Performance Evaluation of Tensorflow Parallel Strategies for Deep Learning Training on HPC Clusters

RAYAN RADDATZ DE MATOS, KENICHI BRUMATI, and MARCELO CARDOSO OLIVEIRA GULART, Federal University of Rio Grande do Sul, Brazil

The growing complexity of Deep Learning models makes training on single devices difficult, requiring distributed strategies like Data Parallelism. However, this approach creates critical synchronization challenges between processing units. This work evaluates TensorFlow parallel strategies on an HPC cluster to analyze how application factors contribute to time reduction. The study compares the standard synchronous method against a proposed Custom Training Loop designed to reduce bottleneck by synchronizing gradients less frequently. Experiments using ResNet50 and EfficientNetB0 models show that the Custom Training Loop significantly lowers communication overhead compared to the standard approach. Additionally, results indicate that increasing the batch size consistently improves performance. The authors conclude that correct configuration allows for performance gains in distributed training, even on networks with lower throughput.

Additional Key Words and Phrases: Deep Learning, HPC, Parallel Training, Distributed Training

ACM Reference Format:
Rayan Raddatz de Matos, Kenichi Brumati, and Marcelo Cardoso Oliveira Gulart. 2025. Performance Evaluation of Tensorflow Parallel Strategies for Deep Learning Training on HPC Clusters. 1, 1 (December 2025), 7 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In the last decade, Deep Learning has established itself as the state-of-the-art for solving complex problems, such as computer vision and natural language processing. Given the progress in algorithmic precision, there has been an exponential growth in the complexity of neural network architectures and the volume of datasets. This scenario imposes a massive computational load, making the training of robust models on a single processing unit an difficult task in terms of time and resources.

To mitigate these challenges, distributed training has become a standard practice, with an emphasis on the Data Parallelism strategy. In this approach, a complete replica of the model is instantiated on each processing unit, and the global dataset is partitioned into smaller, locally-focused batches. Although effective, this technique introduces critical synchronization challenges, as locally calculated gradients must be aggregated and communicated among all devices at every training step, typically through All-Reduce operations.

Given this fact, in this work we decided to analyze data parallelism for AI training using TensorFlow in a HPC cluster to investigate how the factors related to the application contribute to time reduction in a parallel scenario.

Software and Data Availability. We endeavor to make our analysis reproducible for a better science. We made available a companion material hosted in a public GitHub repository at https://github.com/kenichi220/comp-sys-perf-analysis-final-work/tree/main. Our companion contains the source code of this article and the software necessary to handle the created datasets. We also include instructions to run the experiment and figures.

Authors' address: Rayan Raddatz de Matos; Kenichi Brumati; Marcelo Cardoso Oliveira Gulart, Federal University of Rio Grande do Sul, Campus do Vale - Sector 4, Porto Alegre, RS, 91501-970, Brazil.

2025. XXXX-XXXX/2025/12-ART $15.00
https://doi.org/0000001.0000001

2 BACKGROUND AND EXPERIMENTAL CONTEXT

2.1 Background

TensorFlow is a machine learning system that operates at a large scale and in heterogeneous [1]. It uses dataflow graphs to represent the computation, shared state, and the operations that alter this state.

This architecture maps the nodes of a dataflow graph onto many machines within a cluster and, within a machine, onto multiple computational devices (multicore CPUs, GPUs, and TPUs). This offers flexibility to the developer to experiment with new optimizations and training algorithms, supporting a variety of applications focused on deep neural networks. When it comes to distributed training, TensorFlow offers different strategies with the `tf.distribute.Strategy` API. The TensorFlow documentations adverts that *"For synchronous training on many GPUs onmultiple workers, use the tf.distribute.MultiWorkerMirroredStrategy with the Keras Model.fit or a custom training loop."*[4]. Meaning for a distributed training across different nodes in a cluster there is two options: Using the MultiWorkerMirroredStrategy with the model.fit or creating a Custom Training Loop by yourself.

2.2 Experimental Context

In High-Performance Computing (HPC) environments, a highly recommended practice to access to shared computational resources is to mediate the access by workload management systems. In our context Slurm (Simple Linux Utility for Resource Management) is utilized. Slurm is an open-source, fault-tolerant, and highly scalable job scheduler, responsible for three main functions: Allocating exclusive or non-exclusive access to resources/nodes for users, providing a framework to initiate, execute, and monitor the work on the set of allocated nodes and managing a queue of pending jobs [7].

Interaction with Slurm is typically performed through shell scripts written in the BASH language. These scripts contain preprocessing directives that specify resource requirements, such as the number of nodes, tasks per node, and execution time limit, enabling the automation and reproducibility of computational experiments.

3 METHODS AND MATERIALS

3.1 Software Management and Reproducibility

Reproducibility of computational experiments is a critical challenge in computer science. To mitigate software dependency issues, the use of the GNU Guix package manager was explored. Guix implements the functional package management paradigm where each package is installed in a unique and immutable directory in the store, derived from a cryptographic hash of its dependencies [2].

However, the efficient training of deep neural networks using TensorFlow requires the use of proprietary NVIDIA hardware acceleration libraries, which are not officially distributed in standard Guix channels due to its strict Free Software policy.

Given this restriction, the strategy adopted in this work is the use of Python Virtual Environment for environment management. Although a functional version of Guix exists on GitHub without the proprietary libraries.

For managing the specific Deep Learning libraries, the python package manager `pip` with version 25.3 was utilized. The exact version of the used library was frozen and documented using the `pip freeze command`, generating a requirements file (`reqs.txt`) that enables the reconstruction of the execution environment used in the experiments.

3.2 Hardware and Software Configuration

The experiments were run using TensorFlow 2.15 as newer versions showed errors when trying to train using more than one node with the `.fit` and `MultiWorkerMirrodStrategy` and python 3.11.2. The NVIDIA drivers version was 550.54.15 and CUDA 12.4. We used up to three computational nodes connected with a network of 1G, each node contains an Intel(R) Core(TM) i7-14700KF processor at 3.40 GHz with 28 threads and 20 cores, 96 GB DDR5 RAM and NVIDIA GeForce RTX 4070 with 8GB memory accelerator.

3.3 Custom Training Loop

The expected execution from training with the `MultiWorkerMirroredStrategy` would show a slower training time as we increased the number of nodes in the execution. However, we faced a problem due to the connection while training that resulted in a higher training time as we increased the number of nodes for big models with a lot of parameters such as ResNet50. This problem happened because a synchronization is done for every batch computed by the `model.fit()` training method, and as we had only an 1G network, this synchronization resulted in a bottleneck that increased the training time. To mitigate this problem, we proposed a Custom Training Loop that instead of synchronizing the parameters between nodes in every batch, only synchronizes at a interval of batches, then resulting in a lower synchronization overhead. To this work we defined this interval of batches as 4 batches.

3.4 Design of Experiments (DoE)

We selected two different Deep Learning models to train: ResNet50 [5] as a big model with ~25.6M parameters and EfficientNetB0[6] as a small model with ~5.3M parameters. The two networks were trained using the TinyImageNet-200, a subset of the ImageNet[3] containing 100000 downsized to a 64×64 colored image divided among 200 classes. We created a full factorial for each model with varying batch size, the number of nodes used and the type of the training. The levels for the batch size factor were 32, 64 and 100, for the nodes we trained with 1, 2 and 3 nodes and for the type of training we had "sync" that uses the `model.fit` method to train or "ctl" that uses our custom training loop to train. This is a total of 2x3x3x2 different executions, each execution as replicated 5 times in a random order. Our output variable was the training time (full training time and time per epoch).

4 RESULTS

Figure 1 show a full view of our results. The graph presents the mean training time in minutes as a function of the number of nodes, faceted by type and batch size and with the color representing the model. Each bar represents the mean for the five executions for each configuration. The black transparent dots represents our observations. This allow us to see two main results: The first one is that the inclusion of more then one node introduces the necessity of communication, and together communication overhead. This overhead can be better seen with the smaller batches, as the computation is faster and more communication is needed. We can also see that the ResNet50 shows a higher time then the EfficientNetB0 in this cases. This happens because all the parameters needed to be synchronized after every batch, and the ResNet50 is almost 5 times bigger than the EfficientNetB0. The second one is that the ctl showed a considerable smaller communication overhead even for small batch sizes. The ctl showed OOM errors when running with the batch size 100 for training the EfficientNetB0 model.

After the communication overhead introduce by the inclusion of a second node, adding more nodes seemed to decrease the training time. We suppose that with enough nodes, the time gained by computing can overcome the overhead and be better then running it in one machine.

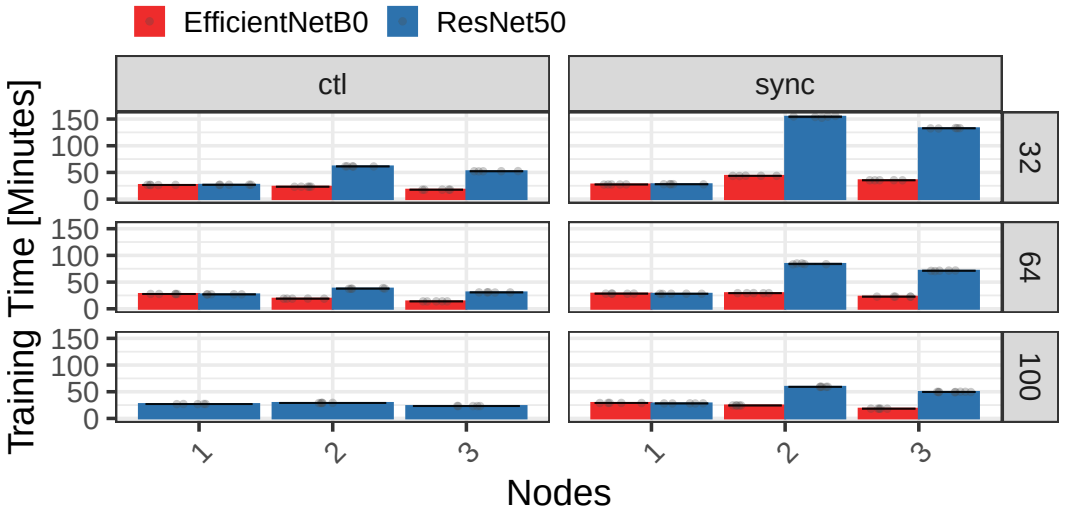

Fig. 1. Mean of experimental results.

4.1 The impact of node number and batch size

Figure 2 shows the mean training time grouped by node count. This enable us to see that in fact the increase of more then two machines results in a lower training time even though this gain is small. Figure 3 shows also an interesting result, we can see that increase of the batch size consistently show performance gains, but also decrease the variance. This means that conforms the batch size grows the training time is almost the same despite the node count.

4.2 Modeling a linear model

We propose a linear model to extrapolate our results and predict the training time. As the training time is almost constant for one node, we decided to just create the model based on the parallel executions, meaning that executions with more then one node were used to the model creation. For each combination of model and type we created a different model.

Figure 4 and 5 show a plot of our two proposed models based on the nodes count and the batch size. In the first model the batch size as a quadratic coefficient, represent the impact viewed in Figure 3. The second show a linear model

5 CONCLUSION

We could conclude that the network and the model size have a huge importance in the distributed training, but that even with a network with a low throughput, it is possible to gain performance when training by correctly configuring the environment and model parameters. As future work we intend to deep investigate the impact of the network and the model size in the distributed training.

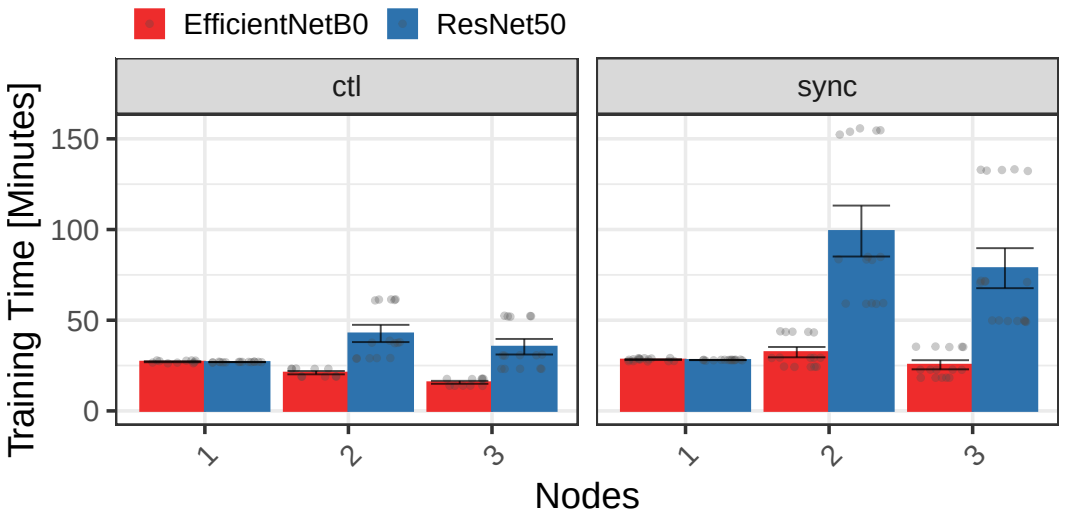

Fig. 2. Mean of experimental results grouped by node count.

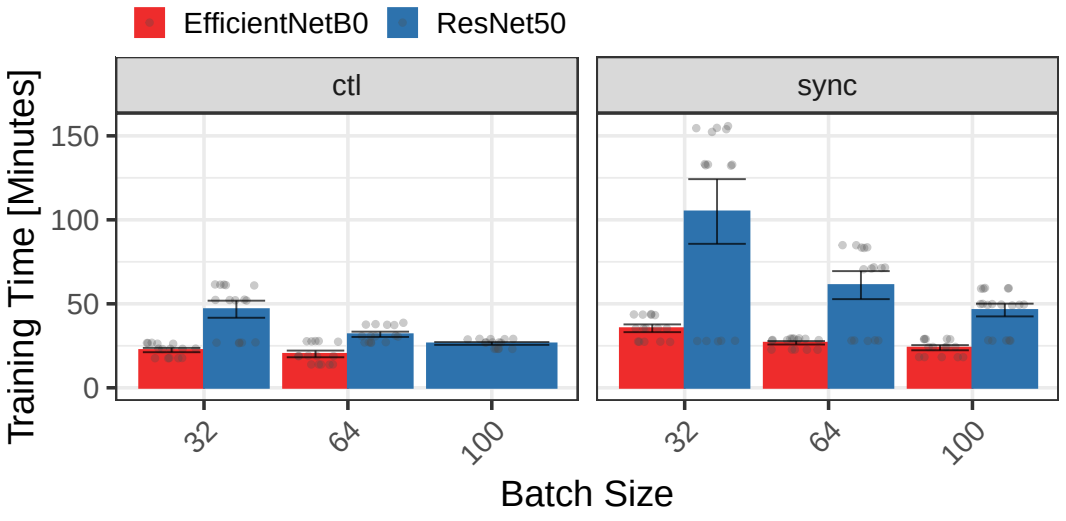

Fig. 3. Mean of experimental results grouped by batch size.

ACKNOWLEDGEMENTS

We would like to express our gratitude to Professor Dr. Lucas Mello Schnorr for his guidance and the valuable insights shared regarding performance analysis. We also would like to thank the PCAD at INF/UFRGS for making infrastructure and hardware used in the experiments available.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow:

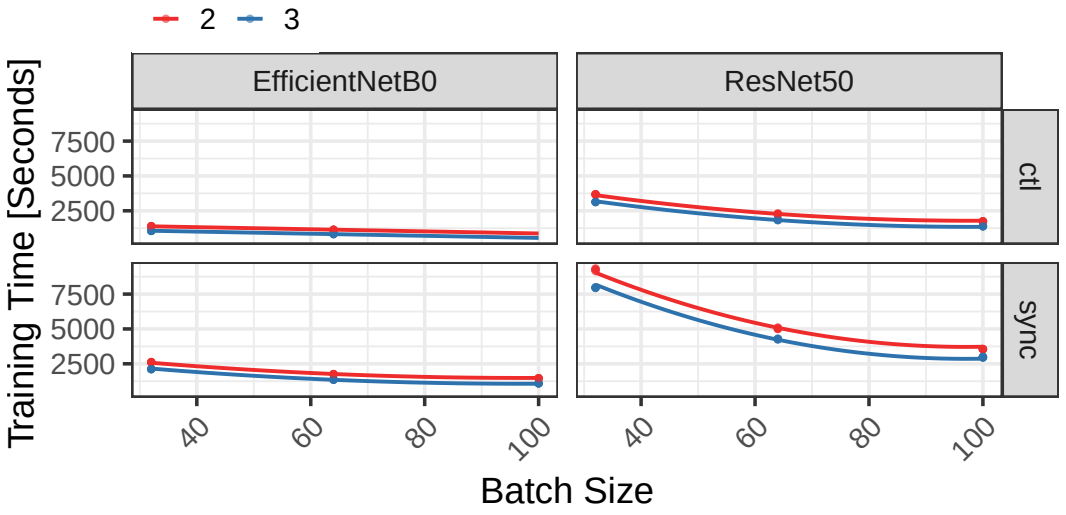

Fig. 4. Quadratic model

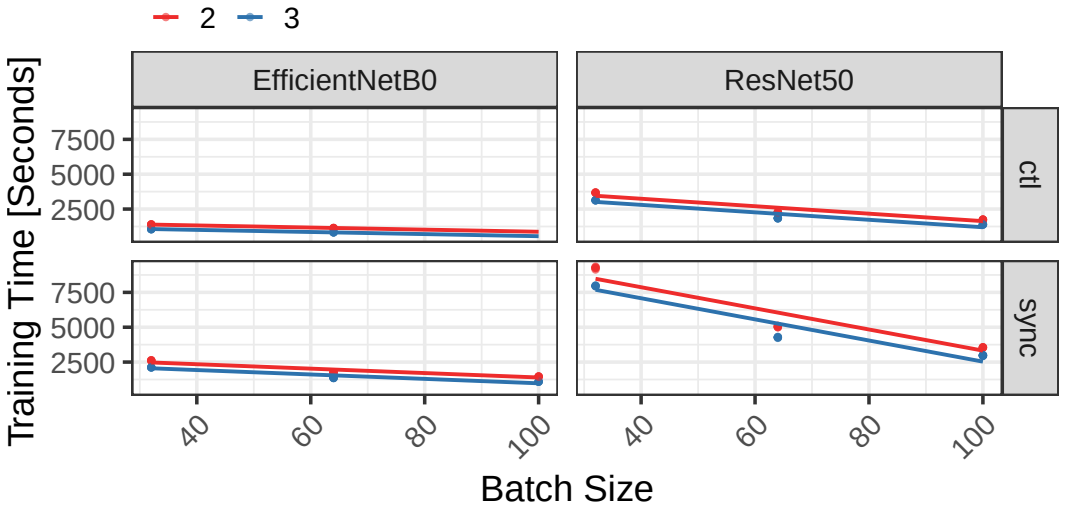

Fig. 5. Linear Model

A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association.

[2] Ludovic Courtès. 2013. Functional Package Management with Guix. In *Proceedings of the 6th European Lisp Symposium (ELS)*. 15–28. https://arxiv.org/abs/1305.4584

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[4] TensorFlow Documentation. 2023. Distributed training with Keras. https://www.tensorflow.org/tutorials/distribute/keras. Accessed: 2025-11-30.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[6] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR* abs/1905.11946 (2019). arXiv:1905.11946 http://arxiv.org/abs/1905.11946

[7] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*. Springer, 44–60. https://doi.org/10.1007/10968987_3