Comparative Study on Distributed Training Strategies

Abstract:

In this project, we delve into the world of distributed training strategies, specifically exploring their impact on training times for a widely used benchmark dataset, Fashion MNIST. The study primarily compares training times across four strategies: without MPI (Single-machine, single-GPU baseline), with MPI (Distributed training), Mirrored Strategy (Synchronous training across multiple GPUs on a single machine using TensorFlow), and Custom Data Parallelism (Manually dividing data and model parameters across GPUs on a single machine). Notably, the project sheds light on the observed variations in training times, emphasizing the significance of Custom Data Parallelism, which emerges as the most efficient method with the shortest training time. The findings provide valuable insights into the impact of different distributed training strategies, especially in the context of a relatively small-scale dataset like Fashion MNIST.

Introduction:

Deep learning models often require significant computational resources to train effectively. Distributed training leverages multiple computing devices, such as CPUs and GPUs, to accelerate training and handle larger datasets. This study evaluates the performance of four distributed training strategies. In the ever-expanding landscape of machine learning, the adoption of distributed training strategies has emerged as a pivotal approach to address the escalating demands posed by largescale datasets and complex model architectures.

From traditional Message Passing Interface approaches to more contemporary methodologies like TensorFlow's Mirrored Strategy and custom data parallelism, each method brings a unique set of advantages and challenges. The Fashion MNIST dataset, chosen as the benchmark for this study, provides a controlled environment to scrutinize the efficiency of these strategies.

Methodology:

Dataset Description

The Fashion MNIST dataset was chosen due to its moderate size and complexity, making it suitable for comparative analysis. Fashion-MNIST is a dataset of Zalando's article images, consists of 60,000 training examples and 10,000 test examples. It consists of grayscale images, each belonging to one of ten fashion categories. Each image is a 28x28 pixel grayscale picture, and the task is to classify these images into categories such as T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.

The dataset is commonly used for training and testing machine learning models, especially in the context of image classification. It offers a diverse set of clothing items, making it suitable for assessing the performance of algorithms in real-world scenarios beyond digit recognition. Fashion MNIST is widely employed for educational purposes, research, and as a standard benchmark in the machine learning community.

Preprocessing: In our project, the Pixel values were normalized to the range [0, 1] to facilitate model convergence.

1. Without MPI

Without MPI refers to a parallel computing approach that does not involve the use of MPI for communication and coordination between different computing nodes or processors. In the context of our project, it means that the training of the neural network model is performed on a single processing unit without distributing the workload across multiple nodes.

- The entire training process, including data loading, model computation, and optimization, takes place on a single machine or node.
- Without MPI, the parallelization of computations is limited to the capabilities of a single machine, and training is sequential.
- This approach is generally simpler to implement and understand, making it suitable for smaller datasets or scenarios where distributed computing may not be necessary.
- Training large neural networks on extensive datasets without parallelism may result in longer training times, especially if the computational demand exceeds the capabilities of a single machine.

Implementation: A neural network model was constructed using TensorFlow/Keras without utilizing MPI.

Configuration: The model was configured with a flattening layer, a dense layer with ReLU activation, dropout layer, and a dense layer with softmax activation.

Training Setup: Training was performed using Adam optimizer, categorical cross-entropy loss, and 10 epochs.

2. With MPI:

MPI, which stands for Message Passing Interface, is a standard communication protocol used for parallel computing. It is particularly prevalent in high-performance computing environments where tasks are distributed across multiple processors or nodes. MPI allows processes to communicate and share data with each other, enabling the coordination of parallel tasks in a distributed computing environment.

- MPI follows a message-passing communication model. Processes communicate by explicitly sending and receiving messages.
- MPI is designed for parallel computing, where multiple processes work together to solve a larger problem. Each process has its memory space and executes independently.
- Processes can send messages directly to each other using point-to-point communication. This allows for efficient data exchange between specific pairs of processes.
- MPI supports collective communication operations, such as broadcasting data from one process to all others or gathering data from all processes.

Implementation: The code includes MPI initialization and finalization to set up and clean up MPI resources. This ensures proper initialization before parallel execution.

Configuration: MPI (comm, rank, size) is used to synchronize the model parameters to ensure consistency across different MPI processes. MPI (comm, rank, size) assists in partitioning the training data (x_train, y_train) among different processes using np.array_split. MPI supports

collective operations, it is used to compute the average training time across all nodes (**train_time_avg = comm.reduce(train_time, op=MPI.SUM) / size).**
Training Setup: The training loop is distributed across MPI processes (MPI.Wtime(), model.fit). Each process handles a portion of the data (x_rank, y_rank) to parallelize the training. The code includes custom training steps (distributed_train_step, train_step), executed in parallel across MPI processes. The training loop is distributed across processes, with each process handling a portion of the data. It ensures that each process contributes to the overall training progress.

3. Mirrored Strategy:

Mirrored Strategy is a synchronous training strategy used in TensorFlow for distributed training across multiple GPUs on a single machine. The basic idea is to replicate the model on each GPU and train them simultaneously, synchronizing their updates to maintain consistency. The model is replicated across all available GPUs. Each replica processes a subset of the input data independently. During training, each GPU computes the gradients independently using its assigned subset of the data. This allows for parallel processing and faster training. The computed gradients are then synchronized across all GPUs. This involves averaging the gradients to ensure that all replicas have the same weight updates. The averaged gradients are used to update the model's weights on each GPU. This ensures that all replicas are consistent and learn from the same information.

Implementation: TensorFlow's Mirrored Strategy was employed for multi-GPU training. It's a popular choice for researchers and practitioners looking to scale up their models using available GPU resources efficiently.
Configuration: The model architecture was defined with MirroredStrategy scope, including flattening, dense, dropout, and softmax layers.
Training Setup: Training was performed using Mirrored Strategy, specifying batch processing and prefetching.

4. Custom Data Parallelism:

Custom Data Parallelism involves manually distributing the training workload across multiple devices or machines. It aims to distribute the training workload across multiple devices or machines by manually implementing data parallelism. We manually distribute the model and training data across devices or machines. A custom training loop is designed where each device processes a portion of the training data and computes gradients independently. It provides fine-grained control over distributed training, allowing customization based on specific requirements. However, it requires careful handling of parallelism and communication aspects, increasing the complexity of the implementation compared to higher-level strategies like Mirrored Strategy.

Implementation: TensorFlow's Mirrored Strategy was chosen for custom data parallelism.

Model Configuration: A custom model was defined within the strategy scope, including layers for flattening, dense, dropout, and softmax.

Training Setup: A custom training loop using tf.function and distributed_train_step was implemented for manual data parallelism.

Results:

The custom data parallelism approach demonstrated significantly lower training time (17.46 seconds), showcasing its efficiency compared to other strategies. Mirrored strategy, designed for synchronous training on multiple GPUs, exhibited good scalability with a training time of 29.21 seconds, providing a balanced performance. Comparing MPI and non-MPI approaches, it's evident that utilizing MPI resulted in a reduced training time (35.49 seconds), indicating the positive impact of message passing for distributed training.

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2878 - accuracy: 0.8922 - val_lo

9]: test_loss, test_accuracy = model.evaluate(x_test, y_test)
    print(f'Test Accuracy: {test_accuracy}')
    train_time_without_mpi = end_time - start_time
    print("Time taken without MPI: {:.2f} seconds".format(train_time_without_mpi))

313/313 [==============================] - 1s 2ms/step - loss: 0.3565 - accuracy: 0.8753
Test Accuracy: 0.8752999901771545
Time taken without MPI: 46.18 seconds
```

Without MPI Training time

```
if rank == 0:
    print("Time taken with MPI: {:.2f} seconds".format(train_time_avg))

Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2743 - ac
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2713 - ac
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2638 - ac
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2606 - ac
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2586 - ac
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2510 - ac
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2499 - ac
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2438 - ac
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2394 - ac
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2372 - ac
Time taken with MPI: 35.49 seconds
```

With MPI Training time

```
]: train_time = end_time - start_time
   print("Time taken with MirroredStrategy: {:.2f} seconds".format(train_time))

Time taken with MirroredStrategy: 29.21 seconds
```

Mirrored Strategy Training time

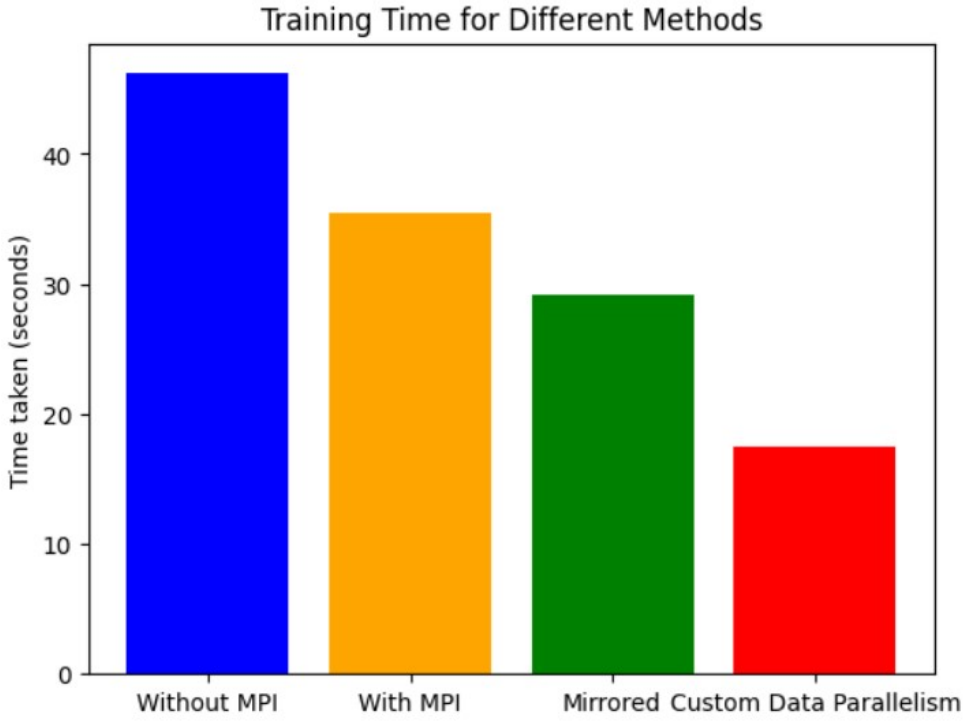
```
train_time_custom = end_time - start_time
print("Time taken with Custom Data Parallelism: {:.2f} seconds".format(train_time_custom))

WARNING:tensorflow:There are non-GPU devices in `tf.distribute.Strategy`, not using nccl al
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/devic
Time taken with Custom Data Parallelism: 17.46 seconds
```

Custom data parallelism Training time

Method	Training time(seconds)
Without MPI	46.18

With MPI	35.49
Mirrored Strategy	29.21
Custom data parallelism	17.46



Conclusion:

This study provides valuable insights into the different distributed training strategies for deep learning. Among the evaluated methods, Custom Data Parallelism demonstrates the most efficient training, with the shortest training time. Mirrored strategy exhibited good scalability providing a balanced performance. The Fashion MNIST dataset is relatively small, so the difference between the training time of our methods is also very small. With larger datasets, we might see clearer distinctions in training efficiency among the strategies.

Future Work:

- We can investigate the performance of the evaluated strategies on larger datasets to better understand their scalability and efficiency with more extensive and diverse data.
- Experimenting with different hyperparameter settings to optimize the performance of each distributed training strategy.
- We can investigate the impact of more complex model architectures on the training efficiency of the evaluated strategies.
- Exploring the implementation of asynchronous training approaches and evaluate their impact on training time and efficiency.

References:

- CNN with Tensorflow|Keras for Fashion MNIST | Kaggle
- https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy
- https://en.wikipedia.org/wiki/Message_Passing_Interface
- https://www.tensorflow.org/guide/distributed_training