Enhancing CNN Training on CIFAR-10 Through MPI Parallelization

Rohitkumar Datchanamourty (2023-26636) [1]

1. Abstract

In the pursuit of accelerating Convolutional Neural Network (CNN) training on the CIFAR-10 dataset, this project employs MPI parallelization strategies. The study investigates the impact of data partitioning and distributed training on both performance and efficiency. Through a systematic exploration of varying batch sizes and the influence of the number of processes, the project aims to optimize training times and uncover insights into the relevance and the scalability of parallelized CNN training. The source code is made available at [2].

2. Introduction

Efficient training of Convolutional Neural Networks (CNNs) poses challenges due to their growing complexity and the vast datasets they handle. This project explores the use of Message Passing Interface (MPI) for parallelizing CNN training on a single 8-core Intel Core CPU i7-9800X @ 3.80GHz, emphasizing the CIFAR-10 dataset.

The core goal is to assess the impact of MPI parallelization on CNN training efficiency and performance within the constraints of the 8-core CPU. The study delves into data partitioning and distributed training strategies, systematically varying parameters like batch size and process count to uncover insights into scalability and optimization.

This project contributes to understanding the challenges and opportunities associated with parallelization in resource-constrained environments. By focusing on the CIFAR-10 dataset, we aim to enhance the knowledge base on parallelized CNN training for modern computing architectures.

3. Background

3.1. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks have emerged as a dominant architecture for image classification tasks. Their ability to automatically learn hierarchical representations makes them well-suited for tasks such as object recognition. CNNs consist of layers of convolutional and pooling operations, followed by fully connected layers, enabling them to capture intricate features within images.

3.2. MPI Parallelization

Message Passing Interface (MPI) is a standardized communication protocol widely used for parallel computing. In the context of neural network training, MPI enables the distribution of computation across multiple processes, facilitating collaborative learning. Utilizing MPI for parallelization can lead to significant reductions in training time and enhanced scalability.

3.3. CIFAR-10 Dataset

The CIFAR-10 dataset comprises 60,000 32x32 color images across ten classes, making it a standard benchmark for image classification tasks. Each class contains 6,000 images, providing a diverse and challenging dataset for training and evaluating machine learning models.

3.4. Motivation for Parallelization

Parallelizing the training of Convolutional Neural Networks (CNNs) becomes imperative due to the computational demands posed by intricate models and large datasets. This project addresses the need for parallelization in training a CNN on the CIFAR-10 dataset, aiming to enhance efficiency on a single 8-core CPU. Employing MPI (Message Passing Interface) with mpi4py, the parallelization framework distributes computational workloads across processes, significantly reducing training time. The project explores MPI's synchronization capabilities, ensuring efficient communication and parameter sharing, unlocking the potential of distributed computing for accelerated CNN training and optimized hardware resource utilization.

4. Experimental Setup

The experiments were designed to evaluate the impact of key hyperparameters on the training performance of the Convolutional Neural Network (CNN). The hyperparameters considered include the number of processes, batch size, and the use of both model replication and data parallelism. The experimental design encompassed variations in these parameters to comprehensively assess their influence on training efficiency and model performance. To implement this project, I used [1] as a base to define a simplified CNN architecture that would suit the task (contained in model/-

model.py in the source code) and implemented by hand the rest, including the parallelization.

4.1. Model Architecture

The CNN architecture employed in this study was implemented using the PyTorch library. It consists of 2 convolutional layers followed by 3 fully connected layers.

4.2. Model Replication

Initially, the baseline performance was measured without any parallelization. Subsequently, a model replication approach was employed, where identical models were duplicated across different processes. This approach aimed to assess the performance gains achieved through parallel model training.

4.3. Data Parallelism

To further enhance parallelization, the dataset was partitioned, and data parallelism was implemented. This involved distributing portions of the dataset to individual processes, allowing models to concurrently train on distinct subsets. For data parallelism, a parameter synchronization approach was adopted. In this approach, each child model performed training on its partition of the dataset independently, updating its weights. Subsequently, the updated weights were sent to a master model, which gathered all the weight updates from the children. The master model then averaged these updates and sent the synchronized model back to each child. This synchronization process aimed to ensure consistency in the learned features across all child models.

The study examined the effectiveness of data parallelism in improving training efficiency, considering the introduced parameter synchronization for maintaining model coherence.

4.4. Number of Processes

To investigate the impact of parallelization, the number of processes was varied from 2 to 8, corresponding to the number of cores available on the CPU. This allowed for a systematic evaluation of the scalability of the training process.

4.5. Batch Size Variation

The influence of batch size on training performance was investigated with batch sizes of 2, 4, 8, 16, and 32.

4.6. Challenges

Embarking on my first-ever project involving parallelization and MPI posed a significant learning curve. The initial challenge revolved around implementing the distribution using the `torch.distributed` package, an MPI implementation in PyTorch. Despite extensive efforts, several days of troubleshooting did not yield a functional solution due to compatibility issues with my system, leading me to switch to an alternative solution: the mpi4py library.

Furthermore, structuring the code to manage the behavior of parent and child processes introduced complexity. The parent process, housing the master model, focuses on parameter synchronization and evaluation on the validation dataset. An excerpt from the parent process responsible for receiving information from child processes and performing parameter synchronization is outlined below:

```
for p in range(size-1):
   data = comm.recv()
   state_dicts.append(data['state_dict'])
   total_losses += data['total_loss']

avg_state_dict = OrderedDict()
for key in state_dicts[0].keys():
   avg_state_dict[key] =
      sum([sd[key] for sd in state_dicts])
         / float(size-1)
```

In addition, when updating the code to measure the execution time of different sections, determining what constitutes the communication, training, and idle phases presented another set of challenges. Hence, some choices of implementation for time measurements might be subject to criticism.

5. Results

5.1. Baseline Performance

The baseline performance was evaluated by training the Convolutional Neural Network (CNN) on the CIFAR-10 dataset without any parallelization or data partitioning, utilizing the resources of an 8-core CPU. The training was conducted for 15 epochs, aiming for an average total training time of around 12 minutes.

The baseline performance, without parallelization, achieved a peak validation accuracy of 63.6% and a minimum validation loss of 1.07% at epoch 6 (Figure 1). Subsequent epochs maintained stability, indicating the model's ability to generalize to the validation set. These values are considered as the baseline for comparison in the following parallelization strategies experiments. Also, the average CPU usage during training was of 97.6%.

5.2. Model Replication

The initial approach involved replicating models across multiple processes, each assigned a copy of the complete dataset. Employing four processes (one master model and three child

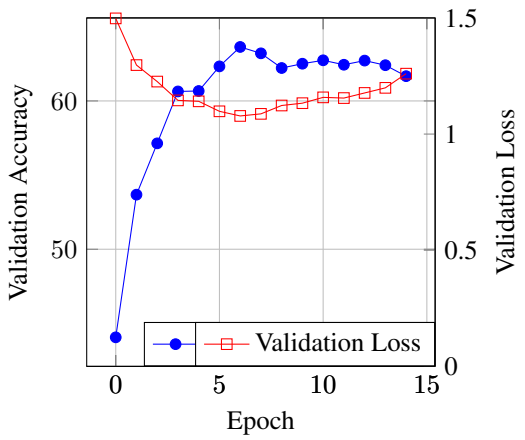

Figure 1. Single process Training

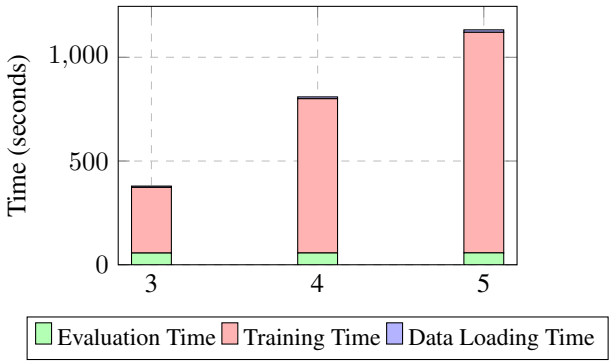

Figure 2. Time Breakdown for Different Number of Processes

workers), the replicated model achieved baseline performance within 8 epochs, completing the training in under 14 minutes. Furthermore, training for 15 epochs in under 23 minutes, we were able to reach an improved validation accuracy of 66.06% and an improved validation loss of 0.99%.

However, despite this success, model replication alone had limitations in achieving further performance gains. While reaching baseline performance, there were no real speedup compared to a single process training. Also, average CPU usage was of 79%, showing that CPU throughput and utilization was not maximised.

5.3. Data Parallelism

The implementation of data parallelism presented significant advantages in terms of speedup and some performance enhancements (Table 1). The experiment involved varying the number of processes, measuring validation accuracy, validation loss, training time, and average CPU usage. The number of epochs was fixed to 25 for all the following experiments.

Results demonstrate that using 3 processes led to performance improvement while reducing training time by approximately 50%. However, scaling to higher process counts showed diminishing returns, as training time increased significantly, potentially due to increase in CPU context switching. Additionally, with more workers, the dataset was partitioned into smaller chunks, leading to potential overfitting and reduced generalization ability, explaining downgrade in performance when the number of processes exceeds 4.

5.4. Influence of Batch Size

The influence of batch size on the parallelized training process was explored by varying the batch size parameter while setting the number of processes to 4 and adopting data parallelism.

The results (Table 2) indicate that smaller batch sizes lead to longer training times, likely due to increased need of computation for performing frequent gradient updates. However, larger batch sizes demonstrate reduced training times but may compromise validation accuracy. Batch size 8 appears to strike a balance, achieving high accuracy with a relatively short training time. The observed increase in training time with even larger batch sizes may be attributed to hardware limitations, specifically memory constraints. Larger batch sizes demand more memory, potentially exceeding the available resources on the CPU.

5.5. Execution Time Analysis

Figure 2 summarizes the execution time breakdown for different process counts, highlighting distinctive patterns in data loading, training, and evaluation times. Data loading times, albeit minimal, increase with more processes, aligning with increased parallelism and concurrent data loading. Then, training time steadily rises with more processes, likely due to heightened CPU utilization and increased context switching. Surprisingly, MPI communication time during trainings remains consistently under 1 second, challenging anticipated outcomes and questioning the choice of the time measurement methodology. Evaluation time remains relatively stable, minimally affected by varying process counts. A deeper investigation into the accuracy of time measurements is essential for a better understanding of communication's impact on training duration.

6. Conclusion

In conclusion, our exploration into parallelizing Convolutional Neural Network (CNN) training using Message Passing Interface (MPI) revealed some insights into the challenges and potentials of distributed computing. While our experiments showcased promising speedup and perfor-

Number of Processes	Validation Accuracy (%)	Validation Loss	Training Time (s)	Average CPU Usage (%)
3	64.4	1.04	375	55.4%
4	63.05	1.05	794	77.8%
5	60.93	1.10	1127	89.4%
6	59.41	1.16	1386	96.9%
7	57.95	1.18	1528	99.5%
8	55.28	1.25	1642	99.9%

Table 1. Varying number of processes for parallelization

Batch Size	Validation Accuracy (%)	Validation Loss	Training Time (s)	Average CPU Usage (%)
1	56.54	1.30	1332	69.3%
2	61.3	1.18	734	72.8%
4	63.48	1.10	578	73.5%
8	65.19	1.03	591	77.6%
16	63.59	1.04	761	77.7%
32	57.68	1.19	1034	75.5%
64	50.86	1.35	1129	71.8%

Table 2. Experimental Results with Varying Batch Sizes

mance improvements, there are huge limitations that must be acknowledged.

6.1. Limitations

One significant limitation lies in the methodology employed for time measurements, especially for communication phases. The accuracy and reliability of our timing results may have been impacted, and future work should address and refine the measurement approach to provide more precise insights into the performance of the parallelized system.

Additionally, the constraints of our experimentation environment, specifically the use of an 8-core CPU, impose limitations on the scalability of our approach. Extending this work to more powerful hardware configurations or distributed systems could offer a broader perspective on the scalability and efficiency of the proposed parallelization strategies.

The use of CIFAR-10, which is a relatively small dataset, leads to very small training subsets when partitioning among too many processes, causing overfitting. Experimentations with more vast datasets could be valuable.

6.2. Future Work

To further advance this project, future work could explore alternative approaches. One promising avenue is the investigation of a gradient upload approach, which involves experimenting with strategies to upload gradients rather than the entire model during synchronization.

Moreover, an exploration of different neural network architectures and datasets could provide a more comprehensive understanding of the generalizability of our findings. Also, using more deep learning parallelization-oriented libraries such as PyTorch's distributed package may facilitate research. Fault tolerance simulation is also a path to explore for more pragmatic results: it is currently implemented in the code but no experiment was run due to time constraints.

In conclusion, this work has explored the initial steps in parallelized CNN training, shedding light on both achievements and challenges encountered during the process. The findings presented in this study serve as a foundation for further exploration into the realm of distributed training methodologies. It is essential to acknowledge the limitations of this work, particularly in the methodology of time measurement, which may have impacted the interpretation of results.

7. References

[1] Code base: PyTorch CIFAR-10 Tutorial. Retrieved from https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[2] Code of the implementation: https://github.com/dat-rohit/distributed-neural-network/