9340, respectively. Introducing multi-threading with 2 GPUs in a DataParallel configuration led to a further decrease in elapsed time to 1630.4431 seconds, corresponding to a speedup of 1.3208.

Specifically focusing on the 2-process multi-processing scenario, the computational task completed in 1169.8231 seconds, resulting in a notable speedup of 1.8409 compared to the sequential execution. On the other hand, employing multi-threading with 2 GPUs in a DataParallel configuration led to an elapsed time of 1630.4431 seconds, corresponding to a speedup of 1.3208 compared to the single-process sequential execution. Notably, the 2-process multi-processing approach outperformed the 2-GPU DataParallel configuration in terms of speedup.

4.3 Hyperparameter Tuning

(1) Mechanism

Optuna, an open source hyperparameter optimization framework to automate hyperparameter search, incorporates multithreading to significantly enhance the efficiency of the optimization process. It leverages the Parallel class from the Joblib library to create a parallel execution context and utilizes delayed function to wrap the function that needs to be executed in parallel. This setup enables Optuna to conduct multiple trials simultaneously, each exploring different sets of hyperparameter configurations.

(2) Hardware

We employed a high-performance computing setup featuring four AMD EPYC 7543 32-Core Processors and a single A100 GPU, as outlined in Table 11.

Table 11

Hardware Information for Hyperparameter Tuning

Hardware	Details
CPU	AMD EPYC 7543 32-Core Processor Architecture: X86_64 Count: 4
GPU	A100-SXM4-80GB Count: 1

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-SXM4-80GB          Off | 00000000:81:00.0 Off |                    0 |
| N/A   39C    P0              59W / 500W |      4MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

(3) Elapsed Time

We tuned four critical hyperparameters—learning rate, step size, weight decay, and momentum—through four sets of experiments, each utilizing 1, 2, 4, and 8 threads, respectively.

Within each experiment, we conducted 8 tuning trials (see Appendix B). Additionally, the comparison of elapsed time and speed is provided in Table 12 and visually represented in Figure 13.

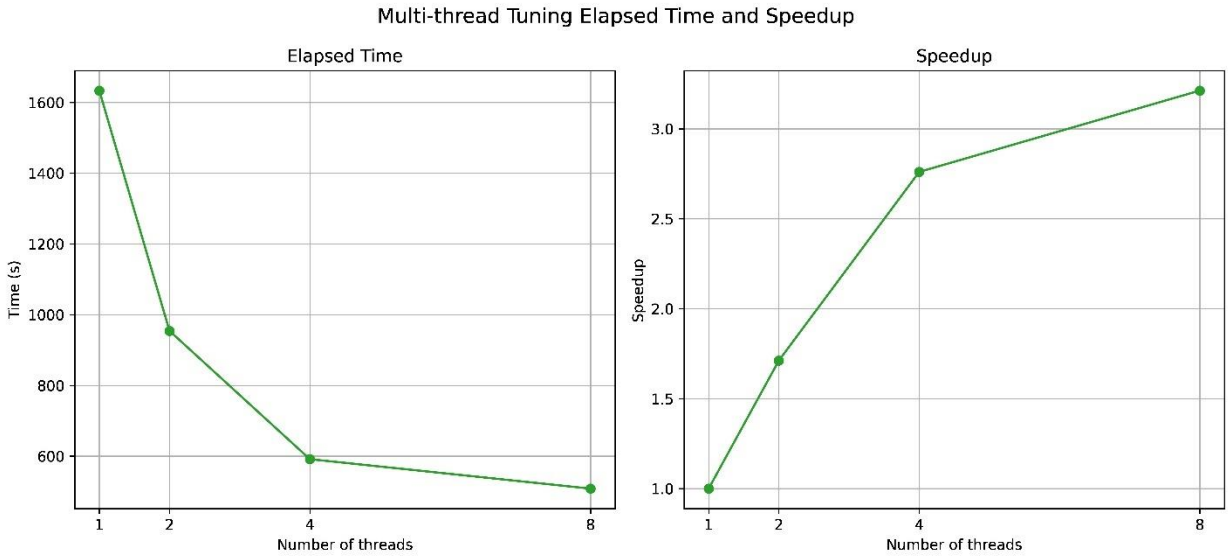
Table 12

Tuning Elapsed Time and Speedup Comparison

Count of Threads	Total Elapsed time (s)	Speedup
Serial (1 thread)	1633.4193	1
2 threads	954.1967	1.7118
4 threads	591.6575	2.7608
8 threads	508.3633	3.2131

Figure 13

Tuning Elapsed Time and Speedup Comparison Plot



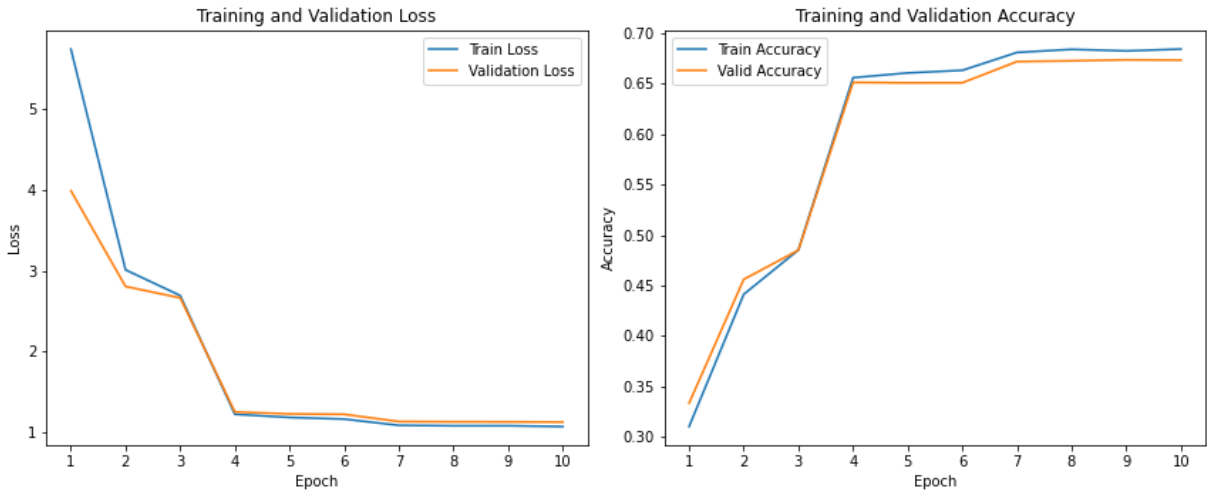
In the serial execution with a single thread, the task required 1633.4193 seconds. As the thread count increased to 2, 4, and 8, the total elapsed times saw significant reductions to 954.1967 seconds, 591.6575 seconds, and 508.3633 seconds, respectively. Correspondingly, the speedup metrics indicated enhanced efficiency, measuring at 1.7118, 2.7608, and 3.2131 for 2, 4, and 8 threads, highlighting the substantial performance gains achieved through parallelization.

4.4 Final Training and Model Evaluation

In the final training phase, we used the optimized set of hyperparameters obtained through the tuning process to train our model over 10 epochs, as illustrated in Figure 14.

Figure 14

Training and Validation Loss and Accuracy



The best-achieved accuracy on the training set reached 68.43%, while the validation accuracy was 67.36%.

The classification report in Figure 15 and the confusion matrix in Figure 16 provide a comprehensive evaluation of the classification model's performance across various plant diseases and health states.

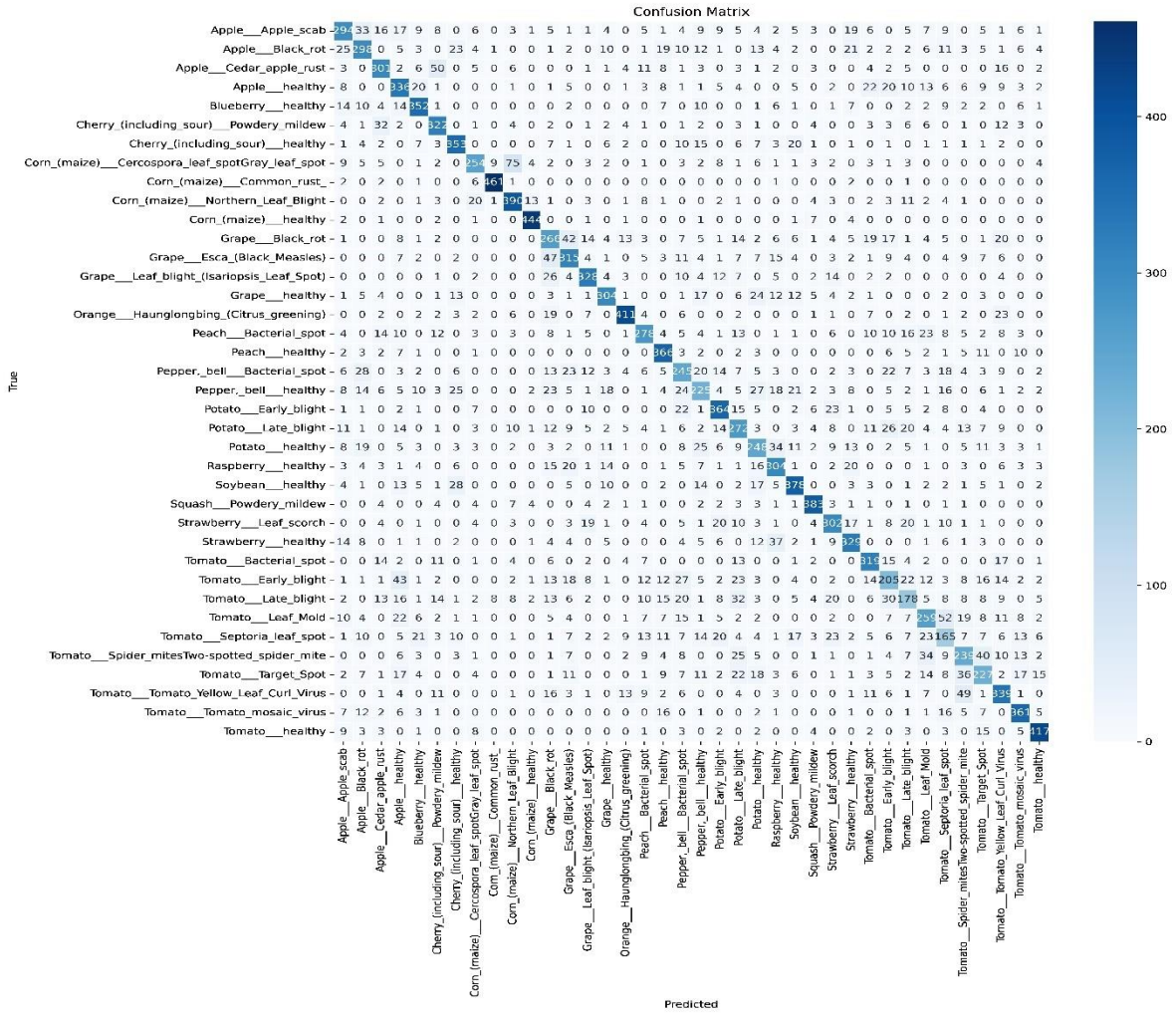
Figure 15

Classification Report

	precision	recall	f1-score	support
Apple___Apple_scab	0.64	0.58	0.61	504
Apple___Black_rot	0.63	0.60	0.62	497
Apple___Cedar_apple_rust	0.69	0.68	0.68	440
Apple___healthy	0.59	0.67	0.63	502
Blueberry___healthy	0.74	0.78	0.76	454
Cherry_(including_sour)___Powdery_mildew	0.70	0.76	0.73	421
Cherry_(including_sour)___healthy	0.74	0.77	0.76	456
Corn_(maize)___Cercospora_leaf_spotGray_leaf_spot	0.74	0.62	0.67	410
Corn_(maize)___Common_rust_	0.96	0.97	0.96	477
Corn_(maize)___Northern_Leaf_Blight	0.74	0.82	0.78	477
Corn_(maize)___healthy	0.94	0.95	0.95	465
Grape___Black_rot	0.52	0.56	0.54	472
Grape___Esca_(Black_Measles)	0.63	0.66	0.64	480
Grape___Leaf_blight_(Isariopsis_Leaf_Spot)	0.75	0.76	0.76	430
Grape___healthy	0.75	0.72	0.73	423
Orange___Haunglongbing_(Citrus_greening)	0.85	0.82	0.83	503
Peach___Bacterial_spot	0.69	0.61	0.64	459
Peach___healthy	0.72	0.85	0.78	432
Pepper,_bell___Bacterial_spot	0.51	0.51	0.51	478
Pepper,_bell___healthy	0.53	0.45	0.49	497
Potato___Early_blight	0.71	0.75	0.73	485
Potato___Late_blight	0.52	0.56	0.54	485
Potato___healthy	0.55	0.54	0.55	456
Raspberry___healthy	0.65	0.68	0.67	445
Soybean___healthy	0.74	0.75	0.74	505
Squash___Powdery_mildew	0.85	0.88	0.87	434
Strawberry___Leaf_scorch	0.67	0.68	0.68	444
Strawberry___healthy	0.71	0.72	0.72	456
Tomato___Bacterial_spot	0.69	0.75	0.72	425
Tomato___Early_blight	0.49	0.43	0.45	480
Tomato___Late_blight	0.49	0.38	0.43	463
Tomato___Leaf_Mold	0.59	0.55	0.57	470
Tomato___Septoria_leaf_spot	0.42	0.38	0.40	436
Tomato___Spider_mitesTwo-spotted_spider_mite	0.55	0.55	0.55	435
Tomato___Target_Spot	0.55	0.50	0.52	457
Tomato___Tomato_Yellow_Leaf_Curl_Virus	0.63	0.69	0.66	490
Tomato___Tomato_mosaic_virus	0.78	0.81	0.79	448
Tomato___healthy	0.87	0.87	0.87	481
accuracy			0.67	17572
macro avg	0.67	0.67	0.67	17572
weighted avg	0.67	0.67	0.67	17572

Figure 16

Confusion Matrix



The model demonstrates a reasonable overall accuracy of 67%, underscoring its ability to classify plant diseases and health statuses, though individual class performance varies.

Notably, classes like "Corn_(maize)__Common_rust" and "Orange___Huanglongbing(Citrus_greening)" exhibit high precision, recall, and F1-scores, with accuracies reaching 96% and 85%, respectively. These well-classified classes underscore the model's proficiency in accurately identifying instances of certain diseases.

However, some classes, such as "Tomato___Late_blight," "Tomato___Septoria_leaf_spot," and "Tomato___Early_blight," show lower precision, recall, and F1-scores, indicating the need for further training or model refinement. For instance, "Tomato___Late_blight" has a precision of 49%, recall of 38%, and an F1-score of 43%, suggesting potential challenges in distinguishing this class.

5 Conclusion

The application of parallelization extends beyond the acceleration of training in a deep learning project; it encompasses a comprehensive enhancement of the entire process. Based our study we can conclude that:

In the realm of data preprocessing, when calculating mean and std using Dask and PyTorch, both frameworks effectively harness parallel processing to expedite computations and enhance overall efficiency. Compared to Dask, PyTorch exhibits a more substantial time reduction compared to Dask during data preprocessing tasks.

In terms of model training, utilizing multiprocessing yields speedup improvements, but diminishing returns occur when processes exceed CPU core count. Increased GPU count with DataParallel accelerates training times, though multiprocessing outperforms in terms of performance.

In the phase of hyperparameters tuning, Optuna demonstrates notable speedup improvements in hyperparameter tuning with an increasing number of threads.

Note that the conclusions are contingent on the specific hardware utilized in the study, and varying hardware configurations may yield different results (see Appendix C). In summary, the study underscores the broad applicability of parallelization techniques across various stages of deep learning workflows.

References

[1] Gula, L. T. (2023, February 6). Researchers helping protect crops from pests. National Institute of Food and Agriculture. https://www.nifa.usda.gov/about-nifa/blogs/researchers-helping-protect-crops-pests

[2] Ingle, A. (2020, December 15). Plant disease classification - resnet- 99.2%. Kaggle. https://www.kaggle.com/code/atharvaingle/plant-disease-classification-resnet-99-2

[3] Brownlee, J. (2019, July 5). How to manually scale image pixel data for deep learning. MachineLearningMastery. https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/

[4] Schmitt, M. (n.d.). Understanding dask architecture: Client, scheduler, workers. Datarevenue. https://www.datarevenue.com/en-blog/understanding-dask-architecture-client-scheduler-workers.

[5] Modi, R. (2022, February 25). ResNet - understand and implement from scratch. Medium. https://medium.com/analytics-vidhya/resnet-understand-and-implement-from-scratch-d0eb9725e0db

[6] Datasets & dataloaders. Datasets & DataLoaders - PyTorch Tutorials 2.1.1+cu121 documentation. (n.d.). https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[7] Zhu, J. (n.d.). Getting started with distributed data parallel. Getting Started with Distributed Data Parallel - PyTorch Tutorials 2.2.0+cu121 documentation. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

Appendix A: Training Progresses

Serial Training Progress

```
Running on 1 GPU with 1 process--------------------------
Epoch [1/5]----------------------------------------------------------------------
| Train Loss: 2.6422 | Train Acc: 0.3406 | Val Loss: 2.2021, Val Acc: 0.4074
| Elapsed Time: 445.8777 s | Max GPU Memory Alloc: 1960.8457 MB
Epoch [2/5]----------------------------------------------------------------------
| Train Loss: 2.0209 | Train Acc: 0.4719 | Val Loss: 1.7971, Val Acc: 0.5216
| Elapsed Time: 424.5686 s | Max GPU Memory Alloc: 1960.8472 MB
Epoch [3/5]----------------------------------------------------------------------
| Train Loss: 1.8274 | Train Acc: 0.5141 | Val Loss: 1.6481, Val Acc: 0.5436
| Elapsed Time: 425.9290 s | Max GPU Memory Alloc: 1960.8481 MB
Epoch [4/5]----------------------------------------------------------------------
| Train Loss: 1.2209 | Train Acc: 0.6451 | Val Loss: 1.2316, Val Acc: 0.6412
| Elapsed Time: 428.5931 s | Max GPU Memory Alloc: 1960.8491 MB
Epoch [5/5]----------------------------------------------------------------------
| Train Loss: 1.2085 | Train Acc: 0.6483 | Val Loss: 1.2196, Val Acc: 0.6451
| Elapsed Time: 427.7931 s | Max GPU Memory Alloc: 1960.8501 MB

total_time: 2153.4934573173523
best_valid_acc: 0.6450603232415206
```

Parallel Training Progresses Using Multi-process

Count of Processes	Training Progress
2	Running on 1 GPU with 2 processes-------------------------- Epoch [1/5]---------- | Train Loss: 2.6422 | Train Acc: 0.3406 | Val Loss: 2.2021, Val Acc: 0.4074 | Elapsed Time: 248.3875 s | Max GPU Memory Alloc: 1960.8457 MB Epoch [2/5]---------- | Train Loss: 2.0209 | Train Acc: 0.4719 | Val Loss: 1.7971, Val Acc: 0.5216 | Elapsed Time: 227.4754 s | Max GPU Memory Alloc: 1960.8472 MB Epoch [3/5]---------- | Train Loss: 1.8274 | Train Acc: 0.5141 | Val Loss: 1.6481, Val Acc: 0.5436 | Elapsed Time: 231.8289 s | Max GPU Memory Alloc: 1960.8481 MB Epoch [4/5]---------- | Train Loss: 1.2209 | Train Acc: 0.6451 | Val Loss: 1.2316, Val Acc: 0.6412 | Elapsed Time: 227.1396 s | Max GPU Memory Alloc: 1960.8491 MB Epoch [5/5]---------- | Train Loss: 1.2085 | Train Acc: 0.6483 | Val Loss: 1.2196, Val Acc: 0.6451 | Elapsed Time: 234.1333 s | Max GPU Memory Alloc: 1960.8501 MB total_time: 1169.8230679035187 best_valid_acc: 0.6450603232415206

4	Running on 1 GPU with 4 processes---------- Epoch [1/5]---------- | Train Loss: 2.6422 | Train Acc: 0.3406 | Val Loss: 2.2021, Val Acc: 0.4074 | Elapsed Time: 223.4081 s | Max GPU Memory Alloc: 1960.8457 MB Epoch [2/5]---------- | Train Loss: 2.0209 | Train Acc: 0.4719 | Val Loss: 1.7971, Val Acc: 0.5216 | Elapsed Time: 222.7155 s | Max GPU Memory Alloc: 1960.8472 MB Epoch [3/5]---------- | Train Loss: 1.8274 | Train Acc: 0.5141 | Val Loss: 1.6481, Val Acc: 0.5436 | Elapsed Time: 223.5448 s | Max GPU Memory Alloc: 1960.8481 MB Epoch [4/5]---------- | Train Loss: 1.2209 | Train Acc: 0.6451 | Val Loss: 1.2316, Val Acc: 0.6412 | Elapsed Time: 220.5944 s | Max GPU Memory Alloc: 1960.8491 MB Epoch [5/5]---------- | Train Loss: 1.2085 | Train Acc: 0.6483 | Val Loss: 1.2196, Val Acc: 0.6451 | Elapsed Time: 223.2077 s | Max GPU Memory Alloc: 1960.8501 MB total_time: 1114.2051734924316 best_valid_acc: 0.6450603232415206
8	Running on 1 GPU with 8 processes---------- Epoch [1/5]---------- | Train Loss: 2.6422 | Train Acc: 0.3406 | Val Loss: 2.2021, Val Acc: 0.4074 | Elapsed Time: 224.6506 s | Max GPU Memory Alloc: 1960.8457 MB Epoch [2/5]---------- | Train Loss: 2.0209 | Train Acc: 0.4719 | Val Loss: 1.7971, Val Acc: 0.5216 | Elapsed Time: 222.2073 s | Max GPU Memory Alloc: 1960.8472 MB Epoch [3/5]---------- | Train Loss: 1.8274 | Train Acc: 0.5141 | Val Loss: 1.6481, Val Acc: 0.5436 | Elapsed Time: 222.1483 s | Max GPU Memory Alloc: 1960.8481 MB Epoch [4/5]---------- | Train Loss: 1.2209 | Train Acc: 0.6451 | Val Loss: 1.2316, Val Acc: 0.6412 | Elapsed Time: 222.4511 s | Max GPU Memory Alloc: 1960.8491 MB Epoch [5/5]---------- | Train Loss: 1.2085 | Train Acc: 0.6483 | Val Loss: 1.2196, Val Acc: 0.6451 | Elapsed Time: 221.2881 s | Max GPU Memory Alloc: 1960.8501 MB total_time: 1113.5023527145386 best_valid_acc: 0.6450603232415206

Parallel Training Progress Using DataParallel

Count of GPUs	Training progress
2	Running DataParrallel on 2 GPUs---------- Epoch [1/5]---------- | Train Loss: 2.7000 | Train Acc: 0.3273 | Val Loss: 2.3335, Val Acc: 0.3904 | Elapsed Time: 335.0122 s | Max GPU Memory Alloc: 1079.4219 MB Epoch [2/5]---------- | Train Loss: 2.0822 | Train Acc: 0.4565 | Val Loss: 1.8347, Val Acc: 0.5254 | Elapsed Time: 321.0952 s | Max GPU Memory Alloc: 1079.4233 MB Epoch [3/5]---------- | Train Loss: 1.9114 | Train Acc: 0.4969 | Val Loss: 1.7686, Val Acc: 0.5411 | Elapsed Time: 319.5789 s | Max GPU Memory Alloc: 1079.4243 MB Epoch [4/5]---------- | Train Loss: 1.2854 | Train Acc: 0.6260 | Val Loss: 1.2389, Val Acc: 0.6403 | Elapsed Time: 324.5508 s | Max GPU Memory Alloc: 1079.4253 MB Epoch [5/5]---------- | Train Loss: 1.2701 | Train Acc: 0.6299 | Val Loss: 1.2242, Val Acc: 0.6469 | Elapsed Time: 329.6849 s | Max GPU Memory Alloc: 1079.4263 MB total_time: 1630.4430594444275 best_valid_acc: 0.6469383109492375

Appendix B: Tuning Progresses

Tuning Progresses

Count of Threads	Tuning Progress
1	Trial 0 starts---------- [I 2023-12-13 02:49:02,477] Trial 0 finished Trial 1 starts---------- [I 2023-12-13 02:52:21,330] Trial 1 finished Trial 2 starts---------- [I 2023-12-13 02:55:39,138] Trial 2 finished Trial 3 starts---------- [I 2023-12-13 02:59:02,808] Trial 3 finished Trial 4 starts---------- [I 2023-12-13 03:02:25,692] Trial 4 finished Trial 5 starts---------- [I 2023-12-13 03:05:50,770] Trial 5 finished Trial 6 starts---------- [I 2023-12-13 03:09:11,952] Trial 6 finished Trial 7 starts---------- [I 2023-12-13 03:12:33,509] Trial 7 finished Elapsed time: 1633.419298171997 s
2	Trial 9 starts---------- Trial 8 starts---------- [I 2023-12-13 03:16:45,775] Trial 9 finished [I 2023-12-13 03:16:45,776] Trial 8 finished Trial 10 starts---------- Trial 11 starts---------- [I 2023-12-13 03:20:42,194] Trial 11 finished [I 2023-12-13 03:20:42,197] Trial 10 finished Trial 12 starts---------- Trial 13 starts---------- [I 2023-12-13 03:24:35,779] Trial 12 finished [I 2023-12-13 03:24:35,780] Trial 13 finished Trial 14 starts---------- Trial 15 starts---------- [I 2023-12-13 03:28:37,332] Trial 15 finished [I 2023-12-13 03:28:37,334] Trial 14 finished Elapsed time: 954.1966986656189 s

4	Trial 17 starts---------- Trial 16 starts---------- Trial 18 starts---------- Trial 19 starts---------- [I 2023-12-13 03:33:46,613] Trial 16 finished [I 2023-12-13 03:33:46,616] Trial 18 finished [I 2023-12-13 03:33:46,617] Trial 19 finished [I 2023-12-13 03:33:46,617] Trial 17 finished Trial 20 starts---------- Trial 21 starts---------- Trial 22 starts---------- Trial 23 starts---------- [I 2023-12-13 03:38:36,570] Trial 23 finished [I 2023-12-13 03:38:36,570] Trial 21 finished [I 2023-12-13 03:38:36,571] Trial 22 finished [I 2023-12-13 03:38:36,572] Trial 20 finished Elapsed time: 591.6575014591217 s
8	Trial 25 starts---------- Trial 26 starts---------- Trial 24 starts---------- Trial 27 starts---------- Trial 28 starts---------- Trial 29 starts---------- Trial 30 starts---------- Trial 31 starts---------- [I 2023-12-13 03:47:09,112] Trial 28 finished [I 2023-12-13 03:47:09,113] Trial 31 finished [I 2023-12-13 03:47:09,115] Trial 30 finished [I 2023-12-13 03:47:09,116] Trial 29 finished [I 2023-12-13 03:47:09,743] Trial 25 finished [I 2023-12-13 03:47:12,015] Trial 27 finished [I 2023-12-13 03:47:12,171] Trial 24 finished [I 2023-12-13 03:47:12,826] Trial 26 finished Elapsed time: 508.36329221725464 s

Appendix C: Hardware Information

Experiments	CPU	GPU
Mean & STD Calculation	Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz Count: 8	-
Serial Training **Multi-process Training**	Intel(R) Xeon(R) Gold 6240 CPU @2.60GHz Count: 4	Tesla T4 Count: 1
DataParallel Training	Intel(R) Xeon(R) Gold 6240 CPU @2.60GHz Count: 4	Tesla T4 Count: 2
Hyperparameters Tuning	AMD EPYC 7543 32-Core Processor Count: 4	A100-SXM4-80GB Count: 1