Evaluating and Improving Large-Scale Machine Learning Frameworks

Dan-Ovidiu Graur

Developed within:
Parallel and Distributed Systems Group - TU Delft
The Systems Group - ETH Zürich

Evaluating and Improving Large-Scale Machine Learning Frameworks

by

Dan-Ovidiu Graur

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday September 10, 2019 at 11:00 AM.

Student number:	4725646	
Project duration:	October 1, 2018 – June 30, 2019	
Thesis committee:	Dr. J. S. Rellermeyer,	TU Delft, supervisor
	Prof. Dr. G. Alonso,	ETH Zürich, supervisor
	Prof. Dr. D. H. J. Epema,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Contents

1 Introduction

Machine Learning has become an increasingly prominent field in the past two decades, continuously soaring to greater and greater levels of popularity. This is largely due to the breakthroughs made in the field of Deep Learning [35], which is arguably today's best known form of Machine Learning. The large degree of traction Deep Learning has gained in the past decade largely stems from the outstanding results it has produced in the context of certain tasks, such as Image Classification [49], Image Captioning [57], Image Generation [20], Speech Recognition [21], Speech Generation [54], Natural Language Processing [58], and many others. Deep Learning has regularly redefined what the state-of-the-art is, and has often significantly surpassed the performance of traditional Machine Learning methods. One such example is AlexNet [33], which in 2012 won the ImageNet Large Scale Visual Recognition Challenge, defeating the runner up by a margin of 10.8% in terms of the Top-5 error. It is thus no surprise that Deep Learning has become such a prominent field of research in both Industry and Academia, finding numerous applications in both practice and theory.

While Deep Learning has made a considerable impact in the field of Computer Science in the last decade, some of the ideas which stand at its foundation date back to as early as the 1940s. In fact, one of the first neural networks, called ADALINE, was proposed in 1960 [34]. The reasons why Deep Learning has only now become popular are numerous. One such reason is the so called *AI winters*[1] [12] which have significantly hindered progress within the field of Artificial Intelligence. Another important issue, that of the *Vanishing Gradients*, was formally identified by Sepp Hochreiter in 1991 [26]. It refers to the problem faced by Neural Networks, which use the Backpropagation algorithm during their training phase, wherein the gradients propagated backwards through the Network's layers decrease to insignificant values, thus allowing the Network's weights to remain nearly or completely unchanged. In such cases, the training phase reaches a point where it exerts little to no change to the neural network, and as a consequence becomes impractical. A number of successful solutions have been proposed to this problem, one popular example being the usage of Rectifier type activation functions, such as the ReLU [40].

Perhaps, however, the most important limiting factor in the research and development of Deep Learning, has been the lack of data and computing power, which are intrinsic to the success of such methods. In fact, one of the major reasons behind the *AI winters* was that many of the methods proposed within the field of Artificial Intelligence, while theoretically promising, were computationally intractable with the technology available at the time [38]. Deep Learning was no different. In recent years, however, it has become a viable avenue for research and real-world applications, due to the considerable developments in terms of computer hardware, and the abundance of data. However, while the volume of data seems to be in a continuous process of expansion, the computation power of a single node has had a difficult time keeping up with the increasing demands of complex Neural Network architectures

[1] Periods of time during which interest and funding of research and development in the field of Artificial Intelligence was decreased considerably.

paired with large scale learning problems. Single nodes with one or more GPUs have been a major force in enabling the training of Neural Networks in a reasonable amount of time, however, even so, there are cases when training can take days, weeks, or even months [24]. Considering the limitations and the expensive nature of scaling up, the more practical solution for increasing computation power and enabling sizable Deep Learning problems seems to be that of scaling out [51].

The progress of Machine Learning has currently no end in sight, largely due to the significant progress being made in the field of Deep Learning. With an ever increasing interest in the development of new Machine Learning methods, it was only natural that general purpose frameworks for developing such techniques would be developed, and made openly available to the Computer Science community. The number of such frameworks is considerable, with examples including: TensorFlow [2] (and Horovod [48]), Caffe2 [50] and Caffe [29], CNTK [47], Torch [11], and MXNet [10].

Considering the large range of available frameworks, Distributed Machine Learning strategies, and the complex, time consuming, and large-scale essence of Machine Learning, a question which naturally arises is: *what are the current limitations of Distributed Deep Learning, and what are their root causes?* Part of this complex question is what this thesis is tying to answer.

1.1. Problem Statement

Evaluating Large-Scale Machine Learning frameworks, with a focus on their Deep Learning capabilities, is a non-trivial task, with a high degree of complexity, which stems from multiple dimensions, including: setting up the systems in a large-scale distributed environment, understanding how the frameworks work and how they should be used, executing time-consuming evaluation runs, and developing the scripts which are necessary to capture the relevant metrics for evaluation. It is thus generally infeasible to perform an in-depth evaluation of all, or many of the available Machine Learning frameworks in a relatively short time frame. As a consequence, a more viable objective is to choose a few of the most popular Machine Learning frameworks, and perform an in depth comparative analysis on them.

Considering that TensorFlow and Caffe2 (now integrated into PyTorch) are some of the best known, and most popular frameworks for Deep Learning, they have been chosen as the focus point of this thesis. Moreover, both of these frameworks provide support for running distributed training on Neural Networks across a cluster of nodes. An additional point that is worth mentioning is that TensorFlow is a highly complex framework, which encompasses a large array of variable update architectures, which are employed in the context of distributed training. These architectures can greatly influence the performance of the framework, and as such, each should be evaluated individually.

Another important aspect of assessing these frameworks refers to deciding what exactly should be evaluated. Generally speaking, such frameworks are gauged against their Forward propagation time, Forward and Backward propagation time [4], processed images per second, CPU consumption, memory footprint and network usage [61]. These metrics should provide a good insight into how these software systems perform.

The evaluation itself should be done at scale, against a sizable number of nodes, in order to reveal the performance aspects of the aforementioned frameworks in the context of large scale learning.

Given the previously highlighted aspects surrounding the task at hand, it is possible to extract a number of **Research Questions (RQ)** which should be addressed by this thesis:

- **RQ1**: Which of the distributed parameter update strategies scale well as more nodes are added?
- **RQ2**: How well does the Parameter Server strategy scale relative to the batch size and the number of nodes?
- **RQ3**: What are the CPU usage, Memory footprint, and Network usage patterns of the distributed parameter update strategies?

The answers to these questions may yield interesting further research paths, which could help improve the quality and performance of Distributed Machine Learning. For instance, the answer to **RQ1** could help identify some of the core causes of limited scalability in distributed parameter update strategies. **RQ2** can help map the behavior and pinpoint the limitations of one of the most popular distributed parameter strategies available, the Parameter Server architecture, across a large range of batch and clusters sizes. Finally, **RQ3** provides an answer towards identifying which resources are essential to the available variable update strategy. This can serve as an indication of where improvements to these strategies can be made in order to obtain immediate performance improvements. As a whole, the results obtained for these questions should also provide a guide towards making the best decision when choosing the most suitable parameter update strategy and framework, given a specific set of resources.

1.2. Approach

In order to answer **RQ1**, a well-known Neural Network is chosen, which can fairly represent the performance of a framework. For this purpose, the chosen Network is ResNet50 [22], a Neural Network which features a landmark generic architecture that allows one to design very deep Neural Networks, to as many as a thousand layers (even if there are accuracy penalties at such depths) [22]. In this particular case, the network has 50 layers. Another important reason why this particular network was chosen is that there are pre-existing implementations of it, in both Caffe2 and TensorFlow, written by the developers of the respective systems. Since for this particular scenario, how well the model is trained is irrelevant, the data used for the study will be randomly generated. The major reason why the accuracy of the trained model is irrelevant is due to the fact that it should virtually be the same for all parameter update strategies in similar evaluation contexts, as long as the optimization method remains constant. The parameter update strategy simply serves the purpose of enabling the distributed learning process. The optimization method, however, is actually tasked with training the model. Choosing random data for the study should avoid biasing the results towards one particular dataset, and should rather show the generic performance of the systems. Finally, in order to actually evaluate how well the frameworks scale given a variable number of nodes, the forward times, forward and backward times, and the number of processed images per second are measured.

In order to answer **RQ2**, it is once more necessary to establish what the chosen Neural Network, and the workloads are. Given the reasons presented in the previous paragraph regarding ResNet50, it will again be the model of choice for running benchmarks. The dataset is chosen to be synthetic, as is the case for **RQ1**. The evaluation method itself refers to varying both the number of nodes, and the batch sizes, while measuring the forward times, forward and backward times, and the number of images per second. More specifically, the worker node count will be kept fixed, while the batch sizes will be increased. Once such a run is complete, the node count will be changed, and a new set of experiments of varying batch sizes will be run.

Answering **RQ3** refers to tracking the CPU usage, Memory footprint, and Network consumption, while running a fixed workload across Caffe2 and the various variable update schemes available in TensorFlow. For this experiment, the workload will once more be synthetic data, while the model selected for evaluation is ResNet50.

The underlying computational infrastructure required for running all of the aforementioned multi-node large-scale experiments is provided by The Distributed ASCI Supercomputer 5 (DAS-5) [5], a 6 cluster wide area system.

1.3. Thesis Outline and Contributions

Chapter 2 provides an overview of Distributed Machine Learning, with a focus on Deep Learning, parameter update architectures, and large scale Machine Learning frameworks. Chapter 3 focuses on evaluating the scalability of different distributed parameter update architectures, and providing insight into the root causes of their limitations. Chapter 4 takes a deeper look at the Parameter Server architecture, and presents the scalability limitations in

terms of batch and cluster size. Chapter 5, presents the hardware demands characteristic to some of the parameter update strategies, as well as the sources of bottleneck for some of the tested strategies. Finally, chapter 6 presents the final conclusions of the thesis, and ideas for further work.

The main contributions of this thesis are the following:

- Provide an in depth performance comparison of some of the most popular distributed parameter update strategies, with regards to the forward propagation time, forward and backward propagation time, number of processed images per second, CPU usage, Memory footprint, and Network consumption, across a large number of experimental setups at large scale.
- Identify performance bottlenecks in some of the parameter update strategies, and provide insight into why these bottlenecks exist.
- Propose an objective means of evaluating and comparing the performance of large-scale Machine Learning frameworks, which should be reproducible for other frameworks as well, thus allowing for a fair comparison. The code of this thesis is made publicly available in the hopes that it will serve as a reference for others wishing to perform similar evaluations, or scrutinize this work[2].
- Attempt to fill in a research gap, which exists due to the lack of publicly available reports and research papers containing objective and in-depth evaluations of parameter update strategies and Machine Learning frameworks.

[2]Thesis GitHub repository: `https://github.com/DanGraur/msc-thesis`

2

Background and Related Work

This chapter provides an overview of the some of the most relevant background information pertaining to the work presented in this thesis. Section 2.1 provides a brief introduction into the world of Machine Learning. Section 2.2 presents some of the major aspects of Deep Learning. Section 2.3 details the ResNet Deep Neural Network. Section 2.4 presents some of the most relevant distributed Machine Learning architectures, topologies and execution strategies. Section 2.5 goes over some of the most popular Machine Learning frameworks, including TensorFlow and Caffe2. Finally, section 2.6 describes the structure of the DAS-5 cluster and its nodes.

2.1. Machine Learning

Machine Learning is a field within Computer Science which focuses on developing methods and algorithms for making predictions on data without making use of explicit instructions or explicit programming, by leveraging patterns and inference via mathematical models [6, 53]. Machine Learning is generally employed when the task of writing an exact algorithm for scenarios wherein a specific task must be completed is either infeasible or too difficult. It generally relies heavily on data in order to deliver a successful solution to a given problem.

Machine Learning is an expansive field, which encapsulates several major subcategories: *Supervised* and *Semi-Supervised Learning*, *Unsupervised Learning*, and *Reinforcement Learning*.

2.1.1. Supervised and Semi-Supervised Learning

In this category of Machine Learning data usually consists of a number of features and a supervisory signal. The supervisory signal is the ground truth output for a given data entry. It is sometimes also known as a label. Each data entry can be seen as a vector, where each of its dimensions represents a feature. The entire set of data entries can be seen as a matrix. If each data entry hosts a supervisory signal, then the type of Machine Learning task at hand is *Fully Supervised*. If, however, only a part of the data entries contain supervisory signals, then the challenge falls within the category of *Semi-Supervised problems*, where some entries have a label, and some do not. Most of Deep Learning falls under the umbrella of Supervised Learning [35].

A large effort has been put into developing methods which are capable of solving such tasks, which are arguably the focus point of current research in Machine Learning [35]. As such, there is no shortage of methods which can tackle Supervised and Semi-Supervised challenges, some of the most popular being: Support Vector Machines, Linear Regression, Logistic Regression, k-Nearest-Neighbor, Linear Discriminant Analysis, Decision Trees, Naïve Bayes, and Neural Networks [6, 53].

Generally speaking, although not always the case, algorithms used in Supervised Learning are defined by two disjoint phases: the *training phase*, and the *prediction phase* [6, 53]. In the training phase, the available labeled data is exploited in order to optimize a mathematical

model with respect to the aforementioned data. In the prediction phase, the optimized model is used towards producing predictions on new, unseen data. There is no shortage of real-world problems which fall into the category of Supervised Learning. Some examples include: predicting a medical diagnosis based on patient data, epidemic outbreak prediction, predicting the success of a university application, image classification, drug activity prediction, and many others.

2.1.2. Unsupervised Learning

For this type of Machine Learning challenges, the available data entries do not have a supervisory label. The general goal of problems falling in this domain refers to discovering patterns in the supplied data [6, 53]. Unsupervised Learning contrasts with Supervised Learning, where the task was that of making predictions on new data given some labeled data from which it is possible to learn. In this case, however, predictions must be made in the absence of ground truth knowledge.

Unsupervised Learning is seen by many as the way forward in Machine Learning [35]. While a large degree of attention has been given to Supervised Learning, Unsupervised Learning has not fallen short of developments designed to tackle it. Some of the most popular methods include: k-Means Clustering, Hierarchical Clustering, Mixture Models, Anomaly Detectors, or Autoencoder Neural Networks [6, 53]. Some methods, such as the DK-means algorithm [28], have even been specifically designed for distributed environments.

The field of Unsupervised Learning can be applied to a large number of real world problems. Examples include: discovering buying patterns in consumer data, discovering hidden relationships between the individuals of a large group, finding faulty products using anomaly detectors on assembly data, and many others.

2.1.3. Reinforcement learning

Reinforcement Learning can be seen as the third major paradigm in Machine Learning. Reinforcement Learning generally models its problems as Markov Decision Processes, where an agent attempts to maximize some score function [31, 46]. Unlike Supervised Learning, Reinforcement Learning does not require labeled data, rather, it attempts to find a balance between exploring and exploiting the environment, in order to gain rewards [31]. A few of the algorithms developed to solve such challenges include: Monte Carlo Methods, Q-Learning, or SARSA.

2.2. Deep Learning

Deep Learning is a broad class of algorithms within the field of Machine Learning, which exploit *Artificial Neural Networks* (ANNs), and are applied to a large array of challenges that generally fall into either the Supervised or Unsupervised Learning categories [35], although Reinforcement Learning problems can also benefit from Deep Learning techniques.

The notion of Artificial Neural Network defines a generic architectural paradigm, which encapsulates a sizable array of possible designs and flavors. The fundamental building block of an ANN is called a *neuron* [46]. Neurons within an ANN are organized in layers such that the output of one layer is fed into the input of the next. More specifically, the output of each neuron is connected to the input of each neuron in the next layer. The initial layer, into which the input data is fed is called the *Input Layer*. The deepest layer, i.e. that which is found at the end of the Neural Network, is called the *Output Layer*. All the layers in between these two are called *Hidden Layers*. The Output Layer is where the effective values used for predictions are generated. They need to be interpreted in one way or another in order to get the predictions. Figure 2.1 exemplifies a simple, 4-layer ANN: one input layer, two hidden layers, and one output layer.

Traditionally, neural networks are subject to two phases during training: the forward pass, and the backward pass. These two phases are repeatedly applied, until some convergence criterion is met, and the network is considered to be sufficiently trained [6, 46, 53]. While the forward pass is used for making predictions on data entries, the backward pass is what actually trains the network, and is responsible with updating the parameters.

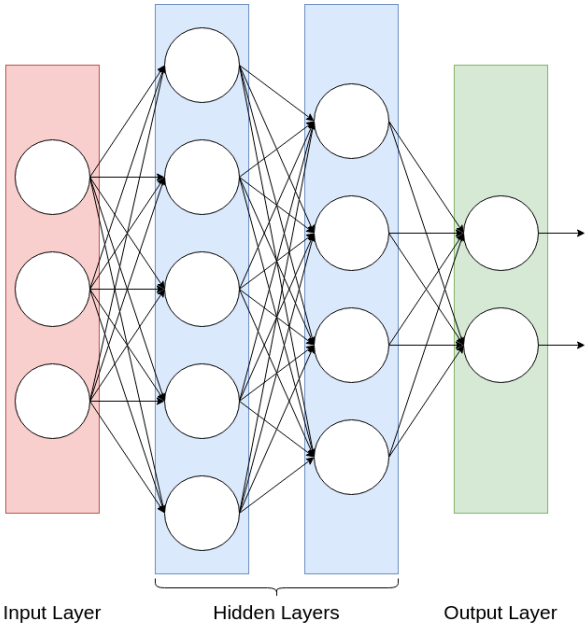


Figure 2.1: A simple example of an ANN architecture. Each colored area represents a layer. Red represents the input layer, blue the hidden layers, and green the output layer. Each circle represents a neuron. Each neuron's output is connected to all the neurons in the next layer.

2.2.1. The Forward Pass

The output of each neuron is usually produced by a non-linear function applied to the weighted sum of the previous layer's outputs. A bias term is also usually added to the input of the non-linear function. The edges in figure 2.1 can be seen as the conduits which carry and weight the outputs from the previous layer to the next layer. More formally, the previous statement can be mathematically modeled as:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i \tag{2.1}$$

Here, in_j is the input supplied to a neuron j on an arbitrary layer. a_i is the output of neuron i on the previous layer. w_{ij} is the weight of the connection between neuron j on the current layer and neuron i on the previous layer. n is the number of neurons on the previous layer. One might notice that the sum actually has $n + 1$ terms, this is due to the extra bias term $a_0 = 1$.

The weighted sum is then fed into an *activation function* g, in order to produce the output of the current neuron a_j:

$$a_j = g(in_j) \tag{2.2}$$

Function g can take many forms. Some of the most popular refer to:

$$\text{The ReLU function: } g(x) = \max(0, x) \tag{2.3}$$

$$\text{The Sigmoid function: } g(x) = \frac{1}{1 + e^x} \tag{2.4}$$

$$\text{The Hyperbolic Tangent function: } g(x) = \frac{e^{2x} + 1}{e^{2x} - 1} \tag{2.5}$$

The previously described technique is presented graphically in figure 2.2.

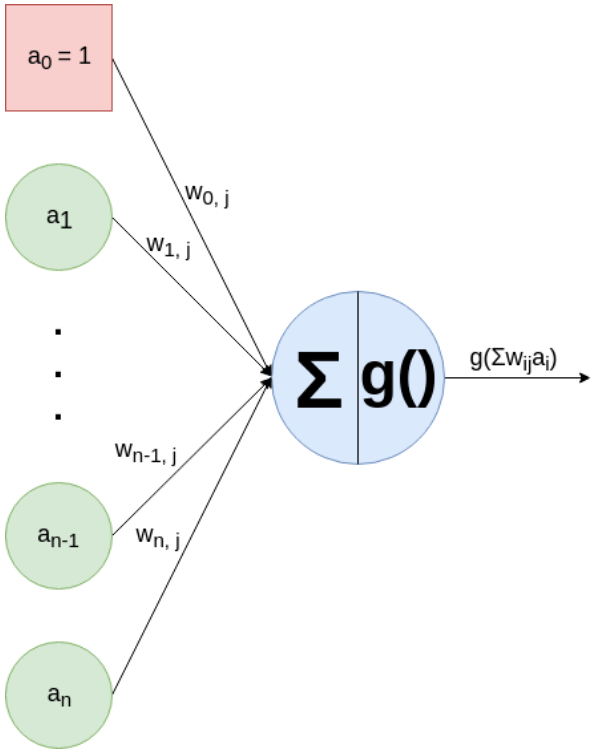


Figure 2.2: Illustration of the operations executed by a neuron during a forward pass step. The red rectangle represents the bias term, while the green circles represent the outputs of the neurons in the previous layer. The blue circle represents the current neuron.

During the forward pass, the process described in equations 2.1 and 2.2 is applied to each layer starting from the second - the first, i.e. the Input Layer, is omitted since data is fed directly into it - and ending up in the last layer, i.e. the Output Layer.

Since generally the values present at the Output Layer are unnormalized, they must usually undergo some process of normalization. A popular method employed in such cases is Softmax [6, 53]. The Softmax function receives as input a vector, and produces a probability distribution. Softmax has the following formula for producing the probability of the i^{th} element in vector z:

$$\sigma(z)_i = \frac{e^{z_i}}{\Sigma_j e^{z_j}} \tag{2.6}$$

The obtained distribution can then be used to generate a prediction.

2.2.2. The Backward Pass

The role of the backward pass in a Neural Network is that of improving the performance of the model. In order to do so, a function which indicates the relative performance of the network must be defined. Such functions are known as *Loss Functions*, and as mentioned previously, their aim is to enable the mathematical optimization of the model. There are numerous types employed in practice: Squared Error, Hinge Loss, Logistic Loss, or the Cross Entropy Loss [6, 53], and many others.

Given a labeled data entry $\langle x, y \rangle$, where x is the feature vector and y is the label associated to x, the Cross Entropy Loss can be computed as follows:

$$l(x, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{2.7}$$

Here, $\hat{y}$ is computed as:

$$\hat{y} = a(x; \theta), \text{ where } a \text{ is a function representing the Neural Network parameterized by } \theta \tag{2.8}$$

More generally, for a dataset D, the Cross Entropy can be computed by averaging the individual losses:

$$\mathcal{L}(D) = \frac{1}{n}\sum_{i=1}^{n} l(x^{(i)}, y^{(i)}), \text{ where } n \text{ is the cardinality of } D \tag{2.9}$$

With a defined loss function, it is now possible to train the Neural Network. The algorithm traditionally employed during the backward pass of a neural network is *Backpropagation* [45]. Backpropagation is an efficient algorithm highly useful in the context of Neural Networks due to its efficient use of computed gradients. It takes on a Gradient Descent approach for minimizing the loss [6, 53]. The process heavily relies on the Chain Rule and the principles of Dynamic Programming in order to reuse previously computed gradients, such that the computation of gradients for the parameters of an arbitrary layer depends solely on the gradients computed previously for the next layer in the network.

Gradient Descent based approaches try to minimize the loss by moving the values of the parameters in the opposite direction of the gradient, thus minimizing the value of the loss function. Formally, this can be modelled mathematically as:

$$W' = W - \eta \frac{\partial \mathcal{L}(D)}{\partial W} \tag{2.10}$$

Here, W represents the model's set of parameters, W' the updated parameters,and η represents the *learning rate*. The learning rate controls the degree of change applied to the parameters. Intuitively, high values change the parameters considerably but are less precise in terms of convergence, while small values are more precise, but exert little change over the parameters.

As previously mentioned, Backpropagation is a means of efficiently computing the gradient $\frac{\partial L(D)}{\partial W}$ by exploiting Dynamic Programming and the Chain Rule. To gain a better understanding of it, it is better to focus on the computation of the gradient along the dimension of one specific parameter in the network for an arbitrary data sample. For the arbitrary parameter w_{ij} and sample $\langle x, y \rangle$ the following would be obtained via differentiation:

$$\frac{\partial l}{\partial w_{ij}} = \frac{\partial l}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial l}{\partial a_j}\frac{\partial a_j}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} \tag{2.11}$$

The in_j term is computed as presented in equation 2.1. The last term, $\frac{\partial in_j}{\partial w_{ij}}$, may be simplified to:

$$\frac{\partial in_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_{k=0}^{n} w_{kj}a_k = a_i \tag{2.12}$$

The second term, $\frac{\partial a_j}{\partial in_j}$, can be rewritten as:

$$\frac{\partial a_j}{\partial in_j} = \frac{\partial g(in_j)}{\partial in_j} \tag{2.13}$$

In the general case, the first term is more complicated to compute. Let $\mathcal{R} = \{r_1, \dots, r_m\}$ be the indices of the nodes which receive input from node j. Then, the following is obtained:

$$\begin{aligned} \frac{\partial l}{\partial a_j} &= \frac{\partial l(in_{r_1}, \dots, in_{r_m})}{\partial a_j} \\ &= \sum_{r\in\mathcal{R}} \frac{\partial l}{\partial a_r}\frac{\partial a_r}{\partial in_r}\frac{\partial in_r}{\partial a_j} \\ &= \sum_{r\in\mathcal{R}} \frac{\partial l}{\partial a_r}\frac{\partial a_r}{\partial in_r} w_{jr} \end{aligned} \tag{2.14}$$

In case the current neuron j is on the Output Layer, then the first term can be computed in a more straight-forward way:

$$\frac{\partial l}{\partial a_j} = \frac{\partial l}{\partial \hat{y}} = \frac{\partial l(x,y)}{\partial g(\hat{y})} \frac{dg(\hat{y})}{d\hat{y}} \tag{2.15}$$

Equation 2.11, after substituting equations 2.12, 2.13, 2.14, and 2.15 in, becomes:

$$\frac{\partial l}{\partial w_{ij}} = \frac{\partial l}{\partial a_j} \frac{\partial a_j}{\partial in_j} \frac{\partial in_j}{\partial w_{ij}} = \frac{\partial l}{\partial a_j} \frac{\partial a_j}{\partial in_j} a_i \tag{2.16}$$

Finally, to summarize, the partial derivative $\frac{\partial l}{\partial w_{ij}}$ can be computed as:

$$\frac{\partial l}{\partial w_{ij}} = a_i \delta_j \tag{2.17}$$

Where:

$$\delta_j = \begin{cases} \frac{\partial l(x,y)}{\partial g(\hat{y})} \frac{dg(\hat{y})}{d\hat{y}} & \text{if } j \text{ is an output neuron} \\ \frac{dg(a_j)}{da_j} \sum_{r \in \mathcal{R}} w_{jr} \delta_r & \text{if } j \text{ is an input neuron} \end{cases} \tag{2.18}$$

2.2.3. Overview on Neural Network Training

Traditionally, the process of training a Neural Network using Stochastic Gradient Descent is structured into *epochs* and *batches*. An epoch can be seen as processing the entire training dataset once, while batches can be seen as manageable workloads of a fixed number of data samples, into which an epoch is divided. After each batch, Backpropagation is applied on the Neural Network in order to update its parameters.

Although fundamentally the same, Gradient Descent can take on different names based on the batch size [7, 44]:

1. **Batch Gradient Descent**: the size of a batch is equal to that of the training set itself.
2. **Mini-batch Gradient Descent**: the size of a batch is greater than one sample, but smaller than the size of the training dataset.
3. **Stochastic Gradient Descent**: the size of each batch is exactly one sample.

It is often the case that Gradient Descent, and Stochastic Gradient Descent are used to refer to **Mini-batch Gradient Descent**, which is the most popular version applied in practice [44].

Regardless of the batch size, Gradient Descent takes on the following generic formula, which was in fact presented in equations 2.9 and 2.10 for the Cross Entropy loss:

$$W' = W - \eta \frac{1}{n} \sum_{i=1}^{n} l(x^{(i)}, y^{(i)}) \tag{2.19}$$

Here n is the size of the batch, and l is the loss function. It has been shown that given the currently available hardware, the best batch sizes, based on generalization power, is between 2 and 32 [37]. Common sizes used in practice include: 32, 64 or 128. While the number of epochs can vary from the tens to the thousands.

Ideally speaking, the optimal batch size is one, as each sample has maximal contribution towards optimizing the network. However, the overhead of applying backpropagation after each data entry is impractical. Conversely, a batch size as large as the training set averages over too many entries, and consequently leads to a significant amount of lost information.

The entire process of training and evaluating a Neural Network relies on three datasets: for *training*, *testing*, and *validation*. The training dataset is self-explanatory, being used for optimizing the performance of the Neural Network during the training phase. The validation

dataset is used periodically during the training phase to measure the generalization performance. It is also used for for adjusting hyperparameters such as the number of layers or neurons per layer. Finally, the training dataset is used to give an indication of the generalization capacity after the training phase is complete. Unlike the validation set, which serves a similar purpose, the training set is never exposed to the Neural Network during the training phase, in the hopes that it will provide an accurate as possible indication of the network's performance.

2.2.4. Popular Neural Network Categories

Neural Networks can be classified into two major types, based on the direction in which their connections are pointing:

- **Feedforward Neural Networks**: are relatively straight forward. The connections present in this kind of network are exclusively forward facing.
- **Recurrent Neural Networks**: have both forward and backward facing connections. Such networks are usually employed when there is a need of past information in order to make correct predictions.

A brief introduction into some of the subtypes which fall into these categories are further presented.

Feedforward Neural Networks

Feedforward Neural Networks encompass a large array of architectures. Some of the most popular designs falling in this category are:

- **Fully Connected Feedforward Neural Networks**: wherein the output of each neuron is connected to the input of each neuron in the next layer. The forward propagation step does not employ any remarkable operation, in addition to the activation function, other than atomic multiplication and addition.
- **Fully Convolutional Feedforward Neural Networks**: are usually employed in challenges involving images. They exclusively make use of so called convolutional layers, which apply convolutions [18] on the input data. Convolutions are exploited for their ability to discover translation invariant patterns.
- **Convolutional Feedforward Neural Networks**: feature a combination of both fully connected and convolutional layers. Traditionally, the fully connected layers are applied towards the end of the network, while the other layers are convolutional.

Recurrent Neural Networks

Recurrent Neural Networks are generally used in challenges where successful solutions benefit from aggregating data over time, or employing some form of memory that makes it possible to access past information. Such networks can have layers where in addition to propagating the outputs to the next layer, they are also fed back into their inputs. Examples of relevant tasks include: video classification or speech recognition. Some popular designs, which fit into this group refer to:

- **Classical Recurrent Neural Networks**: the outputs of recurrent layers are weighted and trivially fed back into the layer's own input. The recurrent and forward weights are disjoint. Such networks are susceptible to vanishing or exploding gradients, and generally have a limited memory.
- **Long Short Term Memory Networks**: make use of a more complicated architecture, wherein the recurrent layers have three types of gates: *input*, *output*, and *forget*. Such networks are not susceptible to vanishing or exploding gradients, and have a longer memory.
- **Gated Recurrent Unit Networks**: are highly similar to Long Short Term Memory Networks, however, no output gates are used in the recurrent units, making them simpler.

It is possible for these methods to exploit the convolution operation.

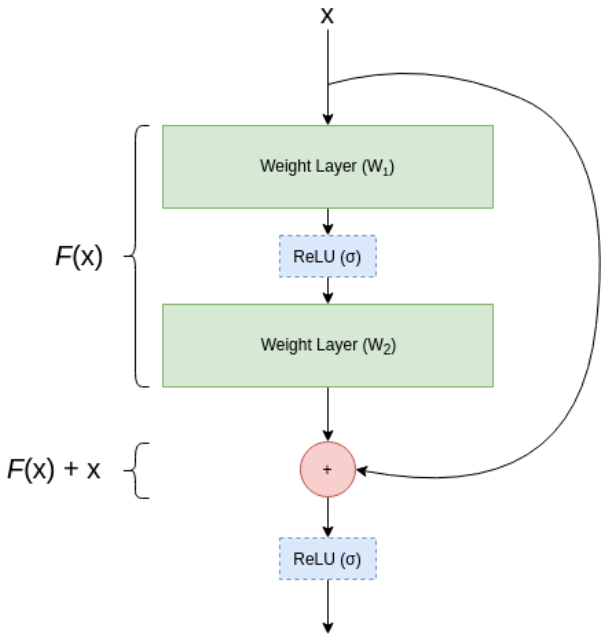


Figure 2.3: A two layer fundamental building block of ResNet Neural Networks. Diagram inspired by [22].

2.3. The ResNet Architecture

Depth has been shown to be essential to the performance of Neural Networks [22, 23]. This might not be surprising considering the first layers of a Neural Network learn more primitive patterns, while the latter levels learn increasingly higher-level structures [35]. For instance, given the task of recognizing images with cars, a Convolutional Neural Network tends to learn more primitive configurations, such as lines, triangles, or circles, at the early layers. The middle layers generally learn mid-level abstractions, such as vague structures that resemble some of the shapes which make up a car. Finally, the latter layers of the network should learn patterns that resemble actual cars. As a consequence, it might seem intuitive to add as many layers as possible to Neural Networks in order to exploit this property, this, however, is highly unlikely to produce the desired effects due to a couple of reasons: the problem of vanishing and exploding gradients - which has been discussed in Chapter 1, and which has generally been solved -, and the empirically demonstrated issue of reduced accuracy, and higher training error [22].

ResNet is a generic Neural Network architecture proposed for the very reason of exploiting very deep Neural Networks and of overcoming the well-known performance issues of trivially stacked deep Neural Network architectures. Since the architecture is generic, instances of the ResNet Neural Network paradigm can span from tens to thousands of layers, with popular examples including: 50 layers, 101 layers, or 152 layers.

At its very foundation, a ResNet instance makes use of the structure presented in figure 2.3 [22].

This structure is generic, and may be customized to contain a small, but arbitrary number of layers. In this specific example, it consists of two *Weight Layers*, which simply apply linear transformations on their inputs, two *ReLU* nonlinearities, and a skip connection with addition against the input x itself. This structure is repeated throughout the network. Its purpose is to enable a mapping from x to $\mathcal{H}(x)$, which is modeled as $\mathcal{H}(x) = \mathcal{F}(x) + x$. $\mathcal{F}(x) = \mathcal{H}(x) - x$ is known as the residual function [22, 23], and is formally implemented as:

$$\mathcal{F}(x; W) = W_2\sigma(W_1 x) \tag{2.20}$$

Here, W represents the block's parameters with $W = \{W_1, W_2\}$. σ is the ReLU function. The complete output of the block presented in figure 2.3 is:

$$\mathcal{H}(x) = \sigma(\mathcal{F}(x; W) + x) \tag{2.21}$$

It should be noted that ResNet building blocks can also contain Convolutional Layers, and need not limit themselves to Fully Connected layers exclusively [22].

A ResNet network may be trained using standard, well-known methods such as Stochastic Gradient Descent [44] and Backpropagation [22]. Finally, although significantly deeper than traditional Neural Networks, ResNets do not have a high complexity, and are viable from a computational standpoint [22].

ResNets have been empirically shown to perform better than shallower architectures, an ensemble of such networks winning the ILSVRC 2015 competition with an error of 3.57% [22]. A concrete example of a 34-layer ResNet is presented in figure 2.4. The network mostly uses 3×3 convolutions. When the feature map size is halved, the number of filters is doubled, such that the complexity throughout the network remains constant. A max pooling operation is used after the first convolution, and an average pooling operation is used after the last one [22]. Finally, a fully connected layer with Softmax [6, 53] outputs the probabilities.

2.4. Paradigms for Distributed Machine Learning

Enabling Large Scale Machine Learning is a highly non-trivial task, which requires the existence of a powerful hardware infrastructure that can support advanced Machine Learning frameworks. In turn, the Machine Learning frameworks should make the aforementioned large scale challenges viable from an implementation standpoint. As scaling out is normally seen as a cheaper alternative to scaling up, due to the ubiquitous nature and low cost of commodity hardware [30], distributing such a task over a cluster of nodes is commonplace. Introducing distribution, however, adds a new level of complexity to the algorithms used for training in Machine Learning, because of the need for nodes to communicate and synchronize. Consequently traditional methods used for training on, for instance, Neural Networks, such as Stochastic Gradient Descent and Backpropagation, require changes in order to support such scenarios. These changes are then implemented in the frameworks themselves as the last step towards making Distributed Large Scale Machine Learning a reality.

2.4.1. Types of Parallelism

There are three major paradigms for enabling distributed Machine Learning: *Data Parallel*, *Model Parallel*, and *Data+Model Parallel* models [30, 39, 56].

Data Parallel

In the context of the Data Parallel model, each distributed worker will be in possession of the full model, while the data itself will be divided among the workers [30]. Each worker has access to the model, either through replication or through some central repository [30]. Workers are tasked with processing the data they are given. Intuitively, training each worker on a different set of data will lead to homologous parameters in different workers, whose values may be vastly different from one another. Thus, some form of synchronization is generally necessary during the course of training. Figure 2.5 shows a simple diagram of the data-parallel model. The data is split disjointly into as many chunks as there are workers. In the end, or during the training process, a final model is created, by aggregating together the individual models in each worker.

Model Parallel

The Model Parallel paradigm refers to splitting up the trained model across the available workers. In such a context, each node will be responsible with training a part of the model, and consequently a part of the model's parameters [30]. Each worker will employ the entirety of the available dataset during training. The model should be divided such that the dependencies between any two segments is minimized, as to reduce the amount of communication necessary between workers. In the case of this approach, the final trained model can usually be rebuilt by putting together all the segments trained in the workers. Figure 2.6 shows a possible configuration for Model Parallelism. All workers have access to the entire dataset, but each worker is in charge of a different part of the model.

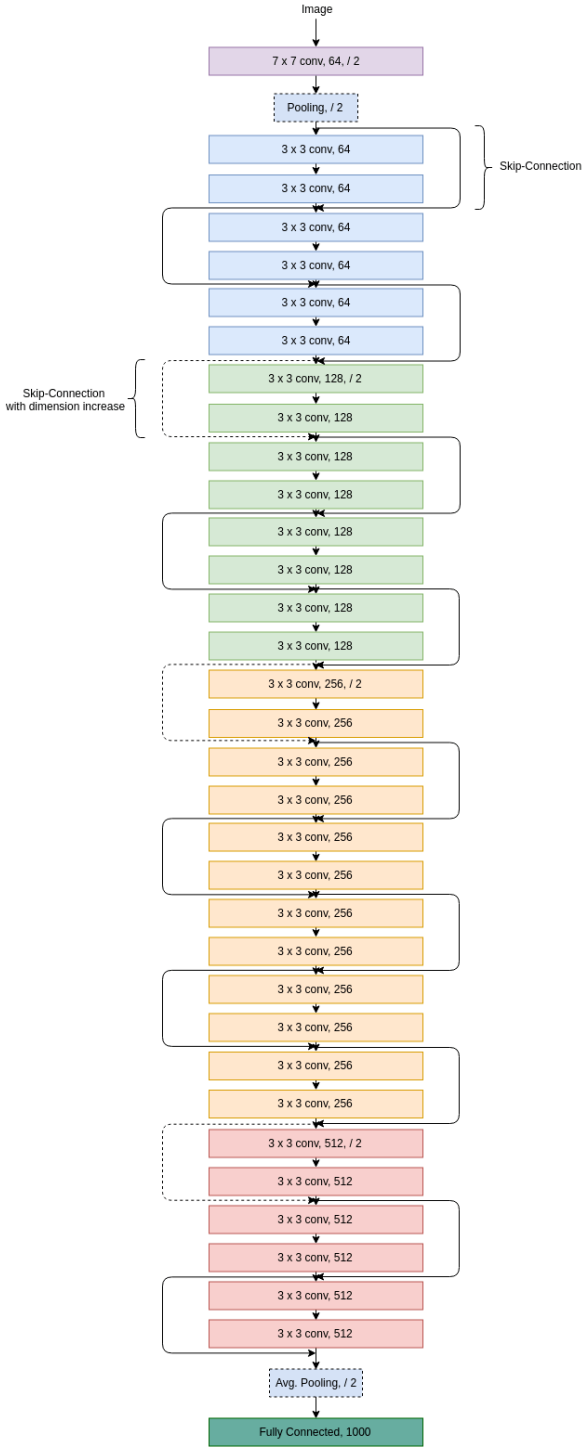


Figure 2.4: A diagram which depicts a 34-layer ResNet. The numbers present in the convolution boxes represent the amount of filters. $/2$ represents a halving of the feature map, and a doubling of the filter number. Dotted skip-connections are employed to accommodate the aforementioned doubling. Diagram inspired by [22].

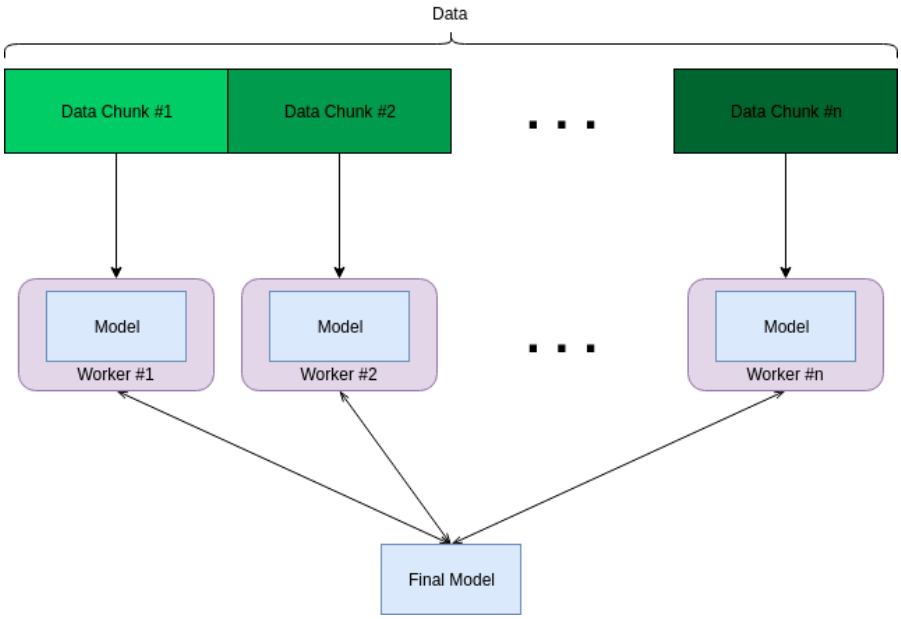


Figure 2.5: A diagram exemplifying the Data Parallel Distributed Learning paradigm. The data is split into as many chunks as there are workers. Each worker has access to the model being trained. A final model is being created either continuously during training, or at the end.

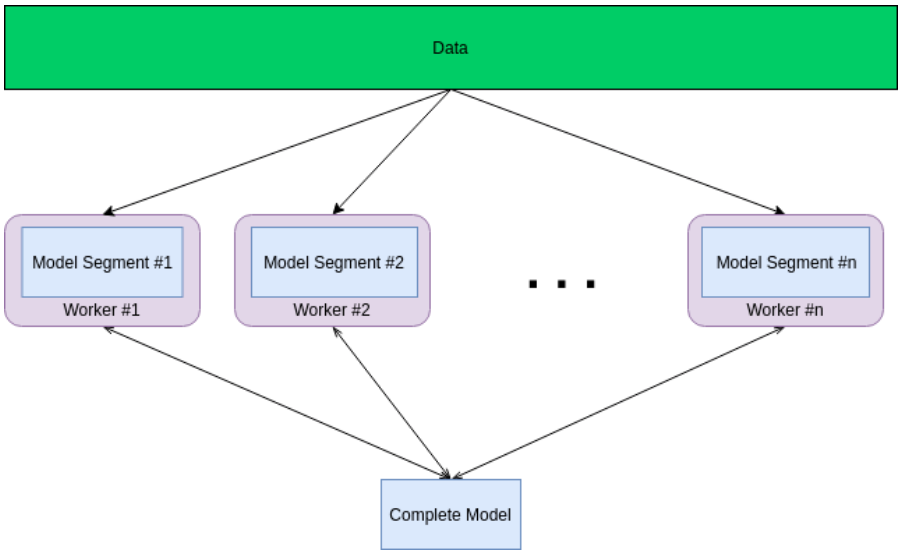


Figure 2.6: A diagram exemplifying the Model Parallel Distributed Learning paradigm. The model is split into as many chunks as there are workers. Each model has access to the same data being used for training. A complete model is being created either continuously during training or at the end.

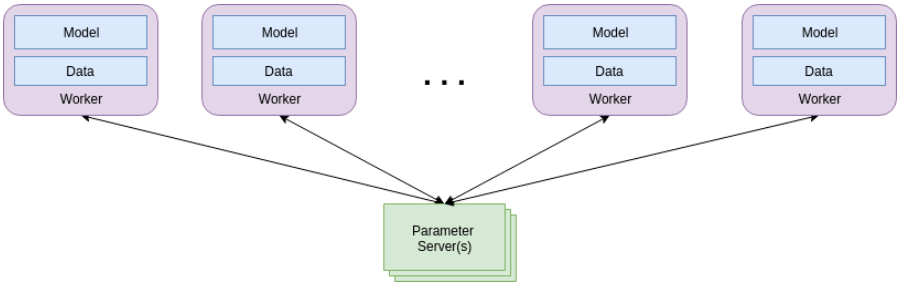


Figure 2.7: Shows a simple Parameter Server architecture. The workers process their part of the workload, and communicate the gradients to the parameter server. The parameter server(s) will aggregate the incoming gradients, and communicate back the results to the workers.

Data+Model Parallel

The Data+Model Parallel procedure merges both the Model and Data Parallel approaches together. In this architecture, both the model and the data are split among the nodes. An example of such a strategy was presented in Google's DistBelief framework, under the name *Sandblaster L-BFGS* [16], wherein each worker would be in possession of the model, and would have access to the entire dataset. The task of training the model would be split into a very large number of subtasks, much larger than n, where n is the number of workers. It follows that each worker would be assigned a subtask as soon as it would finish its previous one [16].

2.4.2. Topologies

Topologies refer to the structure and hierarchy employed towards organizing the workers of a cluster. In the context of Distributed Machine Learning, they are required towards enabling the exchange of gradients or other information between nodes in an efficient fashion. There are generally three major types in use today [30]:

- **Centralized**: in such topologies, there is a central, singular site within the architecture, where the gradients from the workers are collected and aggregated.
- **Decentralized**: such topologies have a designated set of locations, potentially organized in a hierarchy, wherein parameters are gathered and processed. The aforementioned sites communicate with one another as needed, in order to compute the final result. Such architectures are often employed in the context of collective communication algorithms such as Allreduce or Allgather, and feature shapes such as trees or rings [41, 42, 52].
- **Fully Distributed**: such systems do not rely on any set of specially designated locations for gathering and processing information, rather, workers communicate as needed in order to train, and implicitly update the model.

A popular architecture currently in use in many Machine Learning frameworks [2, 2, 9, 14, 16, 25, 61, 62] is the *Parameter Server* [30, 36]. The Parameter Server architecture falls within the Decentralized topology. Figure 2.7 shows a simple example of a Parameter Server architecture. In the context of Gradient Descent, each node will be assigned some workload during each batch. When a worker finishes its computations, it will send the local gradients to the parameter server. Once the parameter server has all the intermediate gradients, it aggregates the data, and sends back the final results. The architecture may feature one or more parameter server instances, usually, each being designated to handle some partition of the model's parameters.

The Peer-to-Peer architecture falls within the Fully Distributed topology. This architecture is comprised solely out of logically identical peers. There are no special nodes with different or permanent additional responsibilities. The nodes communicate freely and directly with one another, in order to train the model. Figure 2.8 exemplifies such a topology.

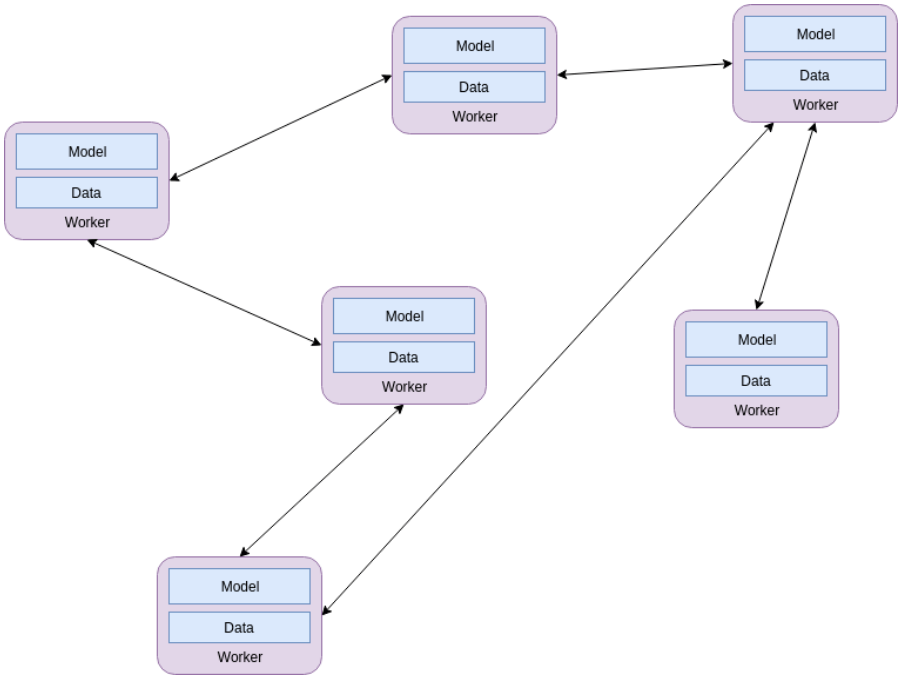


Figure 2.8: Shows a simple Peer-to-Peer architecture. The workers communicate with each other as they see fit. No worker is assigned different responsibilities all throughout the training phase. Workers can temporarily have special tasks, such as introducing or synchronizing nodes.

2.4.3. Distributed Execution Models

Finding the correct balance between communication and computation in the context of distributed Data Processing is a difficult task [30]. Generally speaking, during distributed computation, nodes need to regularly communicate and synchronize in order to produce accurate results. Communication and synchronization is, however, not without a cost, as it takes time away from computation and lengthens the wall-clock time required to complete the computation. It is thus easy to see that there is a trade-off between accuracy and execution time. In Machine Learning, highly frequent synchronization is generally impractical due to the large volume of the datasets employed and the computationally intensive nature of the algorithms. Fortunately, iterative convergence algorithms, such as gradient-based methods, are quite robust to errors stemmed from parameter staleness [13], which often allows for generous periods of computation before any communication is necessary.

In literature there are 4 predominant types of distributed execution models [30]: Bulk Synchronous Parallel, Stale Synchronous Parallel, Approximate Synchronous Parallel, and Totally Asynchronous Parallel. Such methods discretize the execution into units called *clocks*. A clock is further divided into one or more *iterations* [13]. The semantics of an iteration depend on the nature of the distributed computation. In distributed Deep Learning, an iteration generally refers to a batch. After one or more clocks, depending on the strategy, synchronization barriers are utilized, which allow communication to happen.

Each of the four types of execution models are further detailed in the sections that follow.

Stale Synchronous Parallel

The Stale Synchronous Parallel model introduces a slack variable t which defines an acceptable delta between the fastest and slowest workers, in terms of completed clocks. When the delta becomes greater than the allowed slack t, the fastest nodes are forced to stop, so as to let the slower workers catch up. Nodes commit their parameter updates at the end of each clock. Consequently, faster nodes can operate on stale parameters [13, 30]. The slack must be chosen such that the error introduced by the staleness is acceptable.

Bulk Synchronous Parallel

In the Bulk Synchronous Parallel model, a synchronization and communication barrier is placed at the end of each clock [13, 30]. In other words, before any of the workers can move on to clock $n+1$, all of the workers must have finished clock n. At the beginning of each new clock every worker should have the same view over the model's parameters. The considerable disadvantage to this strategy is that at every clock, the entire computation must wait for the slowest node. While at smaller scales, this might be less of an issue, straggler nodes occur with a high probability when the number of nodes is high. Bulk Synchronous Parallel is in fact a specific case of Stale Synchronous Parallel when the slack variable is $t = 0$.

Totally Asynchronous Parallel

The Totally Asynchronous Parallel model allows workers to proceed with their part of the computation without ever having to wait for straggler processes to catch up. This model generally yields the greatest speedup, however, the model may suffer large accuracy penalties due to the lack of barriers which synchronize the workers, and implicitly limit the staleness of the parameters [30]. The Totally Asynchronous Parallel model is a particular case of the Stale Synchronous Parallel strategy where the slack variable is $t = \infty$.

Approximate Synchronous Parallel

The Approximate Synchronous Parallel model gauges the need for parameter updates based on the the amount of change incurred on parameters [30]. More specifically, a parameter will be globally updated when it has changed sufficiently (given a threshold) relative to its global value. Deciding which parameters are worth tracking and what their respective thresholds are is where the crux of this strategy lies [30].

Stale Synchronous Parallel, Bulk Synchronous Parallel, and Totally Asynchronous Parallel are quite often encountered in literature [14, 36, 55, 56]. Generally speaking, it is considered that in the presence of a small degree of stragglers the Bulk Synchronous Parallel model is more suitable, while the Stale Synchronous Parallel model produces better results when the amount of straggler processes is more significant [13].

There is currently no general consensus regarding whether or not Synchronous methods are always significantly better than Asynchronous ones when straggler processes are present, in terms of model accuracy and generalization performance. A number of studies suggest that the Synchronous models consistently outperform Asynchronous ones [9, 13], while others propose methods such as adjusting the learning rate by a measure of the staleness, through which the difference in performance is not significant [62].

There is no shortage of systems which employ one or more of these aforementioned strategies towards making the task of Large Scale Machine Learning possible. Some of these frameworks are presented in more detail in section 2.5.

2.5. Large Scale Machine Learning Frameworks

With the soaring popularity of Machine Learning, it is only natural that dedicated frameworks would be developed and made freely available for the Computer Science community to use. Currently, there is no shortage of systems which fall into this category. Some popular examples include: TensorFlow [1, 2], Caffe2 [50], Caffe [29], Microsoft CNTK [47], MXNet [10], Petuum [56], Torch [11], Ako [55], GeePS [14], DryadLINQ [59], or MLbase [32]. In certain cases, further developments have been made to existing data processing frameworks in an effort to extend their domain of applicability to Machine Learning. For instance, MLlib [39] is a scalable library for Machine Learning meant to be used with Apache Spark [60]. Similarly, efforts have been made by the Computer Science community to further develop existing frameworks, dedicated to Machine Learning. For example, Uber's Horovod [48] is an extension to TensorFlow, developed in an effort of minimizing both the overhead of multi-GPU communication via the NVIDIA Collective Communications Library and the coding effort required for implementing such multi-GPU scenarios. Some of the more popular and interesting frameworks are presented in the sections that follow.

2.5.1. DistBelief

DistBelief was developed in an effort to compensate for the well known limitation of GPU-based training wherein the process becomes highly inefficient when the model does not fit entirely in the GPU's memory [16]. Previous solutions to this referred to reducing the model's parameter count or using less data. DistBelief, however, seeks to solve this problem through distribution, and boasts the ability to train large models with billions of parameters on top of thousands of nodes having a total of tens of thousands of compute cores, in a relatively transparent fashion to the programmer. DistBelief is not an open-source project, nor is it available to the wide Computer Science community. It is an internal system exclusively developed within Google.

DistBelief exploits both the data and model paradigms of parallelism. At the implementation level, both intra-machine parallelization, via threading, and inter-machine parallelization, via message passing, are employed. DistBelief also makes use of an explicit sharded multi-instance Parameter Server architecture, where each instance is responsible with storing and updating a partition of the model's parameters. DistBelief employs a dataflow graph representation for its models which usually consists of a few complex layers [2, 16].

Two novel distributed optimization methods are proposed and used in DistBelief: Downpour Stochastic Gradient Descent, and Distributed L-BFGS.

Downpour Stochastic Gradient Descent

As its name suggests, Downpour Stochastic Gradient Descent is a version of the Mini-batch Stochastic Gradient Descent optimization technique. Via data parallelism, the training dataset is divided into shards, which are distributed among the workers. Each worker holds a replica of the model. This algorithm employs the Asynchronous Parallel model, since it exploits asynchronicity along two dimensions: the parameter server instances run independently of one another, and the workers run independently of one another as well. Communication is employed between the workers and the parameter server, in order to send local gradients to the parameter server instances, and conversely, retrieve the updated model parameters. The action of pushing the local gradients is controlled by the hyperparameter n_{push}, which indicates the number of batches to process before sending the local gradients. Similarly, n_{fetch} is the hyperparameter which indicates the number of batches required to process before updated parameters can be retrieved from the parameter server. The simplest version of the algorithm is when both parameters are $n_{push} = n_{fetch} = 1$. Due to the lack of synchronization between workers and parameter servers, there is a high probability that processes will each have a different view of the model. Regardless, experiments show that Downpour Stochastic Gradient Descent scales very well [16], and provides a large degree of speed-up when paired with AdaGrad [44].

Sandblaster L-BFGS

Sandblaster L-BFGS employs a special process called *Coordinator* which holds the logic for the L-BFGS algorithm. The Coordinator does not have access to the model's parameters. Rather, it is tasked with issuing commands to the workers and the parameter server via small messages. As its name suggests, the central optimization algorithm is L-BFGS. Work is divided into a large number of chunks, much greater than the number of workers, a subset of which being initially assigned to the workers. As soon as a worker finishes their chunk, they get assigned another. This way, efficient nodes do more work, and as such, compensate for straggler processes. Experiments show that Sandblaster L-BFGS makes efficient use of the available bandwidth, and has a high potential in terms of scalability [16].

2.5.2. TensorFlow

TensorFlow is one of the most powerful and popular Machine Learning frameworks publicly available today. Developed by Google, it is their second major framework release dedicated to Machine Learning, after DistBelief. A version of TensorFlow is openly available to the public[1].

[1] TensorFlow GitHub repository: `https://github.com/tensorflow/tensorflow`

TensorFlow was developed as a response to the inherent limitations of DistBelief, which largely stem from its lack of versatility [2]. Some of the major disadvantages of DistBelief that TensorFlow is trying to address include:

- The development of novel layers can be difficult for researchers who are more familiar with scripting languages such as Python, and not C++, which is the main development language of DistBelief.
- The large amount of effort required towards refining training and optimization methods, which often demand changes to the parameter server's implementation. Furthermore, the dedicated Parameter Server architecture is restrictive and may not be suitable for all types of optimization methods.
- Due to the strict execution pattern followed by the workers, i.e. that of: reading data, performing the forward pass, backward pass, and exchanging parameter updates with the parameter server, DistBelief is not able to support other sorts of execution flows. For instance, those involving loops, such as in Recurrent Neural Networks, alternative training phases, such as those required in Autoencoders, or making use of an external loss function as is the case in Reinforcement Learning.
- DistBelief is expressly designed for large scale data processing scenarios, involving up to tens of thousands of computational cores. It is, however, limited when running learning challenges on different types of platforms, such as mobile devices, or single machines.

Similar to DistBelief, TensorFlow makes use of the dataflow abstraction for representing its models. Unlike DistBelief, however, where models were represented as a small set of complex layers, TensorFlow chooses to represent models as a collection of primitive operators [2], such as addition or matrix multiplication. This approach provides TensorFlow with a large degree of flexibility.

TensorFlow's execution process is defined in terms of two distinct phases. The first refers to describing a symbolic dataflow graph with placeholders for the input data and the model's parameters. The second phase refers to running an optimized a version of the aforementioned graph, with operations bound to devices, based on heuristics which try to achieve as good of a performance as possible, given the available hardware and execution context. This process can be done transparently by TensorFlow, or for more complex scenarios, it is possible for the programmer to make suggestions regarding which devices should run which operations.

Unlike DistBelief, models developed in TensorFlow do not make use of an explicit Parameter Server architecture. To gain flexibility, a TensorFlow execution makes use of a set of *tasks*, which interact with one another as needed. There are two major types of tasks: *worker* and *parameter server*, although a third type, called *controller* is also employed in select parameter update strategies such as Distributed Allreduce. By making this abstraction, i.e. that of using generic tasks, which are then assigned specific roles, TensorFlow avoids using a dedicated parameter server. From the set of pre-implemented update methods, the parameter server paradigm is the most frequent [8].

Defining new architectures, with novel layers or training algorithms is made possible in TensorFlow with the help of the dataflow abstraction and the introduction of control-flow constructs such as loops or branches. New components can essentially be seen, and defined as a subgraph of the larger dataflow graph.

One of TensorFlow's major aims was to enable a large degree of portability for its models. For instance, once a model is completely developed using TensorFlow, it should be ready to run on virtually any type of hardware, ranging from small mobile devices, to something as large as a cluster with hundreds of nodes[2]. In order to do this, TensorFlow heavily relies on its dataflow graph abstraction. The operators within the graph are concretely implemented for a specific type of hardware component via a *kernel* [1, 2]. More specifically, each type of operator can have a set of associated kernels, each dedicated for one type of device. Moreover,

[2]Changing non-distributed code to distributed code may require some amount of modifications, however, it is usually not considerable.

communication between devices is transparently handled by TensorFlow, which introduces communication nodes in the dataflow graph. The nodes represent concrete, device specific implementations for sending or receiving data. With these features, it is possible for the programmer to write the implementation once, and run it on virtually any kind of hardware, as long as there is a kernel for it.

The core runtime of TensorFlow is implemented in C++ due to its efficiency. It also offers two clients, one in Python, and the other in C++, which are connected to the C++ core via a C API[3]. The clients are written in Python and C++ respectively due to the languages' popularity in relation to TensorFlow's user base [2].

Parameter Update Strategies in TensorFlow

TensorFlow features a large range of pre-implemented variable update strategies: Parameter Server, Replicated, Distributed Replicated, Independent, Distributed Allreduce, Collective Allreduce and Horovod. Each of these strategies is further detailed in what follows:

- **Parameter Server**: execution is divided among worker and parameter server processes. Parameter server processes store the master copies of the parameters, and apply gradient updates on them after each batch. Each worker process retrieves a master copy of the parameters from the parameter server processes, and uses them in the current iteration. Worker processes then send the local gradient updates to the parameter server processes, after each completed iteration.

- **Replicated**: refers to single node computation only. Only intra-machine parallelism is supported for this strategy. Each device stores a copy of the model's parameters. There is no master copy. After each iteration in the optimization algorithm, some cross-device version of Allreduce is employed to update and synchronize the model's parameters.

- **Distributed Replicated**: unlike *Replicated*, this strategy allows for multi-node distributed training. Moreover, it employs a parameter server process, which handles the parameter updates. Similarly to *Replicated*, each device has its own copy of the parameters, and refreshes them when the parameter server has been updated using all the required gradient updates from the workers.

- **Independent**: is highly similar to the *Replicated* strategy. It refers to single node computations only. Each node has a copy of the model's parameters. Unlike *Replicated*, however, the gradients are not shared among the devices, hence no synchronization or updates are performed. This strategy is generally employed towards measuring the raw performance of GPUs in the absence of communication.

- **Distributed Allreduce**: allows for multi-node distribution. It makes use of no parameter servers, rather it uses a *controller*. It employs the Allreduce algorithm in order to reduce the gradients and generate the parameter updates. If the Distributed Allreduce strategy is only used on a cluster of one node, then this will be equivalent to using *Replicated*. This strategy allows for the exact specification of the implementation of the reduction algorithm. For this study, the following two strategies are employed:

  - **PSCPU**: each distributed tensor is copied in one of the participating worker's CPU. The reduction process takes place at the aforementioned CPU. The result is distributed back to all the participating device.
  - **xRing#1**[4]: one global Ring Allreduce is employed towards reducing the distributed tensors. This contrasts, for instance, with xRing#2, where two simultaneous Ring Allreduce runs are executed, each designated with a half of each distributed tensor.

- **Collective Allreduce**: devices run completely in parallel, except during variable initialization and gradient reduction, which is done via the collective Allreduce algorithm.

[3] API = Application Programming Interface

[4] For the sake of simplicity in notation, xRing#1 is referred to as Ringx1 throughout the thesis.

This architecture does not employ any parameter server process. Additionally, it requires that the group of processes has a size greater than 1. This implicitly means that at least two nodes are required for this strategy.

- **Horovod**: this strategy employs the Horovod extension developed by Uber. Each worker makes use of a single GPU, and uses either MPI or the NVIDIA Collective Communications Library in order to reduce the gradients.

It should perhaps be emphasized once more that not all architectures support multi-node distribution. The architectures that do are the following: Parameter Server, Distributed Replicated, Distributed Allreduce, Collective Allreduce and Horovod. Replicated and Independent are targeted towards single node computation, with intra-machine parallelism.

2.5.3. Caffe and Caffe2

Caffe was initially developed at the University of California, Berkeley, within the Berkeley Vision and Learning Center, in order to provide a modular framework for conventional Convolutional Neural Network applications, intended to be used on single node execution contexts. In theory, it provides true support for off the shelf models, as well as powerful features which allow for the development of novel Deep Learning techniques for Computer Vision [29]. Shortly after its release, however, it started to be used in a wider range of applications, such as: neuroscience, astronomy, robotics, or speech recognition [29].

The primary building blocks, of Neural Networks created in Caffe, are called *layers*. Layers are designed to implement the essential operations for Convolutional Neural Networks, such as: the convolution operations, pooling, activation functions, or the inner product [29], among others. Such layers are built in a bottom-up fashion, which should in theory allow novel layers to be easily developed, by reusing the pre-built lower level components. Conceptually speaking, they are not considerably different from Neural Network layers.

Another essential concept in Caffe are the *blobs*. Blobs are 4 dimensional arrays which are used to store and communicate data [29]. Blobs are employed by layers both as input and output, where one or more such instances may be required or returned respectively. Stochastic Gradient Descent paired with Backpropagation is the algorithm of choice for training Neural Networks in Caffe.

Caffe is relatively limited in terms of supported levels of distribution and portability [50]. Multi-node distribution is not supported, and the only level of parallelism supported is intra-machine via multiple GPUs. That being said, the latter type of parallelism requires relatively homogeneous hardware devices. More specifically, when there is a set of non-homogeneous GPUs available for computation, Caffe will be bounded in performance by the weaker card. In some cases, if the cards differ vastly in terms of performance, the distribution might be completely impossible, or driver issues might occur [43].

Caffe2 has been developed in an effort to overcome Caffe's limitations, provide support for the growing number of novel architectures and techniques forwarded in the field of Machine Learning, as well as for the increasing need of portability and large scale system support [50]. Moreover, in order to provide a more expansive level of support, for both research oriented models and industry grade models, Caffe2 has been merged into PyTorch. PyTorch is more research oriented, whereas Caffe2 is focused more on production-level deployments, with a focus on mobile devices [50]. Caffe2 is primarily maintained by Facebook, and is made available to the wide Computer Science community as an open-source project within the PyTorch repository[5].

In terms of portability, Caffe2 boasts both mobile deployment and first-class support for large scale distribution. It also introduces new Machine Learning techniques, previously unsupported in Caffe, such as Recurrent Neural Networks and Reinforcement Learning [50].

While it still makes use of blobs, Caffe2 no longer employs the Caffe layer abstraction. Rather, a new concept, called *Operator* is brought forward. Operators are highly similar to Caffe layers, and can, to a certain extent, be seen as their more flexible counterparts [50]. Operators aim to be more generic by allowing the programmer to specify a wider range of

[5]PyTorch GitHub repository: `https://github.com/pytorch/pytorch/`

inputs which control the expected behavior of the operation they model. Caffe2 has over 400 implemented operators, and allows the development of new ones [50].

Caffe2 enables multi-node distribution by making use of Facebook's Gloo framework [27] for inter-node communication. Gloo provides collective communication primitives, such as Allreduce, Barrier, or Reduce-scatter, among others. Intra-node communication, for nodes with multiple GPUs, makes use of the NVIDIA Collective Communications Library, and is largely based on Allreduce algorithms [30]. Caffe2 makes use of the Parallel Synchronous Stochastic Gradient Descent optimization approach [19, 30]. Caffe2 has been empirically shown to scale well, and still maintain a high level of performance. An instance of ResNet with 50 layers has been trained on the ImageNet dataset in one hour, using a total of 256 GPUs [19]. In order to overcome the communication overhead, a mini-batch size of 8192 images is chosen. No visible loss in accuracy has been shown to occur for or up to such a large batch size. The study also showed a nearly linear scaling efficiency of 90%, when transitioning from 8 to 256 GPUs.

2.5.4. MXNet

MXNet [10] is a popular open source Machine Learning framework maintained by the Apache Software Foundation. It was initially developed by the authors of *cxxnet*[6], *Minerva*[7] and *purine2*[8]. The framework is designed to be highly scalable and portable, being intended to run on a large range of execution contexts, ranging from something as large as GPU based heterogeneous clusters to something as small as mobile devices. While it can be considered a generic Machine Learning framework, it is mainly focused on Deep Learning. In MXNet, Neural Networks can be trained with standard optimization algorithms, such as Stochastic Gradient Descent. The core runtime of MXNet is written in C++, with clients available in Python, Go, R, Julia, JavaScript, Perl and others.

The frameworks combines both the imperative and declarative paradigms into itself, in order to reap the benefits that each approach brings [10]. The declarative paradigm is employed via so called *declarative symbolic expressions*. Such expressions may represent operations of various complexity, for instance: a simple matrix addition, or a complex neural network layer. MXNet's engine converts symbols to a computation graph, which models both the forward and backward passes. This abstraction, enabled by the declarative paradigm, leaves room for the computation engine to optimize the execution. For instance, the engine can decide to pack all the parameters required for an operation to in a single GPU call whenever possible. Although, generally speaking, more verbose, the imperative paradigm is employed for its greater degree of flexibility, which allows the programmer to specify exactly how certain operations should be executed [10]. This is necessary since the declarative paradigm can hide implementation details which can introduce performance penalties, that might otherwise be fixed using imperative instructions.

MXNet uses a Parameter Server architecture, which is modeled as a KVStore. The interface offers two fundamental operations, *push* and *pull*. It is also possible to specify custom code which describes how the parameters are updated in the Key-Value Store upon a push operation. The parameter server technically works under the synchronous execution model, however, in the current implementation, only the asynchronous model is supported [30]. Regardless, it is structured in two levels. The first handles the synchronization between the devices on the same machine, while the second level is tasked with handling synchronization between the nodes [10].

The computation engine also tries to make efficient use of the available system memory, by deallocating unused variables. The graph model employed by MXNet allows the engine to infer when a variable is no longer used, and as such, its allocated memory can be released.

An experiment on the ILSVRC12 dataset, on 1 and 10 machines respectively reveals a super-linear speedup, although with a slower convergence rate than the single node scenario.

[6] cxxnet GitHub repository: `https://github.com/dmlc/cxxnet`
[7] Minerva GitHub repository: `https://github.com/dmlc/minerva`
[8] purine2 GitHub repository: `https://github.com/purine/purine2`

2.5.5. Petuum

Petuum is a large-scale Machine Learning framework which seeks to provide an approach to iterative convergent Machine Learning challenges which are executed in a distributed environment. Petuum seeks to address and exploit three aspects of distributed Machine Learning problems which are based on iterative convergence: error tolerance to deferred parameter synchronicity, dynamic structural dependency between parameters, and non-uniform parameter convergence [56]. Petuum also tries to overcome some of the limitations present in existing data processing frameworks [56], such as MapReduce [15], Spark [60], or GraphLab and Pregel[9].

Dynamic structural dependency between parameters refers to the continuously changing dependencies and correlations between parameters throughout the training process [56]. While this is not a remarkable problem when the model is monolithic, when employing distribution this becomes an issue which should be taken into account, so as to minimize training errors that may occur as a result of a naïve parallelization strategy [56]. Non-uniform parameter convergence refers to the different convergence rates of a model's parameters during the optimization process [56]. Certain parameters can take longer to train before they converge. As a consequence, they can be prioritized during the training process so as to maximize the number of useful computations and minimize the amount of unnecessary ones.

Petuum employs a lightweight Parameter Server architecture, which exposes a distributed shared memory interface in order to provide the workers in a cluster uniform and global access to the model's parameters [56]. Moreover, due to the straight forward nature of the shared memory interface, the implementation complexity of accessing the parameters is reduced, as it resembles single machine programming [56]. The parameter server uses a bounded asynchronous consistency paradigm by employing the Stale Synchronous Parallel Execution Model. This strategy exploits the proven robustness of gradient based methods to bounded parameter staleness [56]. The parameter server is implemented as a Key-Value store.

Petuum is able to exploit both data and model parallelism. Data parallelism is enabled by partitioning the data up into several disjoint sets which are split among worker processes. The workers are allowed to read the data from whichever source is most suitable, as Petuum does not impose any restrictions on this matter.

Model Parallelism is implemented via a dedicated component called *scheduler*. The scheduler is tasked with assigning each worker a subset of the parameters that then need to be processed and updated. Its behavior can be customized to implement any type of policy for model parallelism. The scheduler, however, serves a more consistent purpose than that: while partitions of data are independent of each other given the model's parameters, the parameters themselves are not independent of each other given the data [56]. In other words, to solve dynamic structural parameter dependencies and non-uniform parameter convergence, a global scheduling mechanism is required [56]. The policies employed by the scheduler dictate if and how the aforementioned issues are tackled. Most policies fall into one or more of these categories [56]:

- **Fixed Schedule**: parameters are allocated in a fixed pattern, such as round-robin, for example.
- **Dependency Aware**: dependency structures and correlations between the model's parameters are analyzed, in order to find the best parameter allocation strategy. It is usually desirable to focus on non-correlated parameters, so as to maximize the information gain obtained via training. Such a strategy might be Coordinate Descent, which optimizes a subset of the parameters, while keeping the others fixed, on each of its iterations.
- **Prioritized Scheduling**: training is focused on those parameters which take more iterations of the optimization algorithm in order to converge. Heuristics which are able

[9] GraphLab and Pregel are systems for large scale graph processing. The graph abstraction employed by these frameworks may not be suitable for certain types of problems in Machine Learning.

to identify such parameters can make use of the magnitude, boundary conditions, or algorithm specific criteria.

Petuum offers a relatively simplistic fault tolerance strategy under the form of checkpoints and restarts. This strategy is generally intended for scales of up to 100+ nodes, however it is not be suitable for thousands of nodes [56]. Although tested on very small scales, Petuum was shown to produce near linear speedup [56]. Petuum's core is written in C++. The software is closed source, and is not made publicly available for free.

2.6. The DAS-5 Cluster

The DAS-5 is a 6 cluster Wide Area Network medium-scale distributed system. The system is designed and implemented by the Advanced School for Computing and Imaging (ASCI). DAS-5 stands for the 5^{th} generation of the Distributed ASCI Supercomputer [5].

DAS systems are built with an aim of providing researchers an infrastructure which can be used towards prototyping and conducting research, and experiments. Each generation of the DAS is build around the present needs of researchers, and the general research trends of the time [5]. A new generation is built whenever need demands it, and the previous system becomes unsuitable for new directions in research. Each new generation is built from scratch, rather than in an iterative fashion, on top of the previous systems.

DAS-5 features 198 nodes, distributed non-uniformly across the 6 available sites. Each site features a head node, while the rest are computation nodes. In total, DAS-5 offers $3,252$ computation cores, spread across the 6 clusters. The machines are generally equipped with Dual Xeon E5-2630 v3 processors. Some of the computation nodes can have special features, such as SDD based storage, GPU accelerators (for instance, the NVIDIA TitanX GPU), or more powerful CPUs. There are two types of interconnections between the nodes within a cluster: 1 Gbps Ethernet, and FDR InfiniBand [3]. The head node has a 10 Gbps Ethernet, instead of 1 Gbps, however, it is not used for computation. The 6 sites are connected via a 100 Gbps optical Wide Area Network paired with Software-Defined Networking[10] [5].

The main features DAS-5 forwards over those pushed by previous DAS iterations are: Heterogeneous Computing, Virtualization, and Big Data processing [5]. Heterogeneous Computing is materialized, for instance, under the form of accelerators, such as GPUs, which may offer better performance for certain types of specialized applications, such as Deep Learning, over traditional hardware. The virtualization of networks, storage, algorithms or libraries is an increasingly prevalent behavior in industry where such features are virtualized and grouped together as services. DAS-5 seeks to strive towards this, to a certain extent, by adding a layer of network virtualization. Big Data processing is generally enabled by technologies which are able to process large volumes of batch or streaming data, and a hardware infrastructure which is able to support such tasks. Through its performant hardware, DAS-5 strives to deliver a platform which enables the aforementioned processing paradigms.

The resources available in the DAS-5 are shared among a set of around 150 researchers, distributed among the available sites. Within a given site, the users are able to reserve any number of idle nodes on a *first come first served* basis, as long as they do not abuse this system by hogging resources for lengthy periods of time, and as a consequence, disallowing others from making use of them. Once reserved, a node belongs fully to a particular user, and is not shared among multiple researchers.

[10] Software-Defined Networking is a technology which allows for a dynamic and efficient network configuration which should lead to improved performance and monitoring.

3

Scalability of Distributed Parameter Update Architectures

This chapter presents a series of large scale evaluations performed in an effort to reveal which parameter update strategies show signs of a scalable behavior. This chapter seeks to forward an answer to **RQ1**: *Which of the distributed parameter update strategies scale well as more nodes are added?*. Section 3.1 presents the chosen frameworks and Neural Network for this experiment, as well as the motivation behind these choices. Section 3.2 presents the methodology behind the evaluation. Section 3.3 provides a concrete description of the experimental setup. Section 3.4 provides an overview of the obtained results. Finally, section 3.5 presents the conclusions of this chapter.

3.1. Selected Neural Network and Frameworks

As the purpose of this part of the evaluation is to identify which parameter update architectures scale well, the chosen frameworks should offer a wide array of implemented architectures, which can be tested. Moreover, the chosen frameworks should be popular within the Computer Science community, so as to maximize the impact of the obtained results. As a consequence of these criteria, both **TensorFlow** and **Caffe2** are chosen. Both frameworks are widely used in industry and academia alike. Moreover, TensorFlow offers a sizable set of available parameter update strategies.

The model chosen for testing is ResNet50, a 50-layer implementation of the generic ResNet architecture [22]. The network has consistently shown that it is able to produce a high degree of performance in terms of classification accuracy, in the context of image recognition tasks.

3.2. Evaluation Methodology

3.2.1. The Workload

There is no lack of publicly available datasets which can be used towards the evaluation of Neural Networks. Popular examples include: the CIFAR datasets, ImageNet [17], MNIST, or MS-COCO. For instance, ImageNet consists of roughly 14.2 million images, with around 20000 classes[1]. Each class contains on average 500 images, with some classes having thousands of examples[2]. Due to its expansive nature, ImageNet is often used to run benchmarks against Neural Networks, for challenges such as object recognition or image captioning.

In spite of these figures, this set of experiments is only conducted against synthetic data. The foremost reason behind this choice refers to the ease of running synthetic data based experiments, as most frameworks have straight-forward means of generating it. Furthermore, the ease of using and generating random data for such experiments implies that these evaluations are easily reproducible. Similarly, new experiments which seek to measure the

[1] These numbers were gathered at the time of writing.
[2] Information on ImageNet: `http://image-net.org/about-overview`

same aspects of different parameter update strategies can easily do so in a fair manner, as the workload is highly similar. In contrast, using real world data would require that the exact same data elements be employed each time. This would make the experiments both more difficult to reproduce, and less transparent.

It is arguable that employing real world data towards these experiments would yield no significant changes to the results obtained on synthetic data. The intuition behind this is that regardless of the nature of the evaluation data, the Neural Network follows the exact same sequence of steps, which translate into the same number of machine code instructions. Hence, the performance should remain roughly the same. Another argument in the same vein refers to the hypothetical case in which the nature of data does indeed affect the results. By using real-world data, there would be bias in the results towards the dataset employed as benchmark. Random data, however, should have no such issues, as long as the distribution from which it is drawn is sufficiently uniform. As a further point, if real world data does bias the results, then the workload would need to be sufficiently large so as to increase its generalization power. Since these experiments take a significant amount of time, this is unlikely to be a viable solution. For instance, given an 8 node cluster with InfiniBand interconnects, the time required to process a 128 image batch with the Parameter Server architecture, using both forward and backward passes would take around 40 second to fully complete in evaluation mode. Using 15 batches during an evaluation run would imply the experiment finishes in around 10 minutes wall-clock time. This is generally the case for synthetic data. Assuming that a hundredth of the ImageNet is able to yield an unbiased view of the architecture performance, that is, 142,000 images, which translate into 139 batches (since the global batch size is $1024 = 128 \cdot 8$). Consequently, this would take roughly 1.5 hours.

It would indeed be highly interesting to run the same experiments on a set of real world data, perhaps originating from multiple canonical datasets, in order to see how they compare among themselves and with their synthetic data counterparts. However, as explained, the time consuming nature of running experiments, even those relying on a small number of batches, means that this is not a viable goal for this study.

3.2.2. The Performance Metrics

In order to be able to compare and asses the true performance of a framework, relevant metrics need to be chosen. The metrics should show how well the framework fares in real-life scenarios. Since the most prevalent field in Machine Learning is Deep Learning, and as a consequence most frameworks generally rely on running Mini-batch Stochastic Gradient Descent during training, the time it takes to process a batch is a good indication of a framework's performance. As a consequence, the following metrics are employed:

- **The forward propagation time**: refers to measuring the time it takes to pass an entire batch of images forward through the network. More specifically, this evaluation implies no backward propagation of the gradients through the network in order to adjust the model's weights. This should mostly be an indication of the time it takes for the framework and the network to produce a classification for a batch of images. This metric is measured in *seconds / batch*.

- **The number of processed images per second in a forward pass**: indicates the number of images which can be processed within the span of a second when only a forward pass is employed. No backward pass is made use of. This metric is similar to *the forward propagation time*, however, unlike the aforementioned metric, it indicates the performance at the granularity of individual images rather than batches. This metric is measured in *images / second*.

- **The forward and backward propagation time**: refers to the time it takes to process an entire batch of images both forward and backward through the network. This evaluation implies that Backpropagation is applied, hence, gradients are propagated backwards throughout the network in order to adjust the model's parameters. This should be a

highly relevant indication of the framework's performance during the training phase. This metric is measured in *seconds / batch*.

- **The number of processed images per second in a forward and backward pass**: indicates the number of images which can be processed within the span of a second when both forward and backward passes are employed. As with *the forward and backward propagation time*, this metric essentially indicates the performance of the framework during a forward and backward pass, but at the granularity of individual images. It is measured in *images / second*.

The set of aforementioned metrics should yield a comprehensive view of a framework's behavior in the context of raw performance. Moreover, these metrics are oftentimes employed in literature towards exemplifying the performance of such systems, and parameter update architectures [2, 4, 10, 19, 48, 61], which is an indication that they are relevant, and may lead to interesting comparisons in the future between different frameworks and parameter update strategies.

3.2.3. The Evaluation Procedure

In order to evaluate the frameworks, the following procedure is employed: each system is evaluated using a total of 15 batches, out of which 5 are used during the warm-up stage, and the other 10 are used for running the true evaluation per se, where metrics are gathered. These 15 batches constitute an evaluation run. One evaluation run needs to be executed for either measuring the forward pass metrics or the forward and backward pass metrics, but not both. Moreover, since the purpose of the evaluation is to better understand the scalability of the distributed parameter update strategies and the frameworks, the underlying hardware resources, on top of which the frameworks run, will be varied. More specifically, the number of underlying nodes will be progressively increased using the following sequence: 1, 2, 4, 8, 16, 24, 32, 40, 48, and 56. Consequently, for each node configuration, two evaluation runs need to be executed: one for the forward pass metrics, and the other for the forward and backward pass metrics. Additionally, since performance under both the Ethernet and InfiniBand connections is being assessed, all the aforementioned experiments need to be executed once on InfiniBand and once on Ethernet. This entire process needs to be repeated for each of the parameter update architectures which are being evaluated. After each evaluation run, the aforementioned set of metrics are collected and processed, in order to produce a final result. This generally refers to averaging the individual results obtained in each node.

It should be mentioned, however, that as explained in section 2.6, the DAS-5 cluster is shared among multiple researchers. Due to the first come first served and fair share nature of the node reservation system, it is generally hard to run tests on a very large number of nodes, such as 48 or 56, since it requires very favorable conditions. As a consequence, for certain architectures such large scale evaluations are not available, since the opportunity did not arise.

3.3. Experimental Setup

This section defines the experimental setup used for running this set of evaluations. The description refers to the hardware, software, parameter update architectures, and concrete workload and framework configurations used.

3.3.1. Hardware

The evaluation runs on top of the VU DAS-5 site, which features a total of 68 nodes. All nodes employed for this experiment have Dual 8-core Intel Xeon E5-2630 v3 Processors at a frequency of 2.4 GHz, with a total of 32 threads. Each machine has 64 GB in RAM memory, and two 4 TB HDD storage devices. Nodes are interconnected both via 1 Gbps Ethernet and FDR InfiniBand which provides 56 Gbps interconnects.

Experiments are run on a variable number of nodes, ranging from 1 to 56, in the sequence: 1, 2, 4, 8, 16, 24, 32, 40, 48 and 56. As explained in section 2.6, the policy via which

resources are shared among the users of the cluster sometimes makes it impossible to run experiments at very large scales, such as 48 or 56 nodes.

Experiments are also run using both Ethernet and InfiniBand connections. More specifically, each experiment executed on Ethernet interconnects is also executed using the exact same parameters over FDR InfiniBand.

3.3.2. Software

The Operating Systems installed on the DAS-5 machines is the CentOS 7. In order to avoid adding any overhead to the evaluation process, the metric gathering task is left to the benchmarking modules available in the selected frameworks. For Caffe2, no additional sources need to be added in order to make this possible, other than the standard build of the framework. For TensorFlow on the other hand, it is necessary to make use of the sources available in the Convolutional Neural Network benchmarking suite[3], more specifically, the `benchmark_cnn.py` source. Intuitively, this is required in addition to the standard Nightly TensorFlow build.

For TensorFlow, a high-performance version of the ResNet50 network is used. This is pre-implemented, and made readily available to the public via the TensorFlow benchmarking suite. A similar approach is taken with Caffe2 as well. A developer implemented version of the ResNet50 Neural Network is used as a base for running the evaluations. This version is also publicly available[4]. Unlike TensorFlow, however, this source is modified in order to more easily allow distributed training to take place. The modified code for this benchmark is available on the thesis' GitHub repository.

Methodology for Spawning Distributed Machine Learning Clusters

In order to truly facilitate the distribution of the training and evaluation phases over a cluster of nodes, scripts have been written for both Caffe2 and TensorFlow. In both cases, the processes involved in the evaluation task need to be spawned across the underlying nodes. The scripts automate this, and handle any other relevant tasks, such as that of monitoring the spawned processes, and terminating any hanging instances left after the evaluation has been completed, such that the nodes can quickly be reused. Furthermore, the scripts take care of framework specific distribution requirements, such as that of specifying the exact topology of the processes and nodes involved in the training phase, as is the case in TensorFlow. Another example, in the context of Caffe2 evaluations, refers specifying a unique *run id* for each execution. The scripts are both called `cluster_spawner_benchmark.py`, one version being used for Caffe2, while the other for TensorFlow. They are available on the thesis repository.

3.3.3. Parameter Update Architectures

The parameter update architecture behind Caffe2 uses a version of Ring Allreduce, with recursive halving and doubling, for better bandwidth and latency guarantees [30]. At the time of writing, Caffe2 offered no additional architectures.

TensorFlow features a large number of possible parameter update schemes. More specifically, it offers the following strategies: Parameter Server, Replicated, Distributed Replicated, Independent, Distributed Allreduce, Collective Allreduce, and Horovod. All of these strategies have been detailed in section 2.5.2. For this evaluation, all architectures, which allow distribution, are chosen, with the exception of Horovod, which is mainly targeted towards GPU training. More specifically: Parameter Server, Distributed Replicated, Distributed Allreduce, and Collective Allreduce. It should also be mentioned that the Parameter Server architecture is evaluated in two flavors: Collocated and Non-collocated. In the Non-collocated version, the node which hosts the parameter server processes does not host any worker processes. Thus, these two different types of processes never contend for resources. In the Collocated version, the machine hosting and running the parameter server processes also hosts a worker process. In such a context, the processes can contend for the machine's resources. Distributed Allreduce is also evaluated in two flavors, based on the underlying reduction strategy: PSCPU and Ringx1. The semantics of these strategies are also presented in 2.5.2.

[3] TensorFlow Benchmarking Suite: `https://github.com/tensorflow/benchmarks`
[4] Caffe2 ResNet50 code: `https://tinyurl.com/y5s7pngy`

TensorFlow also features an option for controlling the operator to device bounding phase. The available options for this phase are: *Soft Placement* and *Hard Placement*. In Hard Placement, the user specifies which types of devices the operators should be placed on. In Soft Placement, on the other hand, the user can only provide suggestions on this matter, however, the final decision belongs to TensorFlow. In some cases, when a GPU card is readily available on a machine, TensorFlow will choose to run operations on it, even if Hard Placement is enabled, as long as TensorFlow's build has GPU support. Consequently, true CPU based Hard Placement generally requires a non-GPU compatible build, if some of the underlying machines are equipped with GPU cards. The aforementioned approach, i.e. that of using a non-GPU compatible build with Hard Placement is employed for all of the experiments, with few exceptions. The Collocated Parameter Server architecture is tested against both the Hard and Soft Placement heuristics, while the Non-collocated Parameter Server strategy is only tested against a Soft Placement heuristic. All the other strategies are exclusively tested using the Hard Placement heuristic.

Caffe2 could not be evaluated for cluster sizes greater than 24 nodes. This is due to consistent timeouts which occur during evaluations that are executed on larger clusters. The timeouts are enforced by the Gloo library, which facilitates inter-node communication. By default these timeouts are very low. Caffe2, however, does not offer a straight-forward way to remove them, or make them more lenient, as access to the Gloo library via Caffe2 is limited.

TensorFlow's Distributed Allreduce also has similar issues regarding more sizable clusters. Both the *Ringx1* and *PSCPU* strategies suffer from the similar issues when it comes to testing for clusters with more than 16 nodes. More specifically, the issues either refer to timeouts, or to evaluations running for an impractical amount of time.

3.3.4. Framework Configurations and Concrete Workload

Mini-batch Stochastic Gradient Descent is used as the optimization algorithm of choice. A base learning rate of 0.1 is selected, although this value should have little, if any impact on the results of the evaluation[5]. The data representation of choice is 32 bit floating point.

As specified in section 3.2.1, both Caffe2 and TensorFlow are evaluated against random data. TensorFlow is evaluated using random images generated based on an ImageNet distribution. This approach is chosen since generating images from a random uniform distribution is not allowed in the TensorFlow benchmarking suite for ResNet50. For Caffe2, purely random images are generated, and used for the evaluation. The fact that the workloads originate from different distributions should not have a noticeable impact on the results. More specifically, the amount of machine operations executed during evaluation should remain constant, since the forward propagation and backpropagation phases execute the same set of operations regardless of the distribution of the underlying data.

The batch size in images is 32 for both Caffe2 and TensorFlow. This batch size is chosen since it has been shown to be the largest value that still produces good generalization and test performance, unlike progressively greater batches, which negatively impact the aforementioned training quality metrics [9]. An evaluation run consists of 15 batches, 5 of which are used for warming up the system, and the other 10 are used for gathering measurements.

For TensorFlow, the dimensions of a generated image are $300 \times 300 \times 3$ (Height $\times$ Width $\times$ Channels). Images are then cropped down to $224 \times 224 \times 3$. The preprocessing operation is not taken into account for the metrics. The *NHWC* data format is chosen for the image inputs.

The *NCHW* packing format was used for the input data in Caffe2. The dimensions of an image are $32 \times 32 \times 3$. No further preprocessing is performed on the images. Even after numerous tries, a greater image size, such as $224 \times 224 \times 3$ was not possible, due to the repeated timeouts encountered during evaluation. This made such a configuration impractical. Caffe2's smaller workload will inevitably have a positive impact on its results. However, due to the layered structure of Neural Networks, the effect of the smaller workload should only be felt in the first hidden layer, since this layer is the only one which depends completely on the

[5] In real training scenarios, the learning rate affects the number of gradient updates required until convergence.

Strategy Name	Parameter Server	Controller	Max. Workers per Node	Total Nr. of Workers	Description
Collocated Parameter Server	×		1	n	Each node has a worker. One node as an additional parameter server process.
Non-collocated Parameter Server	×		1	$n-1$	One node has a parameter server process. The others each has one worker.
Distributed Replicated	×		1	n	Each node has a worker. One node as an additional parameter server process.
Distributed Allreduce		×	1	n	Each node has a worker. One node as an additional controller process.
Collective Allreduce			1	n	Each node has a worker.
(Caffe2) Ring Allreduce			1	n	Each node has a worker.

Table 3.1: The table describes the different configurations of evaluated parameter update architectures. Certain strategies may not feature any Parameter Server or a Controller.

input layer. The other layers remain the same, and execute the same operations, regardless of the size of the input images.

The general rule is that each node hosts one and only one worker process. This approach is preferred to spawning multiple worker processes on the same machine since both frameworks employ multi-threading which completely exploits the available resources. Multiple processes in a single host may imply competition for the shared resources, and as a consequence could cost more to execute due to the context switch overheads. In TensorFlow, with the exception of the Non-collocated Parameter Server and Collective Allreduce architectures, there will always exist one node which has two processes: a worker process and a parameter server or controller process. In the case of the Non-collocated Parameter Server architecture, one node will not host a worker process, but rather a parameter server in its place, while in the case of the Collective Allreduce, all nodes will host a worker process. For Caffe2 each node will host one and only one worker process. Table 3.1 summarizes the facts described above.

3.4. Experimental Results

Section 3.4.1 presents a collection of key findings based on the results obtained by executing the scalability experiments discussed so far. The following sections present an in depth discussion of the aforementioned results. More specifically, section 3.4.2 presents the results obtained over Ethernet interconnects, and section 3.4.3 focuses in on the results obtained for InfiniBand interconnects.

3.4.1. Key Findings

The set of experiments executed over both Ethernet and InfiniBand, in the interest of finding how well distributed parameter update architectures scale over different cluster sizes, reveals some interesting key points:

- *Collective Allreduce* generally yields the best performance out of all the tested strategies. It performs very well for both the forward pass (see sections 3.4.2 and 3.4.3) and the forward and backward passes (see sections 3.4.2 and 3.4.3). It is especially suitable for Ethernet interconnects, where it significantly outperforms any other strategy due to its efficient use of the available bandwidth. This latter claim is reinforced in chapter 5. The strategy continues to show top performance when upgrading to InfiniBand interconnects. The parameter update strategy shows linear scalability, in terms of processed images per second, relative to the number of nodes in all tests, with no signs of performance saturation.

- Without InfiniBand connections, all versions of the TensorFlow's *Parameter Server* strategy show logarithmic growth in terms of the processed images per second. In the Ethernet experiments, they are unable to derive a considerable amount of benefits from the addition of more resources past the 8 node mark (see figures 3.2 and 3.4). Consequently, in such a context, these strategies cannot scale well, and require larger workloads in order to minimize the frequency of communication. This latter claim is reinforced in chapter 4 figure 4.4. However, as seen in [37], increasing the batch size has negative impacts on the quality of the trained model. Moreover, as argued in a future section 4.4.3, increasing the batch size has its own limitations in terms of benefits to the training speed.

- When the *Parameter Server* strategies are employed over InfiniBand, scalability appears

to no longer be an issue. The architecture shows linear scalability in the number of processed images per second, relative to the number of nodes, with no significant sign of performance saturation (see figures 3.6 and 3.8 in section 3.4.3). For these experiments, Parameter Server performs similarly to *Collective Allreduce*. Consequently, it should generally only be employed if high bandwidth connections are available. Otherwise, it does not scale well. This conclusion is also supported by further experiments detailed in section 4.4.3, and in chapter 5.

- Although not concrete, certain experiments, such as the forward pass runs on either InfiniBand or Ethernet (see sections 3.4.3 and 3.4.2 respectively) reveal that, at times, the Soft Placement version of Collocated Parameter Server performs worse than its Hard Placement counterpart. This is visible in figures 3.1 and 3.5, in sections 3.4.2 and 3.4.3. Such behavior may suggest that the Soft Placement heuristics can sometimes incur performance penalties. This is however inconclusive, and further work is needed to better understand this phenomenon.

- Based on the metrics evaluated in this chapter, there is evidence which supports the hypothesis that Collocated Parameter Server performs better than its Non-collocated counterpart for forward passes. This can clearly be seen in figures 3.1 and 3.5 in sections 3.4.2 and 3.4.3. However, the competition between the two strategies for forward and backward passes is tighter (see figures 3.3 and 3.7), with no clear winner.

- While proving to have a fast and scalable forward pass, comparable to that of Collective Allreduce, *Distributed Replicated* becomes inefficient when the backward pass is added. Figures 3.6 and 3.8 can be compared in order to observe this behavior. In the forward and backward evaluations, the strategy behaves very similarly to the Parameter Server architectures, by quickly saturating in performance for Ethernet connections (see figure 3.4), and showing decent scalability for InfiniBand (see figure 3.8). As argued in chapter 5, this is due to the bandwidth limitations imposed by the Ethernet interconnects on the parameter server process.

- *Distributed Allreduce* underperforms for both Ethernet and InfiniBand. This strategy does not show signs of scaling well. This point is discussed in both sections 3.4.2 and 3.4.3. Moreover, as seen in the case of the Ethernet forward and backward evaluation (see figure 3.4), this update architecture shows signs of worsened performance as more nodes are added (see figure 3.4).

- *Caffe2* performs best among all strategies in the single node contexts, although its workload is smaller than that of the TensorFlow strategies. This is consistent across all the presented figures in this chapter. As distribution is employed, Caffe2 becomes less efficient than its single node case. After the addition of a second node, Caffe2's per batch time increases roughly 19 times (see either figure 3.3 or 3.7). The addition of more nodes produces progressively better performance, however, it is still less efficient than the single node case.

- *Caffe2* does not benefit in any form from InfiniBand, as its performance remains constant regardless of the underlying interconnects. For reference, one can compare figures 3.3 and 3.7 in order to see that the batch time remains roughly the same regardless of the interconnections. This can indicate that the Ring Allreduce strategy employed by Caffe2 makes efficient use of the available bandwidth, even when only Ethernet is available. This latter point is argued and reinforced in chapter 5, by inspecting Caffe2's communication patterns and Network Traffic. The efficient use of the available bandwidth would suggest that Caffe2's lack of performance in the context of multi-node distribution is not due to bandwidth limitations, but rather due to some other aspect of distribution. A likely candidate for this is the high consumption of CPU resources, a point which is further argued in chapter 5.

3.4.2. Ethernet Results

This section presents the results of the evaluations executed over an Ethernet connection, for both the forward pass and the backward pass experiments.

Forward Pass

Figures 3.1 and 3.2 show a graphical representation of the forward pass metrics, given Ethernet interconnections. These plots show the results obtained for Caffe2 and all of TensorFlow's architectures, with the exception of Distributed Allreduce, which cannot be tested exclusively for its forward pass.

Figure 3.1 describes the time required to process a batch of 32 images, given an increasing number of nodes. For a singular node, Caffe2 outperforms any of the other TensorFlow architectures. TensorFlow finds itself quite far behind, with the Hard Placement Collocated Parameter Server and Distributed Replicated strategies, proving to be the next best options. A surprising result is that the Soft Placement Collocated Parameter Server yields a considerably poorer performance than its Hard Placement counterpart. This might suggest that at certain times, the heuristics employed by TensorFlow towards automatically binding operators to devices might incur performance penalties. No values are present for the single node case of Collective Allreduce and Non-collocated Parameter Server, due to the fact that these strategies require at least two running processes in the session.

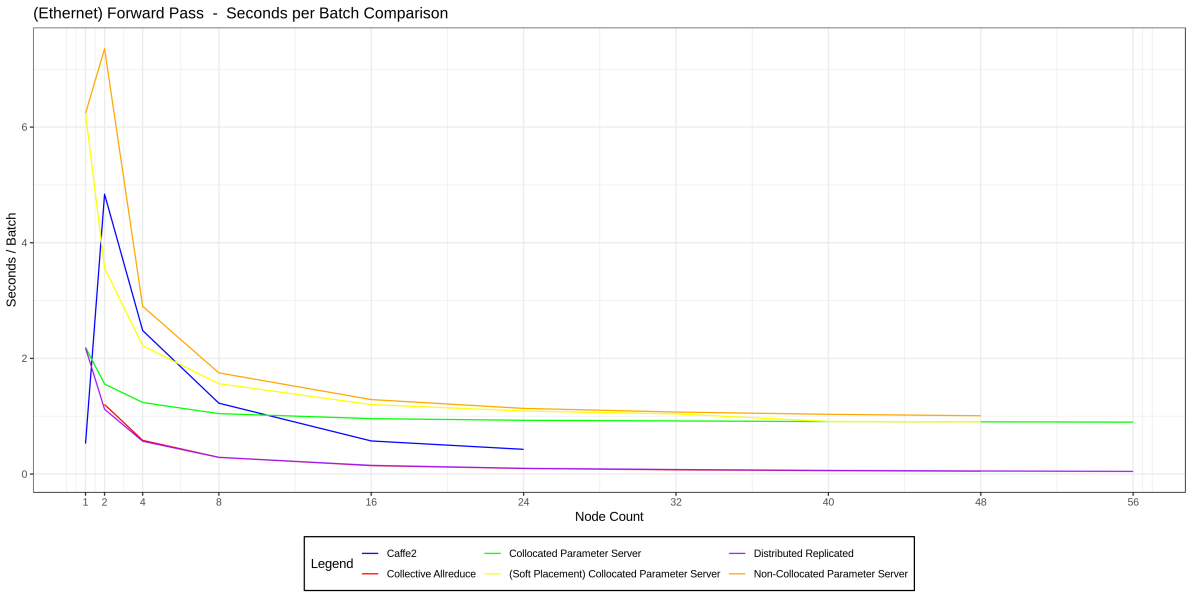


Figure 3.1: Number of seconds required in order to process a batch of 32 images given an increasing number of nodes. Lower values are better. Forward pass with Ethernet interconnects scenario.

For the distributed case, TensorFlow produces the best results with the Collective Allreduce and Distributed Replicated strategies. Both these architectures show similar results, as their performance in terms of time per batch sees only small improvements past 0.1 seconds. This occurs at the 16 nodes mark. The Hard Placement Collocated Parameter Server converges relatively quickly to a time of 1 second per batch, at 8 nodes. Both the Soft Placement strategies converge more slowly to the same time, at around the 24 nodes mark. This continues to indicate that TensorFlow's Soft Placement heuristics may sometimes have a negative impact on performance. Upon distribution, Caffe2 produces a spike, which, however, stabilizes as a greater number of nodes is added. More precisely, the time per batch increases from 0.5 seconds to 10.2 seconds. Caffe2 manages to yield a time of around 0.09 seconds for 24 nodes, however, it still lags behind the aforementioned TensorFlow architectures. The root cause for Caffe2's lack of performance is not due to bandwidth limitations. This point is reinforced in section 3.4.3, where InfiniBand connections yield no obvious benefit. Moreover, chapter 5 argues the same point after inspecting Caffe2's network traffic patterns. This loss

in performance is likely created by another intrinsic aspect of multi-node distribution, whose negative impact gets ameliorated as more nodes are added.

Figure 3.2 describes the number of processed images per second in the context of a forward pass. These results reinforce what was seen in figure 3.1, and more clearly show the benefits of multi-node distribution. Both the Distributed Replicated and the Collective Allreduce strategies prove to scale linearly, with Collective Allreduce having the slight edge. Caffe2 also benefits from linear scalability, however its growth constant is lower than the other two strategies. A likely cause behind this limited scalability is presented in chapter 5, which takes a deeper look at the CPU consumption pattern behind Caffe2's Ring Allreduce strategy. The pattern reveals that Caffe2 is highly CPU intensive, much more so than other architectures. The rest of the tested strategies show logarithmic growth in the case of the forward pass, as their performance quickly saturates.

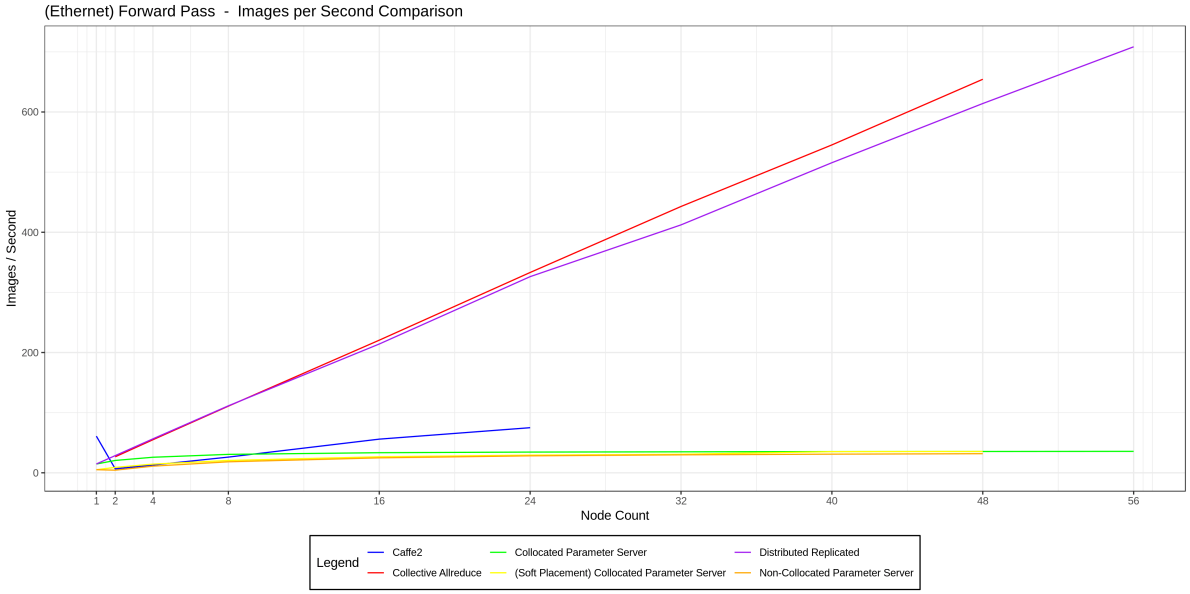


Figure 3.2: Number of processed images per second given a progressively greater number of nodes. Greater values are better. Forward pass with Ethernet interconnects scenario.

Forward and Backward Pass

Figure 3.3 reveals that, once more, Caffe2 is able to outperform all of the other architectures by quite a considerable margin when only one node is used. Its batch time is around 2 seconds. In contrast, TensorFlow's architectures lag behind. The Soft Placement Collocated Parameter Server yield the best performance relative to the other TensorFlow architectures, with a time of around 7.8 seconds. This contrasts to the forward pass evaluations, where the Soft Placement version performed worse than its Hard Placement counterpart. The Distributed Replicated and Hard Placement Collocated Parameter Server yield the same batch time of around 8.8 seconds. As argued in chapter 5, Distributed Replicated is unable to scale in the forward and backward pass using Ethernet interconnects due to the bandwidth limitations imposed on its parameter server process during the parameter update dissemination phase of the global gradient updates. This phenomenon is also visible in the Parameter Server architecture. The Collective Allreduce, Distributed Allreduce and Non-collocated Parameter server strategies cannot be tested for single node performance.

For the multi-node case, Collective Allreduce once more yields the best result out of the available architectures. Its time per batch sees little improvement past the 16 node mark, at 0.55 seconds. All the other TensorFlow architectures, bar Distributed Allreduce, produce similar results to each other. Their batch time improves marginally past 1.5 seconds per batch, at the 16 node mark. The Distributed Allreduce strategies yield poor performance, with the *Ringx1* version showing a slight increase in the batch time as more nodes are added. Similar

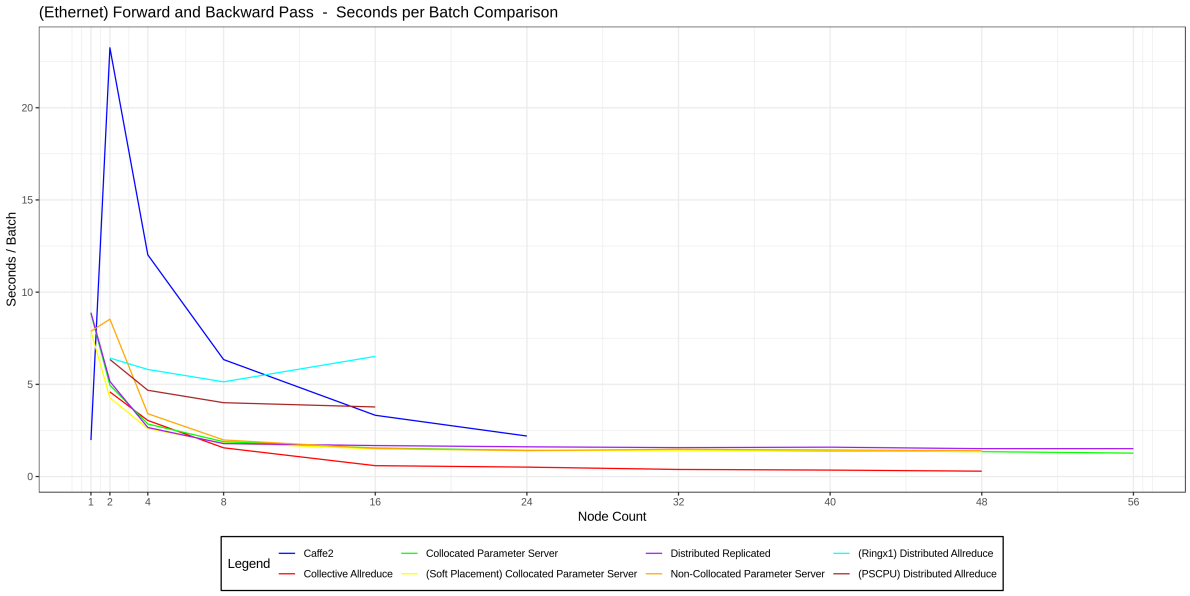


Figure 3.3: Number of seconds required in order to process a batch of 32 images given an increasing number of nodes. Lower values are better. Forward and backward pass with Ethernet interconnects scenario.

to the forward evaluations, Caffe2 shows an initial loss in performance after the addition of a second node. As the number of nodes is increased, Caffe2's performance improves, producing very similar results to the TensorFlow strategies. In spite of this, it still remains less efficient than for the single node case. This reinforces the notion that Caffe2's distribution incurs performance penalties, and that distributed Caffe2 requires a sizable number of machines before a high degree of performance can be gained.

Figure 3.4 describes the considerable advantage that Collective Allreduce has over the other architectures, exhibiting linear scalability relative to the number of underlying nodes available for computation. Caffe2's performance here is also linear, however, it lags far behind Collective Allreduce. For instance Caffe2 processes 15 images at the 24 node mark, while Collective Allreduce processes around 61. As already mentioned this is likely due to the CPU intensive nature of the parameter update scheme. This point is argued in chapter 5. All other architectures show logarithmic growth, and are unable to improve considerably past 20 images per second. The Distributed Allreduce strategies exhibit very poor performance, constantly under 7 images per second, which decreases as more nodes are added.

3.4.3. InfiniBand Results

This section presents the results of the evaluations executed over an FDR InfiniBand connection, for both the forward pass and the forward and backward pass experiments.

Forward Pass

Figures 3.5 and 3.6 show a graphical representation of the forward pass metrics given InfiniBand interconnects. These plots describe the results obtained for Caffe2 and all of TensorFlow's architectures, with the exception of Distributed Allreduce, which cannot be tested exclusively for its forward pass.

Figure 3.5 shows the seconds required in order to process a batch using only a forward pass. Caffe2 behaves no different for InfiniBand than it does for Ethernet, following the same pattern as before (see section 3.4.2): it performs well in the single node context, but becomes less efficient as distribution is used. Moreover, it produces highly similar values across the two connection types: a forward batch time at 24 nodes is 0.13 for both Ethernet and InfiniBand. This suggests that Caffe2's lack in performance in the context of distribution does not stem from bandwidth limitations, since the addition of InfiniBand would have had a positive impact on performance.

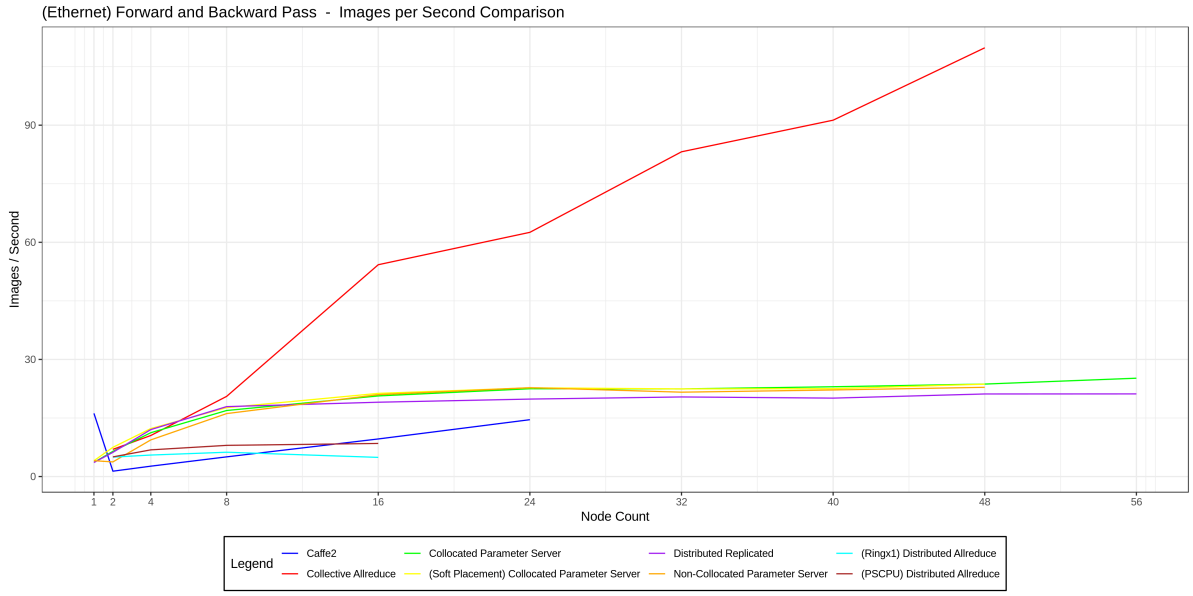


Figure 3.4: Number of processed images per second given a progressively greater number of nodes. Greater values are better. Forward and backward pass with Ethernet interconnects scenario.

In contrast, TensorFlow's behavior for InfiniBand connections is vastly different than that of Caffe2. Bar the Non-collocated Parameter Server architecture, all of the framework's strategies converge to a very similar time per batch of around 0.1 seconds. Distributed Replicated, Collective Allreduce, and Collocated Parameter Server manage to do this quickly, at around the 16 node mark. The Soft Placement version of the Parameter Server manages to do this as well, however, it requires more resources: around 40 nodes.The Non-collocated Parameter Server falls short of converging to the same time as the framework's other parameter update architectures, likely due to the fact that it underuses the resources in one of its nodes. More specifically, as will be seen in chapter 5, the node hosting the parameter server makes little use of the computational resources when the other worker nodes are processing batches.

Even in the case of the forward pass, when communication is less significant, the benefits of a faster interconnection can be seen quite clearly for TensorFlow. All of its architectures behave relatively similar in terms of the seconds per batch metric. Figure 3.6, however, shows the difference in performance between these strategies more clearly. Consistent with the Ethernet case, Collective Allreduce performs best, having a slight edge over Distributed Replicated. While still lagging in performance, the Collocated Parameter Server architectures produce much better results for InfiniBand than for Ethernet, exhibiting linear scalability relative to the number of nodes. Both the Non-collocated Parameter Server and Caffe2 strategies show similar performance, with the TensorFlow architecture proving to have a slight edge. As Caffe2 does not find any performance improvement by making use of InfiniBand, TensorFlow scales better for all its available architectures. In contrast, in the forward pass Ethernet case, only Distributed Replicated and Collective Allreduce proved to be better.

Forward and Backward Pass

Figures 3.7 and 3.8 show a graphical representation of the forward and backward pass metrics using InfiniBand interconnections. These plots show the results obtained for Caffe2 and all of TensorFlow's architectures. This time around, Distributed Allreduce is evaluated using two of its implementations: *PSCPU* and *Ringx1*.

Figure 3.7 describes the batch time, across different sized clusters, when both a forward and backward pass are employed. The results of this metric show that the available TensorFlow strategies seem to perform very similarly to each other. The average value to which the strategies converge is 0.5 seconds. There is one exception to this rule: Distributed Allreduce. It performs significantly worse than any of the other strategies. This applies to both

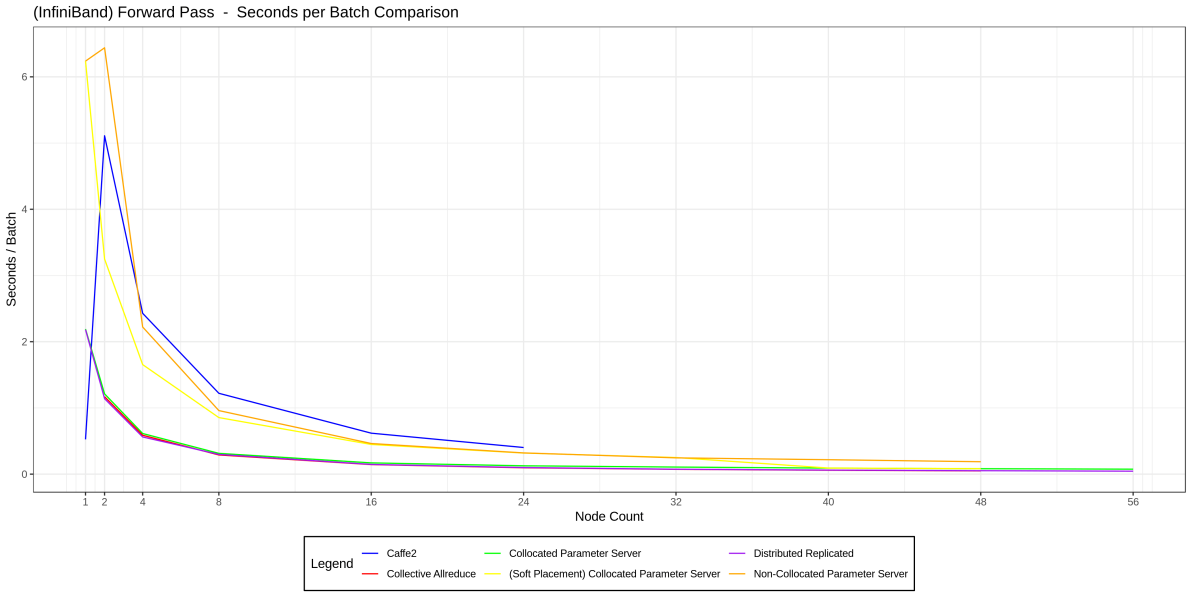


Figure 3.5: Number of seconds required in order to process a batch of 32 images given an increasing number of nodes. Lower values are better. Forward pass with InfiniBand interconnects scenario.

the Ringx1 and the PSCPU reduction approaches. This observation is consistent to what was seen in section 3.4.2 for Ethernet connections.

As in the forward pass case, InfiniBand does not produce any improvement for Caffe2 in the backward pass scenario. Caffe2's behavior is identical to what was seen for the same experiment in an Ethernet medium (see section 3.4.2). As Caffe2 does not take advantage of the underlying interconnects, unlike TensorFlow's architectures, it severely falls behind in terms of performance. For example, at the 24 node mark, Caffe2 completes a forward and backward pass on a batch in 2.15 seconds, whereas Collocated Parameter Server yields 0.52 seconds.

Figure 3.8 shows the number of processed images in the span of a second using both a forward and a backward pass. This plot reveals the performance of each of the strategies more clearly. Similarly to the Ethernet based forward and backward pass evaluations, Collective Allreduce exhibits a high degree of performance. Most of TensorFlow's strategies greatly benefit from the InfiniBand interconnections. All versions of the Parameter Server strategy roughly perform as well as Collective Allreduce. This is vastly different from the Ethernet case, where Collective Allreduce was performing much better than any of the competing strategies. This suggests the the Parameter Server strategy is highly dependent on the available bandwidth. This point is further reinforced and argued in chapter 5. With InfiniBand, there is no clear winner among these four strategies, however Collective Allreduce seems to be the most consistent, given all experiments, while the Hard Placement Collocated Parameter Server seems to have the most predictable growth for InfiniBand. Distributed Replicated also benefits from the greater bandwidth, however to a lesser degree, as it lags behind some of its counterparts. The improvement in performance supports the argument that Distributed Replicated is bandwidth limited by Ethernet, a point which is further argued in chapter 5. Caffe2 and the Distributed Allreduce strategies exhibit similar performance, however, Caffe2 still shows a linear increase in performance as more nodes are added, unlike Distributed Allreduce whose performance dips.

3.5. Experiment Conclusions

RQ1: *which of the distributed parameter update strategies scale well as more nodes are added?* For brevity purposes, it should be mentioned that in what follows, scalability refers to the number of processed images per second relative to the size of the cluster:

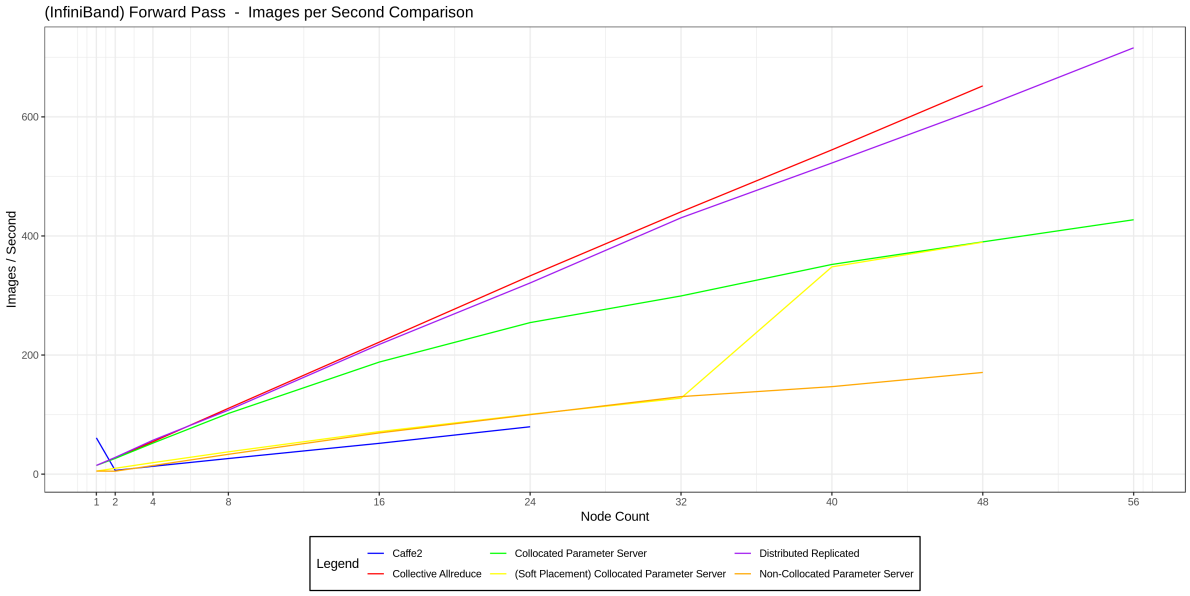


Figure 3.6: Number of processed images per second given a progressively greater number of nodes. Greater values are better. Forward pass with InfiniBand interconnects scenario.

- Collective Allreduce scales very well for both the individual forward pass and the forward and backward passes together, across all cluster sizes and underlying interconnects, showing linear scalability. It consistently yields the best performance among the tested strategies, and is especially recommended for Ethernet based mediums, as it outperforms all other strategies by a large margin.
- The Parameter Server architecture does not scale well for Ethernet interconnects, showing logarithmic growth for both types of pass combinations. However, when paired with InfiniBand connections, the architecture scales linearly, and proves to be a high performance architecture for InfiniBand based clusters. In these contexts, it is comparable to Collective Allreduce.
- Caffe2 outperforms every competitor in terms of single node evaluations. In the distributed context, however, the underlying Ring Allreduce strategy significantly lacks in performance. While it indeed defines linear scalability, its growth factor is significantly lower than the other parameter update strategies. Moreover, Caffe2 does not derive any benefits from InfiniBand interconnects, as performance is equal to Ethernet based mediums.
- The Distributed Replicated strategy offers excellent linear scalability for its forward passes, regardless of communication medium, which is comparable to Collective Allreduce. When both the forward and backward passes are used, the strategy fails to scale well for Ethernet interconnects, showing logarithmic growth. For InfiniBand connections, the growth is linear, however not as good as Parameter Server or Collective Allreduce.
- Distributed Replicated, using Ringx1 and PSCPU, is unable to scale at all in any of the tested mediums. This strategy scales the worst, in certain cases showing decreased performance as more nodes are added.

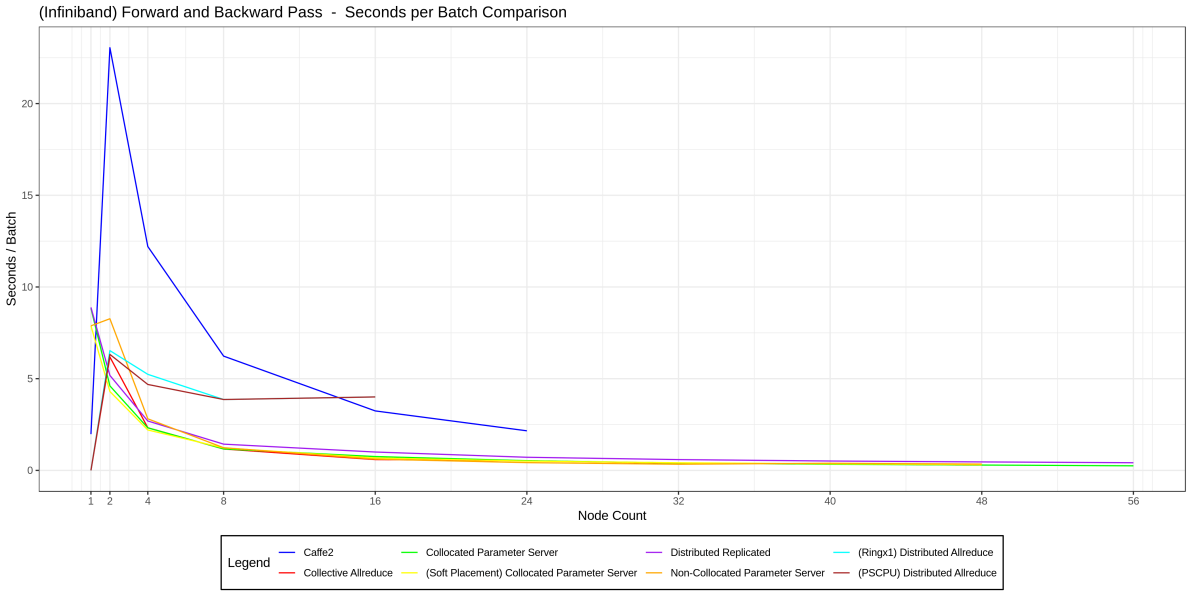


Figure 3.7: Number of seconds required in order to process a batch of 32 images given an increasing number of nodes. Lower values are better. Forward and backward pass with InfiniBand interconnects scenario.

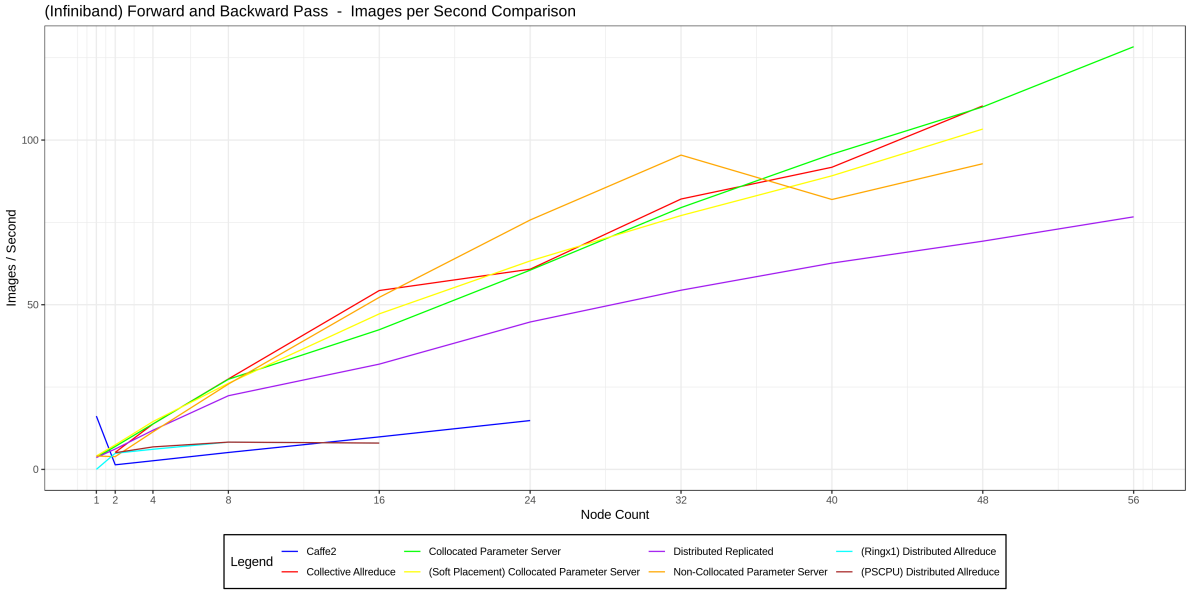


Figure 3.8: Number of processed images per second given a progressively greater number of nodes. Greater values are better. Forward and backward pass with InfiniBand interconnects scenario.

4

Batch Size Impact on the Parameter Server Architecture

This chapter presents the large scale experiments executed in an effort to answer **RQ2**: *How well does the Parameter Server strategy scale relative to the batch size and the number of nodes?*. Section 4.1 presents the chosen framework and Neural Network for this experiment. Section 4.2 presents the methodology behind the evaluation. Section 4.3 provides a concrete description of the experimental setup. Section 4.4 provides an overview of the obtained results. Finally, section 4.5 presents the conclusions of this chapter.

4.1. Selected Neural Network and Framework

The framework selected to run this set of experiments is TensorFlow. Besides its popularity and wide support from the Computer Science community, TensorFlow heavily relies on the Parameter Server update architecture for its large scale distributed Machine Learning training processes. It is, in fact, the framework's default means of enabling distributed Machine Learning. As such, TensorFlow's implementation of the Parameter Server architecture should provide a good indication of the strategy's true behavior in regards to variable node counts and changing batch sizes. Moreover, continuing to use TensorFlow as a platform for evaluation is beneficial towards building on top of the framework's results obtained in chapter 3.

The Neural Network of choice for this set of experiments is the ResNet50. This is the same network that was used in chapter 3. The reasons for choosing it remain the same, and are not reproduced here, for the sake of brevity.

4.2. Evaluation Methodology

4.2.1. The Workload

As was the case in chapter 3, the data chosen for this set of experiments is randomly generated synthetic data. The reasons behind this decision are presented in detail in section 3.2.1. Moreover, by continuing to use synthetic data, it is possible to build on top of the results presented in some of the other chapters of this thesis, since the workload is the same.

4.2.2. The Performance Metrics

The metrics employed towards this set of experiments remain the same as for the scalability evaluation of variable update strategies, presented in chapter 3. The metrics are: *the forward propagation time*, *the number of processed images per second in a forward pass*, *the forward and backward propagation time*, and *the number of processed images per second in a forward and backward pass*. They are presented in more detail in section 3.2.2.

These metrics are chosen for this set of experiments due to their ability to reveal the raw performance and scalability of a parameter update strategy primarily in relation to a changing

batch size. As specified in section 3.2.2, they are often employed in literature to showcase the performance of frameworks and Neural Networks alike.

4.2.3. The Evaluation Procedure

In order to evaluate the scalability of the Parameter Server strategy, for a fixed number of nodes, and a fixed batch size, the following strategy is employed: 15 iterations are used, out of which 5 are utilized towards warming up the system, and the other 10 are employed towards gathering the actual metric values. Each iteration refers to processing a batch of images. Such an experiment is executed for either the forward pass metrics or the forward and backward pass metrics, however, not both at the same time. Hence, two separate experiments are required to capture both types of metrics. This process is repeated for each batch size in the following sequence: 8, 16, 32, 64, 128 and 256. Since the effects of the underlying communication channel is highly relevant, each experiment is repeated twice: once for Ethernet and once for InfiniBand, using the same set of parameter values. Finally, in order to gain a better understanding of the way in which scalability is affected by the batch sizes, the experiments are repeated across the following range of nodes: 1, 2, 4, 8, 16, 24, 32 and 40. The final result for each of the experiments is computed by averaging the values obtained on each individual node involved in the computation. As each node measures the metrics locally, there can be some slight variation in the value, however, its value is small enough that it can be disregarded.

4.3. Experimental Setup

This section defines the experimental setup used for running this set of evaluations. The description refers to the hardware, software, the parameter update architecture, and concrete workload and framework configurations used.

4.3.1. Hardware

The core underlying hardware remains the same as for the parameter update architecture scalability experiments presented in section 3.3.

Experiments are executed across the following range of nodes: 1, 2, 4, 8, 16, 24, 32, 40. Moreover, each experiment is executed twice: once over Ethernet interconnections and once over InfiniBand interconnects, using the exact same set of experiment configuration parameters.

4.3.2. Software

The Operating System installed on the DAS-5 machines is CentOS 7. The Nightly TensorFlow release version 1.13.0.dev20190226 is used for running these experiments. It should be mentioned that the non-GPU compatible version of the Nightly build is used exclusively here, in order to ensure a proper CPU pinned Hard Placement. The TensorFlow benchmarking suite is employed (via the `benchmark_cnn.py` source) to maximize the precision of the gathered metrics, and minimize the overhead of capturing them.

The same high performance implementation of the ResNet50 Neural Network, as used in chapter 3 for the TensorFlow experiments, is also employed here.

Methodology for Spawning Distributed Machine Learning Clusters

In order to facilitate the distributed execution of TensorFlow across a cluster of nodes, a custom script is made use of. This script handles the task of spawning the TensorFlow processes involved in the evaluation process, managing them throughout their execution, and terminating any hanging instances after the execution finishes gracefully or otherwise. The script also handles any TensorFlow specific requirements and configurations for enabling the distribution process. The script is called `cluster_spawner_benchmark.py`, and is available in the thesis' repository.

4.3.3. The Parameter Update Architecture

As the focus of this set of experiments is to better understand the scalability of the Parameter Server architecture, the chosen architecture must find itself in this vein. Considering the results seen in chapter 3, the Hard Placement Collocated Parameter Server appears to be a good choice, since it performs best among its other Parameter Server counterparts. Moreover, the Hard Placement version of this type of variable update strategy should minimize the uncertainty and variability around the accuracy of the results, introduced by the operator to device binding heuristics of the Soft Placement approach.

As explained in section 3.3.3, in this architecture, each node involved in the computation, bar exactly one, hosts one and only one process. This process is of the *worker* type. The exception to this rule hosts two processes: a *worker* process and a *parameter server* process. Due to the nature of Mini-batch Stochastic Gradient Descent, the collocated worker and parameter server processes are unlikely to compete for resources to a remarkable extent. More specifically, the parameter server usually receives local gradient updates, and computes the global gradient after the end of the forward pass, and before the beginning of the backward pass. During this phase, the workers remain idle. The converse holds true for the forward and backward passes.

4.3.4. Framework Configurations and Concrete Workload

Mini-batch Stochastic Gradient Descent is chosen as the optimization algorithm for this set of experiments, as it stands at the foundation of Deep Learning. The base learning rate is 0.1. Data is represented using the 32 bit floating point format.

The workload employed for these experiments consists of random images drawn from a distribution based on the ImageNet dataset. The images are generated by the benchmarking suite. Images are grouped into progressively greater batch sizes for each experiment, following the sequence: 8, 16, 32, 64, 128 and 256. An evaluation run consists of 15 batches, 5 of which are used for warming up the system, and the other 10 are used for gathering measurements.

The dimensions of a single randomly generated image are $300 \times 300 \times 3$ (Height $\times$ Width $\times$ Channels). Before being used as input, the images undergo a cropping process to reduce their size to $224 \times 224 \times 3$. The time taken to preprocess the images is not taken into account towards gathering the metrics. The input format of the images is *NHWC*.

4.4. Experimental Results

Section 4.4.1 presents a collection of key findings based on the results obtained studying the scalability of the Parameter Server architecture. The following sections present an in depth discussion of the aforementioned results. More specifically, section 4.4.2 presents the results obtained over Ethernet interconnects, and section 4.4.3 focuses in on the results obtained for InfiniBand interconnects.

4.4.1. Key Findings

From the set of executed experiments on the scalability of the Collocated Parameter Server, it is possible to extract a few interesting key points, which apply to the aforementioned architecture:

- Employing multi-node distribution always produces better results than a single node. This applies to both large and small batch sizes, with either Ethernet or InfiniBand connections. This can be seen in all the figures in this section, and is argued in both sections 4.4.2 and 4.4.3.
- The time per batch scales linearly relative to the batch size for the Parameter Server architecture (sub-linear for smaller batches, then linear for greater ones). This applies to both the forward pass and the forward and backward pass cases, over either Ethernet or InfiniBand connections (see for instance figures 4.3 or 4.5). The 16 node cluster seems to define a separation line between the smaller cluster sizes, which are strongly impacted by the size of the batch, and the bigger clusters, which are impacted to a

lesser extent (for instance, see figures 4.1 or 4.7). This point is further discussed in both sections 4.4.2 and 4.4.3.

- As discussed in section 4.4.2, in the Ethernet case, purely for the forward pass, the 16 node cluster appears to be the best choice, as more sizable clusters yield very little improvement in performance (see figures 4.1 and 4.2). In the forward and backward pass, the 32 node cluster is the best choice, as greater clusters yield worse performance (see figure 4.4).
- In the Ethernet case, no additional performance is gained for clusters with more than 32 nodes. In the forward pass, performance remains constant for larger clusters (see figure 4.2). For the forward and backward passes, performance decreases for larger clusters (see figure 4.4). This limited performance is likely due to the bandwidth limitations Ethernet imposes on the parameter server processes, a point which is reinforced in chapter 5.
- When the batch size is small (generally less than 32 images), the addition of more nodes in Ethernet based mediums, does not produce a significant amount of benefits. This is visible in figures 4.2 and 4.4. Hence, it usually does not make sense to use large clusters for small batch sizes when only Ethernet is available. This issue is further discussed in section 4.4.2
- In the tested InfiniBand mediums, the 40 node cluster performs best. This is less obvious for the forward pass case, where a noticeable improvement over the 32 node cluster is only visible for batch sizes greater than 128 (see figure 4.6). This is discussed in section 4.4.3. In the forward and backward case, on the other hand, this cluster is the clear winner (see figure 4.8). Generally speaking, evidence shows that the addition of more nodes is a good idea in InfiniBand mediums, as performance increases proportionally (see figure 4.8). This is further discussed in section 4.4.3 and confirms the conclusion drawn in chapter 3 on linear scalability in the number of nodes of the Parameter Server.
- An interesting phenomenon which occurs throughout many of the performed experiments refers to the logarithmic growth in terms of the number of processed images per second, which all of the tested configurations exhibit. This can be seen in both the InfiniBand experiments, for both pass combinations (see figures 4.6 and 4.8 respectively), and in the forward and backward pass case for Ethernet interconnects (see figures 4.2 and 4.4 respectively). In the Ethernet context this is likely due to the bandwidth limitations imposed by the interconnections. Evidence of this is presented in chapter 5. Moreover, as argued earlier, Ethernet proves to be detrimental for performance when more than 32 nodes are added (see figure 4.4). Performance saturation appears quite late for this interconnection type, at the 256 images per batch mark.
- In the InfiniBand case, performance saturation occurs at the 64 images per batch size (see figure 4.8). Due to the large bandwidth characteristic to FDR InfiniBand connections, communication overhead and bandwidth limitations are unlikely to be the source for this issue. Interestingly, as is seen in chapter 5, where the CPU and Memory consumption patterns are investigated more closely, hardware limitations do not seem to be the problem. Further research is required in order to discover the true explanation of the phenomenon. A potential hypothesis is that the Parameter Server architecture, has inherent flaws which limit its ability to scale for larger batches. This is further discussed in section 4.4.3.
- The logarithmic growth exhibited by the clusters for the InfiniBand case (see figure 4.8), paired with the results obtained in [37], regarding the superior generalization performance of Neural Networks trained on smaller batch sizes, leads to an interesting hypothesis. More specifically, using very large batch sizes can not only be detrimental to the generalization performance of a Neural Network, but it can also prove to have no significant benefits towards training Neural Networks faster. The clusters quickly saturate in performance when the batch size is increased. More specifically, past the 64

images per batch size, the clusters yield marginal performance improvements (see figure 4.8 and section 4.4.3). Further investigation of this phenomenon might yield some interesting results, which could be used to find the optimal batch size for the parameter server update architecture.

4.4.2. Ethernet Results

This section presents the results of the evaluations executed over an Ethernet connection for both the forward pass and the forward and backward pass experiments.

Forward Pass

Figures 4.1 and 4.2 show a graphical representation of the forward pass metrics, given Ethernet interconnections. These plots show the results obtained for TensorFlow's Hard Placement Collocated Parameter Server architecture.

Figure 4.1 shows the seconds per batch required to complete a forward pass. The plot reveals the impact of the node count on the batch time, across a wide range of batch sizes. Even at the 8 images per batch size, it is immediately visible that multi-node distribution can have a considerable positive impact on performance. The time per forward pass drops from 1.87 seconds, for a single node, to 1.34 seconds in the case of 2 nodes. The behavior is maintained up to the 8 node mark, with a time of around 0.9 seconds. Past the 8 node mark, for the size of 8 images per batch, no considerable improvement can be obtained from the addition of more nodes. This is likely due to the communication overhead incurred by the addition of more resources, an argument for which is presented in chapter 5.

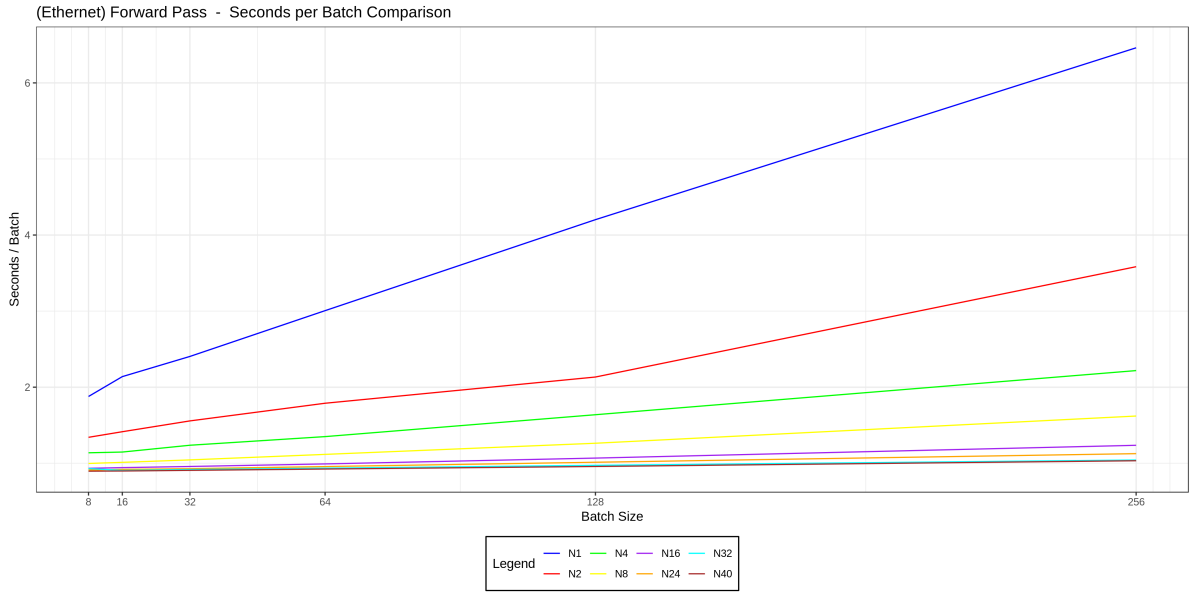


Figure 4.1: Number of seconds required in order to process a batch of images. Each line represents a cluster size. The x axis indicates the size of the batch. Lower values are better. Forward pass with Ethernet interconnects.

As the batch size increases, it becomes more obvious that single node computation falls severely behind multi-node computation. The time per forward pass of a single node cluster aggressively increases linearly in the batch size. For instance, it reaches a batch time of 4.2 seconds at the 128 images mark. This contrasts with the 2 node case, where a batch is completed in around 2.1 seconds. Moreover, the 8 node cluster yields a batch in 1.2 seconds and the 16 node cluster is able to produce a time of 1.06 seconds per batch. As the batch size continues to increase, it becomes more evident that an increased number of nodes is necessary in order to maintain the batch time small. The single node completes a 256 image batch in 6.46 seconds. In contrast, although its performance starts to deteriorate at this batch size, a 2 node cluster yields a batch time of 3.6 seconds. Similarly, the 8 node cluster completes a pass in 1.6 seconds, while the 16 node cluster yields a batch in 1.2 seconds.

The evaluation also reveals that the addition of more than 16 nodes to a cluster configuration yields marginal benefits. For instance, the 40 node case yields a time of 1 second for a batch size of 256 images. This would suggest that there is minimal return on investment past 16 nodes, and that the 16 node count might be the optimal cluster size for Collocated Parameter Server when only the forward pass is employed.

Figure 4.2 shows the number of processed images per second. This figure further reinforces the previous ideas. It indicates that both the 1 and 2 node clusters saturate in performance at the 128 images mark. The other cluster sizes show better performance, however, even so, there is a decrease in growth past this batch size. The 16 node cluster seems to perform best across all the tested configurations, having a linear growth in performance in the batch size. Clusters of greater sizes show marginal improvement in performance in comparison. For instance, 16 nodes process 207 images per second, while 40 nodes process 247 images. This is a 19% increase in performance for the addition of 150% more resources. Moreover, the 40 node cluster marginally outperforms the 32 node cluster, which would suggest that the computation is bandwidth limited by the Ethernet interconnects. This point is further argued in chapter 5, when a closer look is taken at this strategy's Network Traffic.

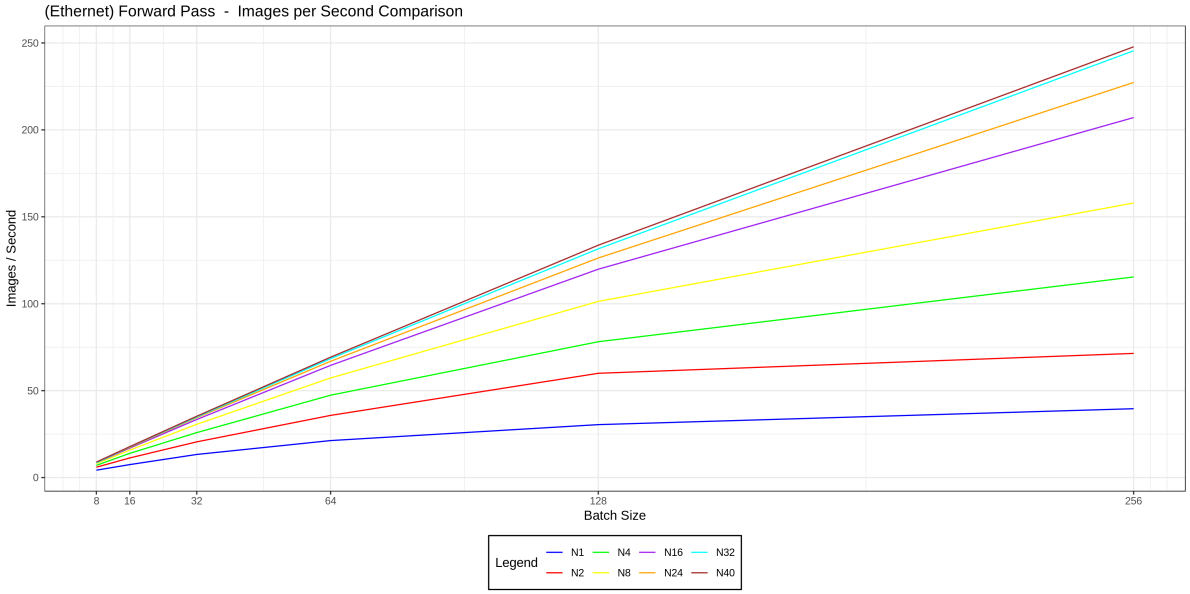


Figure 4.2: Number of processed images per second given a progressively greater batch size. Each line represents a cluster size. The x axis indicates the size of the batch. Greater values are better. Forward pass with Ethernet interconnects.

Forward and Backward Pass

Figures 4.3 and 4.4 show a graphical representation of the forward and backward pass metrics, given Ethernet interconnections. These plots show the results obtained for TensorFlow's Hard Placement Collocated Parameter Server architecture.

Figure 4.3 shows the seconds per batch when both a forward and backward pass are employed. The times per batch results are similar to their forward pass counterparts. The single node cluster performs considerably worse than the 2 node case across all batch sizes. The benefits of multi-node distribution can be seen as early as the 8 images batch size, with a considerable improvement in batch time. More specifically, the time is improved from 4.35 seconds to 2.56 seconds. As the batch size is increased, the batch time increases linearly for for all of the tested configurations. Performance degenerates quite quickly for the single node case, with a time of 40.5 seconds at the 128 images per batch size. This contrasts with the 2 node case, which completes a batch in 13 seconds. Interestingly, the batch time appears to increase less aggressively for the single node case past the aforementioned batch size. Intuitively, for the single node case, the benefits of greater node counts can be seen better when the batch size is large, and communication is less frequent. This does not apply to InfiniBand mediums, however, as seen in section 4.4.3.

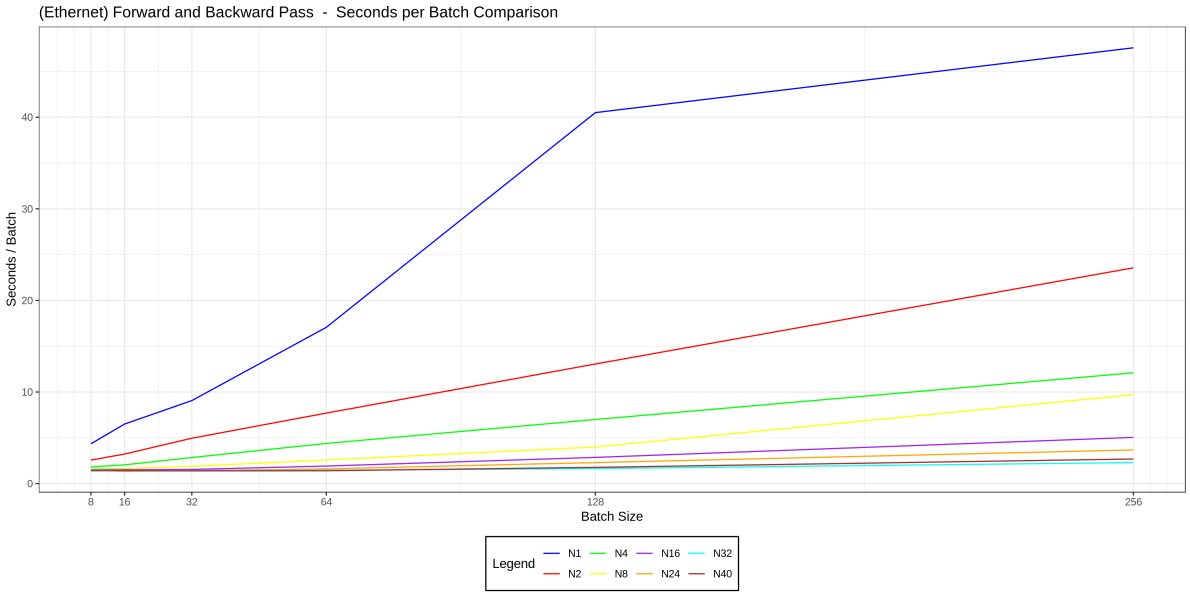


Figure 4.3: Number of seconds required in order to process a batch of images. Each line represents a cluster size. The x axis indicates the size of the batch. Lower values are better. Forward and backward pass with Ethernet interconnects.

Once more, at first sight, similar to the forward pass case, the 16 node cluster appears to be quite efficient, as it considerably outperforms the smaller configurations, while still producing times very close to those yielded by greater clusters. However, unlike the forward pass case, where the differences in time were small both in the relative and absolute, for the forward and backward case, these differences are small in the absolute but large in the relative. For instance, at the 256 images per batch mark, the 16 node configuration completes a batch in 5 seconds, while the best performing configuration of 32 nodes finishes a batch in 2.3 seconds. In the case of this particular comparison, the addition of 100% more resources, from 16 nodes to 32, yields a performance increase of 117%. The 8 node configuration completes a batch in 9.7 seconds. This is an increase of resources of 100% with a performance increase of 94%, in terms of batch time. It would thus appear that 32 nodes might be the best cluster configuration.

An interesting fact to point out is that the 40 node cluster performs worse than a 32 node cluster. A similar version of this phenomenon was also seen for the forward pass case, when the 32 node cluster performed very nearly as well as the 40 node configuration. The reason behind this behavior is very likely the bandwidth limitation imposed by the Ethernet interconnections, a point which is more strongly argued in chapter 5. This would suggest that 32 nodes is the maximal cluster size before performance starts to drop for the Collocated Parameter Server architecture, in either of the possible pass configurations.

Figure 4.4 shows a more clear picture in terms of the number of processed images per second for each cluster configuration. The plot clearly shows that the addition of more nodes in the cluster leads to a greater number of processed images. This is especially true for the batch sizes greater than 64. As reference, the single node case processes a total of 5.3 images per second when the batch size is 256. The 16 node cluster yields 50.7 images per second. The 32 node configuration performs best, yielding a total of 111.5 images per second, more than twice of what the 16 node cluster produces. As previously discussed, the 40 node cluster performs as well as the 32 node configuration up to the 64 images per batch size. After this point, the latter cluster significantly underperforms, with 95.1 images per second at the 256 images per batch mark.

At the 64 images per batch mark, both the 8 and 16 node clusters produce good results. Clusters of smaller size saturate quickly relative to the number of processed images per second, while clusters of greater size are unable to benefit from the additional resources likely due to the high communication frequency.

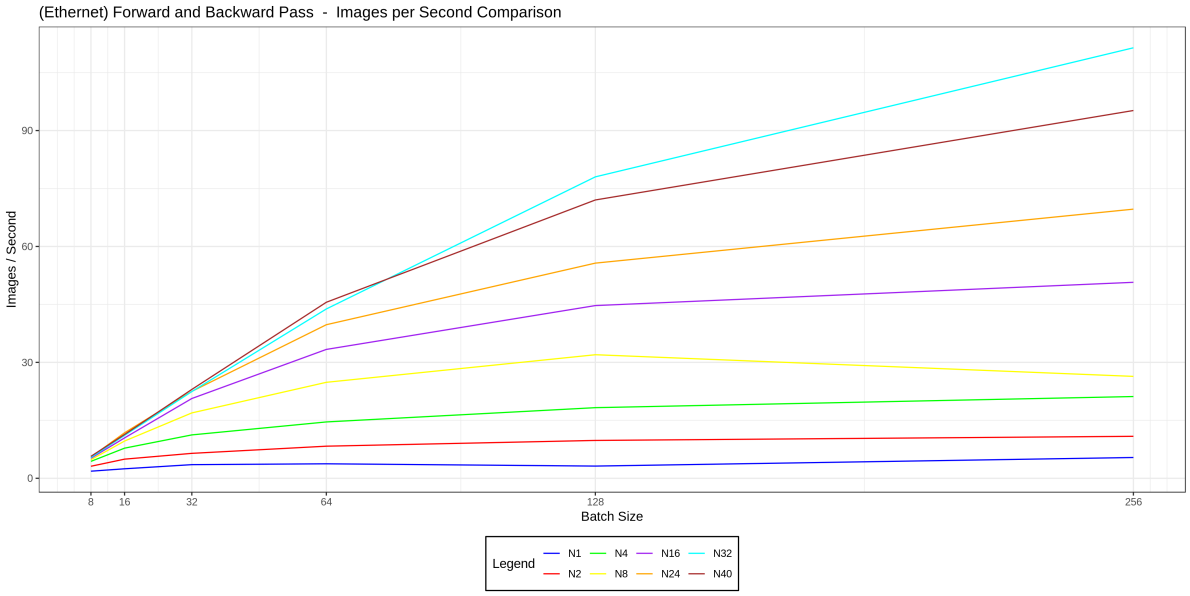


Figure 4.4: Number of processed images per second given greater batch sizes. Each line represents a cluster size. The x axis indicates the size of the batch. Greater values are better. Forward and backward pass with Ethernet interconnects.

Consequently, for batch sizes greater than 64, the 32 node cluster configuration fares best, while for batch sizes lower than or equal to 64, both the 8 and 16 node clusters yield the a high degree of performance.

It should be noted, however, that all clusters tend to show a logarithmic growth relative to the number of images per batch. Judging by the images per second curves, there is a high probability that for experiments past the 256 batch size, the clusters would be unable to produce significantly better results This suggests that Ethernet limits the performance of the Parameter Server architecture quite severely, and that with greater batch sizes there would be little improvement in the number of processed images per second, past a certain point.

Perhaps obvious, both figures 4.2 and 4.4 exemplify the need of greater batch sizes for large clusters in the context of bandwidth limited mediums.

4.4.3. InfiniBand Results

This section presents the results of the evaluations executed over an InfiniBand connection for both the forward pass and the forward and backward pass experiments.

Forward Pass

Figures 4.5 and 4.6 show a graphical representation of the forward pass metrics, given InfiniBand interconnections. These plots show the results obtained for TensorFlow's Hard Placement Collocated Parameter Server architecture.

Figure 4.5 describes the seconds per batch required to complete a forward pass for different types of cluster configurations. Similar to the Ethernet case, distribution significantly improves performance, even for small batch sizes. The single node case yields a per batch time of 1.88, while the 2 node cluster nearly halves the time to 0.96 seconds.

As the batch size increases, the batch time intuitively increases as well. The growth is less significant for clusters that are bigger than 8 nodes. The growth for smaller clusters is non-trivial, however, with the single node case being particularly affected. For instance, at the 8 images mark, the single node configuration yields a time of 1.88 seconds, while at the 256 images mark it produces a time of 6.48 seconds. In comparison, the 16 node case produces a time of 0.14 seconds for the 8 image batch, and 0.42 seconds for the 256 image batch size. Moreover, in terms of pure batch time, the addition of more resources past 16 nodes does not produce proportional results. The time for the 256 image batch for the 40 node cluster is 0.22 seconds. That is, with 1.5 times more resources, the time obtained is

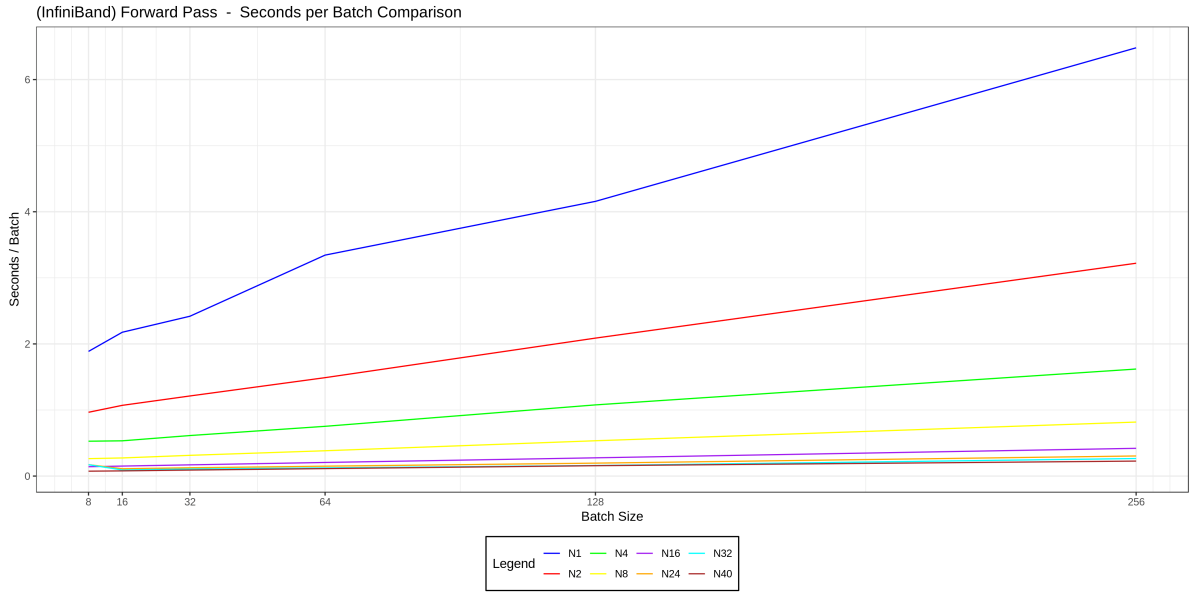


Figure 4.5: Number of seconds required in order to process a batch of images. Each line represents a cluster size. The x axis indicates the size of the batch. Lower values are better. Forward pass with InfiniBand interconnects.

only 0.52 times better.

Figure 4.6 better reveals the raw performance of the strategies in terms of number of processed images per second. While all strategies show good growth, it is immediately obvious that clusters with more nodes are able to process considerably more images. For instance, the 16 node cluster processes 310.5 images for a batch of 64 and 610.5 images for a batch of 256. In contrast, the 4 node cluster processes 85 images for a batch of 64 and 158 for a batch of 256 images. For this particular example, the 16 node cluster exhibits a growth of 1.96 times, whereas the 4 node cluster shows a growth of 1.85.

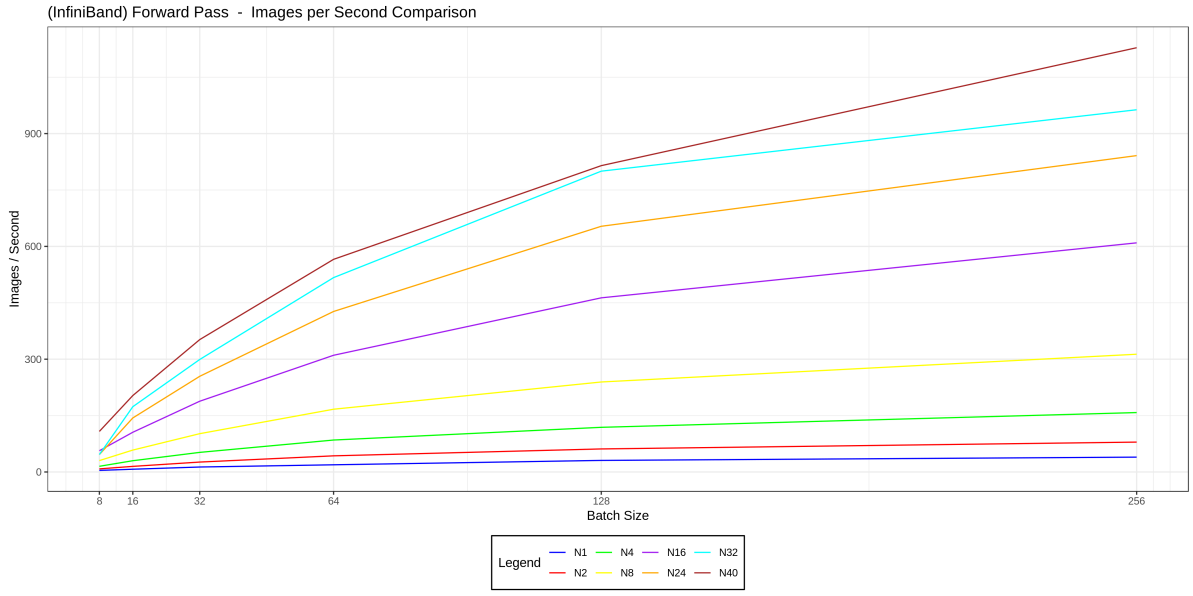


Figure 4.6: Number of processed images per second given greater batch sizes. Each line represents a cluster size. The x axis indicates the size of the batch. Greater values are better. Forward pass with InfiniBand interconnects.

While all cluster configurations show a constant growth in the number of images they are able to process, as in the case of the forward and backward Ethernet experiments, the

number of processed images exhibit a logarithmic growth. This is especially visible for smaller clusters. This phenomenon was seen in the forward and backward pass in Ethernet mediums (see section 4.4.2), and in the forward and backward pass experiments for InfiniBand, as is seen in the next section.

A surprising phenomenon is that the 40 node cluster and the 32 node configuration behave highly similar to each other. The addition of 8 extra nodes does not seem to yield any remarkable improvement for the forward pass. This is especially true for batch sizes which are smaller or equal to 128 images. For instance, at the aforementioned batch size, the number of processed images for the 32 node case is 800.1 images, while for the 40 node case it is 814.7 images. This performance improves at the 256 images batch size. The 32 node cluster yields a total of 963.45 images, while the 40 node cluster processes 1128.4.

Forward and Backward Pass

Figure 4.7 and 4.8 show a graphical representation of the forward and backward pass metrics, given InfiniBand interconnections. These plots show the results obtained for TensorFlow's Hard Placement Collocated Parameter Server architecture.

Figure 4.7 is consistent with what has been seen in this chapter so far in terms of employing multi-node distribution. The single node configuration exhibits much worse performance than the other strategies. All configurations exhibit linear scalability of the batch time, relative to the batch size. For instance, the single node case takes 4.2 seconds to process a batch of 8 images, and 56 seconds for a 256 image batch. A cluster of 4 nodes requires 1.1 seconds for 8 images, and 11.7 seconds for 256 images. Similarly, a configuration of 40 nodes yields 0.14 seconds for an 8 image batch, and 1.93 seconds for 256 images. As in the previous cases, there is a visible divide in performance between the clusters which have less than 16 nodes, and those that have 16 nodes or more. For instance, the difference in the batch time at the 256 image mark between 16 and 8 nodes is 2.9 seconds. The same time difference between 24 and 16 nodes is 0.04 seconds. A further example is the difference between 32 nodes and 24 nodes, which is 0.68 seconds. More specifically, the time difference is more noticeable at the lower end of cluster sizes.

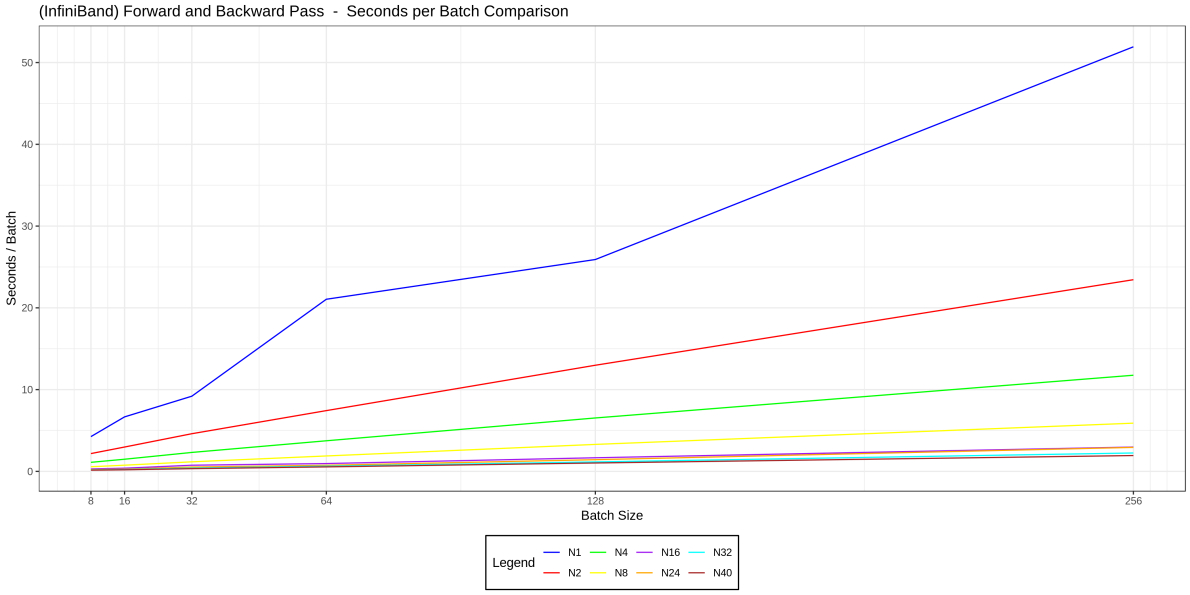


Figure 4.7: Number of seconds required in order to process a batch of images. Each line represents a cluster size. The x axis indicates the size of the batch. Lower values are better. Forward and backward pass with InfiniBand interconnects.

Figure 4.8 shows a clearer picture in terms of the raw performance of the different cluster structures. An interesting fact that can be pointed out immediately, is that all strategies seem to show a logarithmic growth. This phenomenon was noticeable for the both the Ethernet and forward InfiniBand experiments, however, in the former context it was less pronounced.

In the Ethernet case this was likely due to the limitations imposed by the available interconnects, which disallowed the clusters to reach near peak performance quickly. In the InfiniBand case, communication costs are unlikely to be the core cause for this issue. Both these points are supported by arguments presented in section 5. As will be further pointed out in chapter 5, it is also unlikely that the available CPU and Memory resources are the root cause of this behavior. This suggests that the Parameter Server's limited performance is due to some other issue, perhaps intrinsic to the architecture itself.

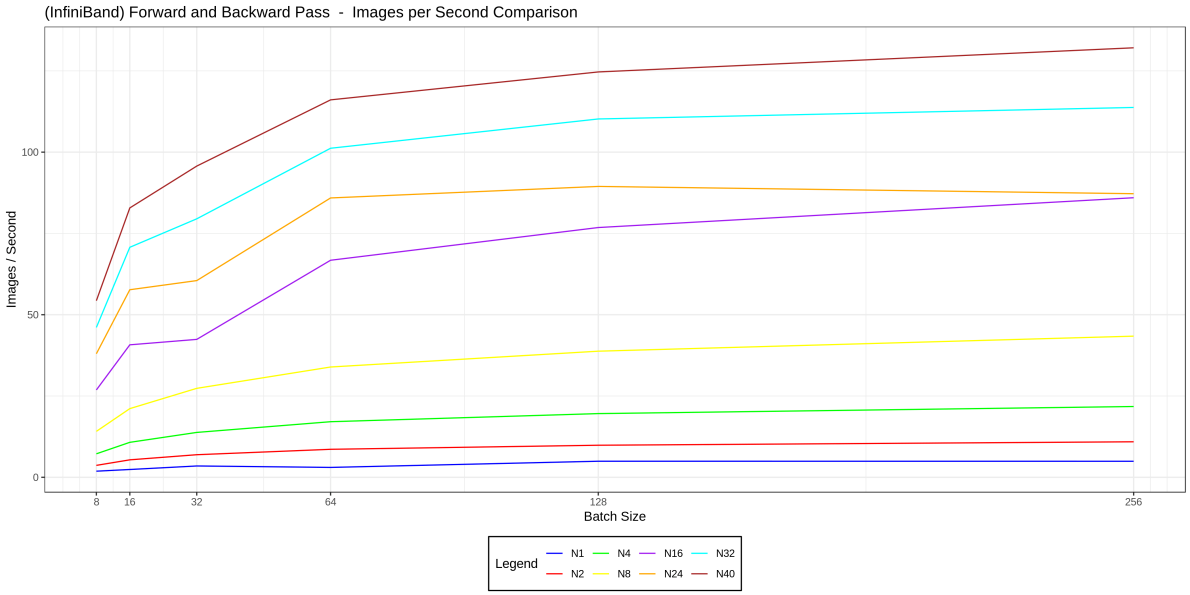


Figure 4.8: Number of processed images per second given greater batch sizes. Each line represents a cluster size. The x axis indicates the size of the batch. Greater values are better. Forward and backward pass with InfiniBand interconnects.

All clusters show high improvements as the batch size increases up to the 64 image mark. Past this batch size, however, the growth is limited. For instance, for the sequence of 64, 128, and 256 images per batch, the 40 node cluster processes 116, 124.7 and 132 images. Smaller clusters fare even worse. For example, a 4 node configuration yields the sequence of 17.1, 19.6, 21.8 processed images per second, given the aforementioned batch sizes. This would suggest that in spite of the large bandwidth, the Collocated Parameter Server architecture saturates fairly fast, and is unable to derive a considerable amount of benefit from larger batch sizes than 64 images.

A potential extension of this phenomenon builds on top of the research in [37]. More specifically, [37] shows that training with small batch sizes, equal to or below 32 images, yields better generalization power in models than when training on large batches. The experiments presented here extend this, and suggest that after a certain point, in this case 64 images, it makes little sense to increase the batch size, as there are no significant benefits of doing so in an effort to save time. This strengthens the hypothesis in [37], and supports the idea that training with large batch sizes, at least on CPU for the Parameter Server architecture, makes little sense.

4.5. Experiment Conclusions

RQ2: *how well does the Parameter Server strategy scale relative to the batch size and the number of nodes?* For brevity purposes, it should be mentioned that in what follows, scalability refers to the number of processed images per second, where not specifically mentioned otherwise:

- The time per batch scales linearly relative to the batch size for both the Ethernet and InfiniBand connections. The growth is inversely proportional to the size of the clusters.

- For Ethernet connections, Parameter Server is unable to scale up past 32 nodes. A 40 node cluster is unable to show any significant performance improvement for the forward pass, and shows decreased performance for both the forward and backward pass. This behavior applies across all the tested batch sizes, and is due to the bandwidth limitations imposed by Ethernet. This is consistent with what was seen in chapter 3.
- For InfiniBand connections, larger clusters generally yield better performance. This is especially true for the forward and backward pass case, where the addition of more nodes sees a linear increase in the number of processed images across all the batch sizes. In the forward pass case, experiments show that large clusters require progressively larger batches to outperform their smaller counterparts.
- The InfiniBand based clusters, which make use of either of the pass combinations show a logarithmic increase relative to the batch size. This is especially visible for the forward and backward pass, as in these experiments, past a batch of 64 images, no significant improvements are obtained. This behavior is unlikely to be caused by hardware limitations such as CPU, Memory or Network Traffic.
- The Ethernet based clusters, using either pass combination show signs of logarithmic growth. This is especially visible for the forward and backward pass case, where all clusters are affected.

5

Resource Consumption of Distributed Parameter Update Architectures

This chapter presents the large scale experiments executed in an effort to answer **RQ3**: *What are the CPU usage, Memory footprint, and Network usage patterns of the distributed parameter update strategies?*. Section 5.1 presents the chosen frameworks and Neural Network for this experiment. Section 5.2 presents the methodology behind the evaluation. Section 5.3 provides a concrete description of the experimental setup. Section 5.4 provides an overview of the obtained results. Finally, section 5.5 presents the conclusions of this chapter.

5.1. Selected Neural Network and Frameworks

The selected frameworks for these experiments remain the same as in chapters 3 and 4: Caffe2 and TensorFlow. Moreover, the selected network is ResNet50 which has also been chosen for the previous sets of experiments. The reasons for these choices are the same as in the aforementioned chapters, and for the sake of brevity, are not reproduced here.

5.2. Evaluation Methodology

5.2.1. The Workload

The workload for this set of experiments will be image based synthetic data. As argued in section 3.2.1, it has a number of advantages, such as ease of use, reproducibility and allows for other similar experiments to be executed, and be fairly compared to these ones. Moreover, continuing to choose synthetic data allows one to build on previous results, and obtain a more complete image of the architecture's performance.

5.2.2. The Performance Metrics

The performance metrics chosen for this set of experiments must be able to reveal the resource consumption of the various parameter server architectures across three dimensions: CPU usage, Memory footprint, and Network consumption. The chosen metrics are surveyed once every second. This particular period is chosen, as it should produce a fine grained view of the resources given the general duration of an experiment run. Moreover, the aforementioned interval should not incur significant strain on the CPU, and should leave the results unbiased:

- **CPU utilization**: refers to measuring the normalized amount of CPU resources used by the processes which are involved in the distributed training. Repeated measurements, at 1 second intervals, are executed all throughout the duration of an experiment. This strategy is applied to all the processes involved in the distributed training process, which should faithfully reveal the pattern that the tested parameter update strategies manifest on the available computational resources. As mentioned, this metric is normalized, hence it is represented as a *percentage*.

- **Memory footprint**: refers to the amount of Gigabytes of RAM used by the parameter update architectures. This metric is recorded every second by sampling the amount of memory employed by each processes involved in the distributed training process. The metric should reveal the memory utilization habits of the tested architectures. Straight forwardly, it is measured in *Gigabytes*.

- **Network Traffic**: refers to the amount of input and output data that is received and sent over the network channels. In order to gain a complete understanding of the communication patterns employed by the variable update strategies, this metric is recorded all throughout the duration of an experiment, at 1 second intervals. More specifically, the amount of data communicated in the past second is collected and reported at 1 second intervals. The measurements are made individually for outgoing and incoming traffic. The metric is reported in *kilobits / second*.

5.2.3. The Evaluation Procedure

In order to evaluate the resource consumption patterns of the tested variable update architectures, the following strategy is employed: 15 iterations are used, 5 out of which are used to warm up the system, and the other 10 are evaluation runs. Each iteration is a batch. Unlike the experiments presented in chapters 3 and 4, all iterations are monitored, that is, including the warm up ones. In order to minimize the overhead associated to measuring the metrics, not all 3 values are tracked during an experiment run. More specifically, both the *CPU utilization* and the *Memory footprint* are measured together, since there is no additional cost to measuring both as opposed to just one. The *Network Traffic* is measured independently of the other two. As a consequence, each experiment is executed twice: once for the CPU utilization and Memory footprint, and a second time for the Network Traffic. It is also interesting to see if the type of underlying interconnects has a significant effect on the aforementioned performance aspects. Hence, each experimented is executed once for Ethernet and once for InfiniBand. Finally, these experiments are repeated for each of the tested architectures, and for each cluster size in the sequence: 4, 8, 12, and 16. Moreover, in the case of the Non-collocated Parameter Server, an additional set of experiments on 5, 9, 13, and 17 nodes is conducted, where the additional node hosts a singular worker process. It is worth pointing out that all evaluation runs explained above employ both a forward and a backward pass for each batch. The reason behind this is that the bulk of computation, memory consumption and network communication is largely due to the backpropagation algorithm, which is the main part of the backward pass. Moreover, the forward and backward pass experiments implicitly survey the hardware demands of the forward pass, however, it is more difficult to isolate them from the backward pass.

To also gain a better understanding of what the forward pass looks like, forward pass experiments are executed for the Parameter Server and Distributed Replicated strategies. These experiments are exclusively conducted over a cluster of 16 nodes, using both Ethernet and InfiniBand connections, and are compared to their forward and backward counterparts. In addition to revealing the patterns of the forward pass, these results should provide some additional insight into why Parameter Server is unable to perform well in Ethernet based mediums (as seen in chapters 3 and 4), and why Distributed Replicated performs well when only its forward pass is used (as seen in chapter 3).

The batch size is chosen to be fixed at 128 images. The greater size should better reveal the resource consumption patterns of the update architectures, since the computation and communication phases should be longer and thus easier to observe.

The metric values are gathered locally in each participant process. Since keeping track of many time series can be distracting, the time series of the worker processes are aggregated into one, through an averaging process. This is possible since the workers follow the same patterns, with minor variations in their behavior. The parameter server or controller process is excluded from the averaging operation, as it follows different communication and computation patterns, and as a consequence, can skew the typical behavior of the working processes. Consequently, this process' time series is presented individually.

5.3. Experimental Setup

This section defines the experimental setup used for running this set of evaluations. The description refers to the hardware, software, parameter update architectures, and concrete workload and framework configurations used.

5.3.1. Hardware

The hardware employed for these experiments remains the same as in chapters 3 and 4. More specifically, the VU DAS-5 site is employed. The aforementioned site features 68 machines. The nodes used to run these experiments are fitted with Dual 8-core Intel Xeon E5-2630 v3 CPUs, providing a total of 32 threads and a frequency of 2.4 GHz. Each machine also features two 4 TB HDD storage devices, and 64 GB of RAM. Nodes are interconnected both via 56 Gbps FDR InfiniBand and 1 Gbps Ethernet.

Each evaluation is executed with the exact same configuration over both Ethernet and InfiniBand interconnects. Moreover, experiments are run over a range of cluster sizes in the sequence: 4, 8, 12 and 16 nodes.

5.3.2. Software

The Operating System installed on each machine is CentOS 7. As already specified, Caffe2 version 0.8.2, as well as the non-GPU compatible Nightly TensorFlow release version 1.13.0.dev20190226 are employed. The non-GPU compatible version of TensorFlow is used in order to ensure that CPU-only pinning is correct.

In the case of Caffe2, no additional software is used to run experiments, other than the main build. In the case of TensorFlow, the Convolutional Neural Network benchmarking suite is used in order to execute experiments. The benchmarking suite is not employed towards gathering the values of the tracked metrics, as it does not offer any functionality in this sense. The same applies to Caffe2, in the sense that the framework itself does not offer any means of tracking the metrics.

Methodology for Measuring Performance Indicators

As the frameworks offer no tools for measuring the discussed metrics, custom scripts are written in order to enable the experiments. One script handles the logging and collection of the memory footprint and the CPU usage via the `top` command. The other script handles the logging and collection of network statistics by periodically inspecting the `rx_bytes` and `tx_bytes` files stored in the `statistics` folder associated to each network interface. The `rx_bytes` and `tx_bytes` files are maintained by the Operating System. The script also performs some operations on the data retrieved from the aforementioned files, in order to yield the actual value of the metric. A process running the script is spawned for each process involved in the distributed evaluation task, and is terminated when the evaluation terminates. The network monitoring script is called `network_monitor.sh`, while the CPU and Memory script is called `monitor.sh`. Both of the aforementioned scripts are available on the thesis repository.

Additional scripts are used for both Caffe2 and TensorFlow. These spawn the processes involved in the evaluation process across the underlying cluster, as well as the monitor processes. The scripts track these processes throughout the entire execution and terminate any hanging instances after each execution, be it successful or faulty. Moreover, the scrips handle the complexity of configuring the frameworks in order to simplify the creation of a distributed training environment. The scripts are both called `cluster_spawner_benchmark.py`, with versions available for both TensorFlow and Caffe2. Both are available on the thesis repository.

5.3.3. The Parameter Update Architectures

For TensorFlow, the following variable update architectures are chosen: Collocated Parameter Server, Non-collocated Parameter Server, Distributed Replicated, Distributed Allreduce, and Collective Allreduce. These strategies have been detailed in section 2.5.2. Distributed Replicated is tested in two flavors of the underlying reduction strategy: Ringx1 and PSCPU.

It is worth reinforcing that only the CPU pinned Hard Placement versions of the strategies are used. In terms of Caffe2, the default version of Ring Allreduce is used, as it is the only available architecture.

As a reminder, parameter update strategies are used towards enabling the distributed learning process. These strategies define a means of collecting the local parameter updates, which are obtained at the end of a batch, and generally consist of gradients. The updates are local since the nodes train on disjoint data subsets. Consequently, these local updates must be aggregated in an effort to obtain a global update. The aggregation process usually consists of averaging the updates together. The parameter update strategies also disseminate the global update to the individual workers, all of which then update their local copy of the model in a uniform fashion. These strategies are essential to distributed Machine Learning, as they drive the learning process. As a consequence, looking into their hardware consumption patterns is essential towards better understanding their performance, and hardware demands. In turn, these can reveal the potential areas of improvement for these architectures.

5.3.4. Framework Configurations and Concrete Workload

The workload and setup is the same as was presented in section 3.3.4. The only exception to this refers to the 128 images per batch size for TensorFlow. The batch size for Caffe2 remains 32 images, as it is not possible to conduct the experiments with greater sizes due to timeouts.

15 batches are used for both frameworks: 5 batches for warming up the system and 10 batches for evaluation. However, since the hardware metrics are tracked externally, and not by the frameworks themselves, both stages are used for the evaluation.

The concrete configurations of the evaluated parameter update strategies remain the same as in section 3.3.4. Moreover, table 3.1 provides a concise description of the configurations for each of the update strategies employed.

5.4. Experimental Results

Section 5.4.1 presents a collection of key findings based on the results obtained by surveying the hardware consumption patterns of the tested parameter update strategies. The following sections present an in depth discussion of the aforementioned results. More specifically, section 5.4.2 presents the results obtained by studying the Network Traffic, section 5.4.3 focuses in on the results obtained for the Memory Footprint, and section 5.4.4 looks into the CPU consumption of the parameter update strategies.

5.4.1. Key Findings

The set of executed experiments reveal the following observations:

- The Network Traffic of Caffe2, Collective Allreduce and Ringx1 Distributed Allreduce does not scale with the size of the cluster. This is unlike the other architectures which show an increased amount of network communication as the size of the cluster is increased (sub-linear in most cases, bar Parameter Server which scales linearly). Figure 5.2 exemplifies this. The commonality between Caffe2 (which uses Ring Allreduce), Collective Allreduce, and Ringx1 Distributed Allreduce, is that they all use a ring based reduction method, where the amount of data communicated is independent of the number of nodes, as the strategy largely relies on unicast communication. It should be clarified that this refers to the amount of data transmitted from one node to another, and not the cumulative amount of data sent through the network. The communication complexity of ring based methods is, however, linear in the number of model parameters. Unlike the aforementioned strategies, Parameter Server, Distributed Replicated and PSPCU Distributed Allreduce all rely on the parameter server arch-type. Implicitly, in this arch-type, the updates are communicated by the parameter server process to each individual worker. As a consequence, the amount of data communicated by these strategies is linear in the number of worker processes. This issue is further discussed in section 5.4.2.
- All architectures, bar Caffe2 and Collective Allreduce, benefit from the addition of In-

finiBand. The aforementioned architectures use well below the bandwidth of Ethernet, as can be seen in figure 5.1, which supports the idea that InfiniBand yields no benefits to them. This is very likely due to their ring based reduction strategies. The other architectures show an increase in the network traffic when InfiniBand is added. This is especially visible for Parameter Server (see figures 5.3c, 5.3d and 5.3e), Distributed Replicated (see figure 5.3f) and PSCPU Distributed Allreduce (see 5.3g). All the aforementioned strategies rely on a parameter server process. The centralized nature of the parameter server, which communicates with each individual worker, combined with the synchronous communication pattern, can turn the parameter server into a bottleneck. Hence, InfiniBand greatly benefits these architectures. This is further discussed in section 5.4.2.

- A further look into the differences in network traffic for Parameter Server and Distributed Replicated given the two different passes, reveal results which strongly correlate and reinforce the conclusions reached in chapter 3 (for Parameter Server and Distributed Replicated) and chapter 4 (for Parameter Server). Namely, when the parameter server process is highly dependent on bandwidth, the strategy's scalability and performance is limited. This is visible in figures 5.4 and 5.5. Moreover, this is further discussed in section 5.4.2.

- Parameter Server is especially dependent on bandwidth. It is its major bottleneck, as can be clearly seen when comparing figures 5.1e and 5.2d. While the workers make moderate use of the available bandwidth, the parameter server processes require a considerable amount of bandwidth in order to enable quick communication and avoid forming plateaus in the network traffic patterns (see figure 5.1e). As mentioned before, parameter server processes employ point-to-point communication with each of their workers. This implies that network traffic grows linearly in the number of worker processes, which supports the ideas forwarded in chapters 3 and 4 that Parameter Server cannot scale well in Ethernet based mediums due to bandwidth limitations.

- InfiniBand and cluster size exhibit no influence on the memory footprint of the tested architectures. This is further discussed in section 5.4.3, and is visible in figures 5.6 and 5.7.

- Caffe2 uses the least amount of memory among all the tested architectures, using less than 1 GB in all of the tested scenarios. Figures 5.6a and 5.7a show this. The issue is discussed further in section 5.4.3.

- While all of TensorFlow's architectures show relatively similar memory consumption patterns, Collective Allreduce proves to consume the highest amount of memory. Moreover, its memory consumption appears to be a function of the elapsed training time. More specifically, as the elapsed training time increases, the peak memory consumption of the strategy increases as well. This is visible in figures 5.6b and 5.7b. The subject is further discussed in section 5.4.3.

- Tests on Parameter Server and Distributed Replicated reveal that memory consumption is very low for forward passes, when compared to the footprints of the backward passes. This phenomenon is visible in figures 5.8 and 5.9, and is further detailed in section 5.4.2.

- Caffe2 is bottlenecked by its CPU consumption (see figures 5.10a and 5.12a), which is a likely cause behind the strategy's inability to scale, as seen in chapter 3. This argument is further reinforced by Caffe2's low CPU consumption in the single node case (see figure 5.11), which in such a setting performs better than any other strategy (see chapter 3). Hence, it appears that CPU consumption directly correlates with Caffe2's performance: when CPU consumption is low, Caffe2 performs well, when the CPU consumption is high Caffe2 consistently underperforms.

- Bar the Parameter Server strategy, the CPU consumption of all of TensorFlow's parameter update strategies do not appear to be influenced by the size of the cluster or the InfiniBand interconnects. None of the strategies maximize the CPU usage, nor do they employ the CPU resources at high capacity for prolonged periods of time. These aspects are further discussed in section 5.4.4. Moreover, these issues are exemplified in figures 5.10 and 5.12.
- As seen in figures 5.12c, 5.12e and 5.12d, the Parameter Server strategy is influenced by the available bandwidth. More specifically, CPU consumption visibly increases when more bandwidth is available. For instance, figure 5.3e clearly shows that the parameter server processes are highly dependent on the underlying communication channel. InfiniBand offers a considerable amount of bandwidth, which enables quick communication between parameter server and workers. This likely enables the CPU resources to be used more efficiently, since data is more readily available for both parties. This issue is further discussed in section 5.4.4.
- Experiments which focus on the difference between the forward and backward pass of the Distributed Replicated and Parameter Server architectures (both parameter server arch-type methods), show further correlation between the available bandwidth and the cluster's ability to scale. More specifically, when pairing the results obtained in sections 3.4.2 and 3.4.2 with those in section 5.4.2, it is possible to observe that when the parameter server process is limited by bandwidth, the cluster does not scale. This is further evidence that bandwidth limitations impede parameter server based architectures from scaling.

5.4.2. Network Traffic Results

This section presents the results of the Network Traffic evaluations for the tested parameter update strategies. For the sake of brevity, only a part of the results are presented: a comparative look at the footprints for two differently sized clusters, for both Ethernet and InfiniBand interconnects, and a survey of the network traffic across equally sized clusters in different communication mediums. An additional subsection also takes a look at the different network consumption patterns of the Collocated Parameter Server and Distributed Replicated strategies given different pass combinations, i.e. only forward or both forward and backward. This latter comparison is carried out for both Ethernet and InfiniBand.

8 Nodes vs. 16 Nodes

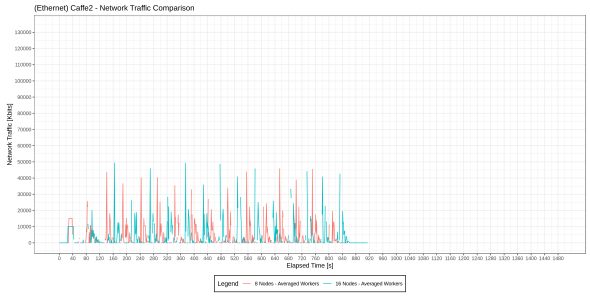
Figure 5.1 provides a comparative look at the Network Traffic patterns of the different parameter update strategies. The tests focus on clusters of different sizes: 8 and 16 nodes respectively. Two such cases are presented: one for Ethernet and one for InfiniBand.

Figure 5.1a shows the Caffe2 Ethernet Network Traffic. The strategy makes moderate use of the available bandwidth, and does not show any significant changes as the size of the cluster doubles. Among the tested architectures, its peaks are the lowest.

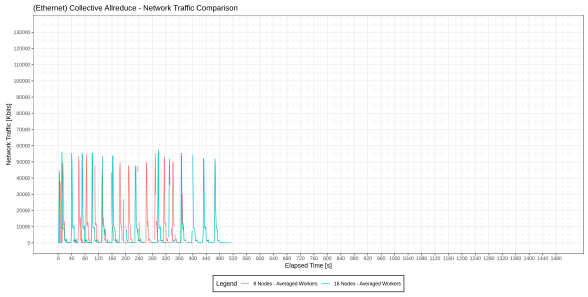
Collective Allreduce, presented in figure 5.1b, also shows modest use of the available bandwidth, which is likely one of the reasons behind it performing remarkably well for Ethernet based mediums, as was seen in chapter 3. Moreover, the size of the cluster does not appear to be a factor which influences the Network Traffic in any way.

The Parameter Server strategy shows that it is highly dependent on the available bandwidth. This phenomenon can consistently be seen in figures 5.1c, 5.1d, and 5.1e, as well as in the other tested cluster configurations. As can be seen, however, the worker processes make little use of the available bandwidth. The parameter server process, however, employs all of the available bandwidth. Moreover, it is forced to wait for more bandwidth to become available in order to send any remaining data, which is the reason for the plateaus visible in all the aforementioned figures. The limited bandwidth that the parameter server process is forced to deal with is arguably the main reason why the architecture is unable to scale well for Ethernet based mediums, as was seen in chapters 3 and 4.

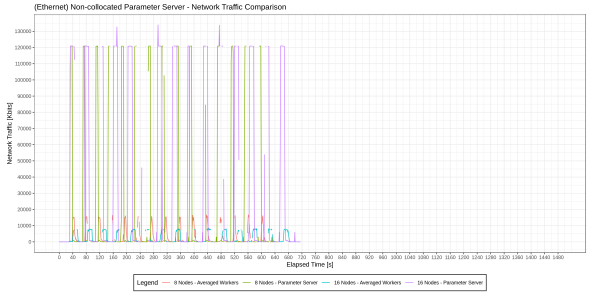
The Distributed Replicated architecture (see figure 5.1f) shows a similar behavior to the Parameter Server strategy, wherein the parameter server process is severely limited by the



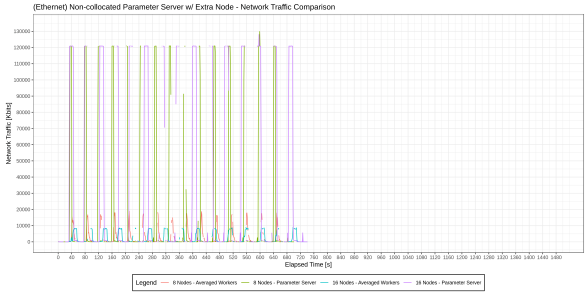
(a) Caffe2



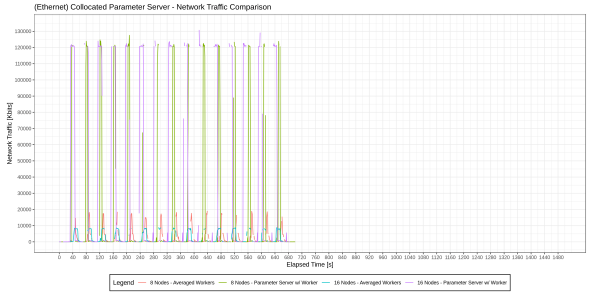
(b) Collective Allreduce



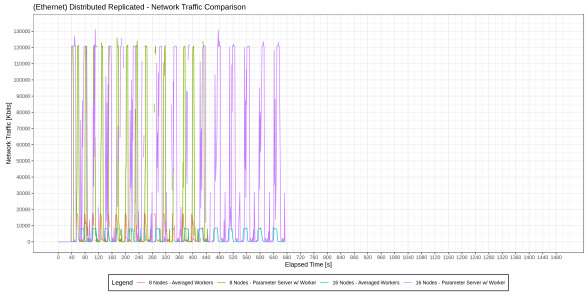
(c) Non-collocated Parameter Server



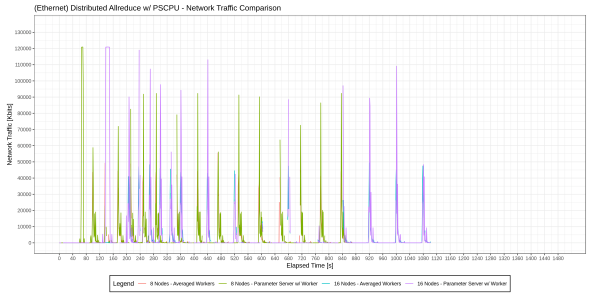
(d) Non-collocated Parameter Server with extra worker



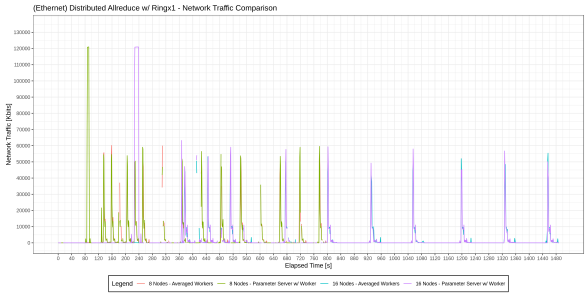
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure 5.1: The Network Traffic, in terms of transmitted kbits, of the parameter update strategies for a cluster of 8 nodes and a cluster of 16 nodes, both using Ethernet interconnects. The x axis represents the elapsed time in seconds. The y axis represents the amount of transmitted kbits. Both the x axes and the y axes span the same range in each plot. The time series of all the worker processes are averaged together into one time series. The parameter server's time series is not averaged into the aforementioned time series.

available bandwidth. The workers use up very little bandwidth. As in the case of Parameter Server, this is very likely the reason why Distributed Replicated underperforms in Ethernet based mediums when both the forward and the backward passes are used, and reinforces the results presented in chapter 3.

While the Ringx1 Distributed Allreduce strategy does not show signs of being affected by the size of the cluster, as shown in figure 5.1h, the same cannot be said concretely about its PSCPU counterpart presented in figure 5.1g. In both strategies, however, the controller process exhibits an initial spike at the beginning of the training phase, which reaches the Ethernet bandwidth limit, however it does not occur more than once. Moreover, unlike some of the other strategies, there is no discernible difference between the workers and the controller processes.

While it is possible to conclude that the size of a cluster does not exert any influence on the Network Traffic for Caffe2, Collective Allreduce or the Ringx1 Distributed Allreduce strategy, the bandwidth limitations of Ethernet makes it difficult to reach a definite conclusion in the case of Parameter Server, Distributed Replicated and PSCPU Distributed Allreduce. Hence, it is worth inspecting the Network Traffic pattern across clusters of different size both using InfiniBand interconnects. These results are presented in figure 5.2.

Figures 5.2a and 5.2g reinforce what has been seen in figure 5.1, namely that these architectures are not influenced by the cluster size. No figure is present for Caffe2 due to repeated timeouts during the evaluation process.

Figures 5.2b, 5.2c and 5.2d reveal that a greater cluster does impact the Network Traffic. This is intuitive, since the parameter server process must distribute data to more worker processes. In fact, the network traffic is likely to be predicted by a linear function parameterized by the number of worker processes, since the parameter server employs point-to-point communication to update its workers after the global parameter update is computed. It should be noted, however, that the traffic in the worker processes remains the same.

Distributed Replicated, presented in figure 5.2, is also susceptible to more traffic when more nodes are in the cluster. This architecture also depends on a parameter server, which is also tasked with computing the global parameter updates. Hence, it is in a similar situation with the Parameter Server architecture. This behavior is likely one of the factors behind Distributed Replicated's inability to scale very well when using both forward and backward passes, as discussed in chapter 3.

PSCPU Distributed Allreduce also confirms that it is influenced by the size of the cluster, as seen in figure 5.2f. This is intuitive, since the PSCPU strategy implies that the gradients are gathered in the CPU of the controller process, averaged, and then disseminated to the participating worker processes, as described in section 2.5.2. Hence, the Network Traffic is once more a function of the number of nodes.

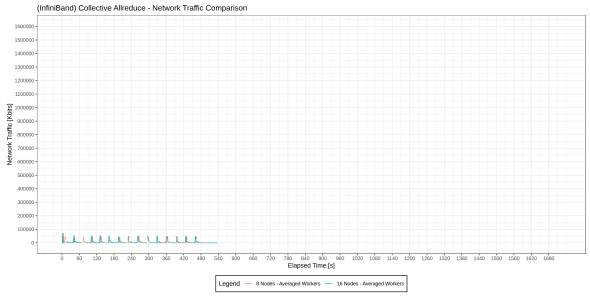
As a conclusion to this subset of experiments it is possible to say that Caffe2, Collective Allreduce and Ringx1 Distributed Allreduce are not influenced by the size of the cluster, whereas the other strategies are. The explanation of why the aforementioned three architectures are not influenced by the cluster size is straight-forward: they all use a ring based reduction algorithm, which employs mostly unicast communication. In such cases, the communication between a pair of processes is only dependent on the number of model parameters - as gradient updates are the core data that is being communicated -. The other architectures are generally based on a Parameter Server paradigm, which implies frequent point-to-point communication between the parameter server and its workers.

Ethernet vs. InfiniBand

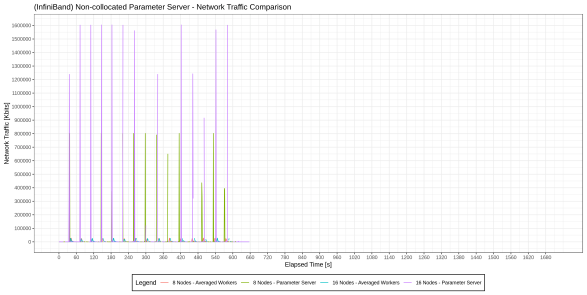
Figure 5.3 provides a comparative look at the Network Traffic patterns of the different parameter update strategies. The tests focus on two clusters, both having 8 nodes, each, however, using a different type of interconnections: Ethernet and InfiniBand.

Figure 5.3a shows that Caffe2 does not derive any benefit from the addition of InfiniBand connections. This reinforces what was seen in chapter 3, where the Ring Allreduce strategy, which stands at the foundation of Caffe2 distribution, did not see any improvement when more bandwidth was added.

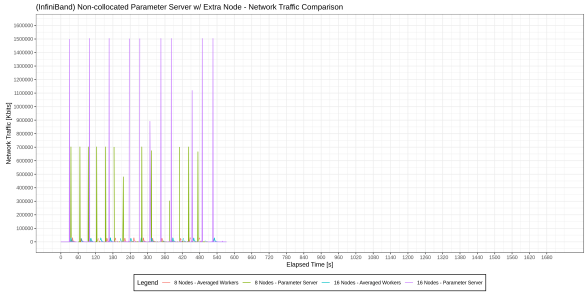
Collective Allreduce, does not use up more bandwidth when InfiniBand is added. It's



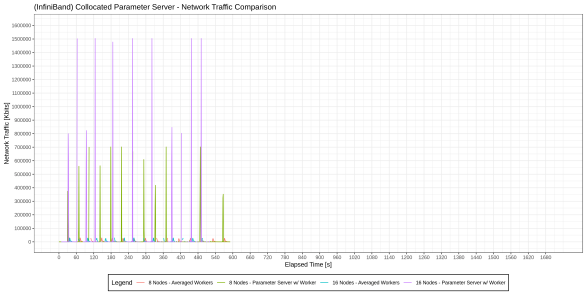
(a) Collective Allreduce



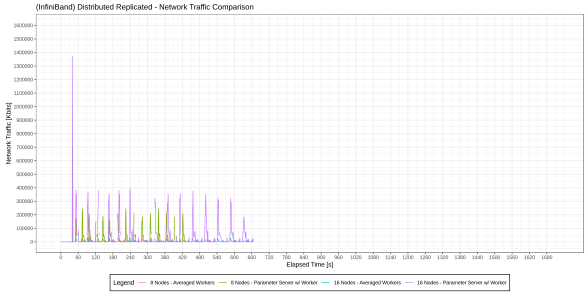
(b) Non-collocated Parameter Server



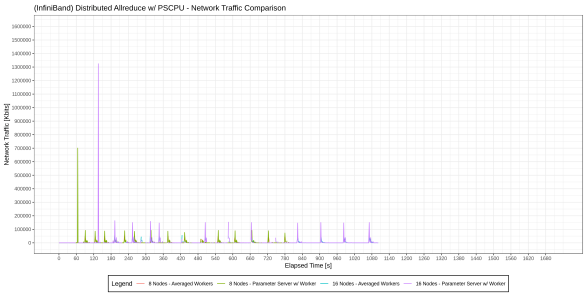
(c) Non-collocated Parameter Server with extra worker



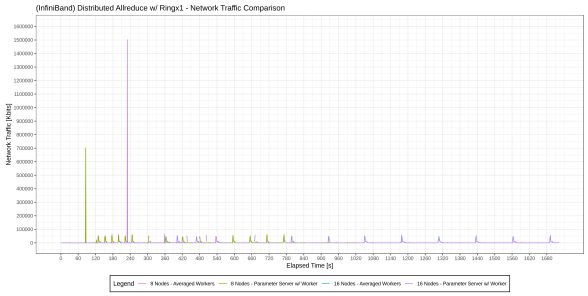
(d) Collocated Parameter Server



(e) Distributed Replicated

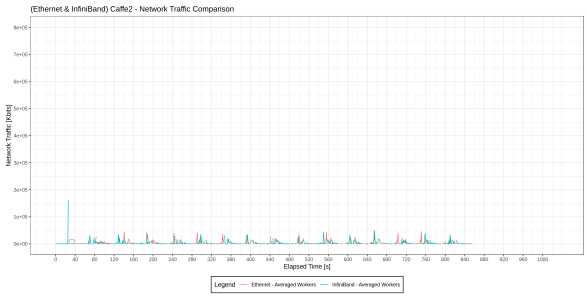


(f) Distributed Allreduce with PSCPU reduction strategy

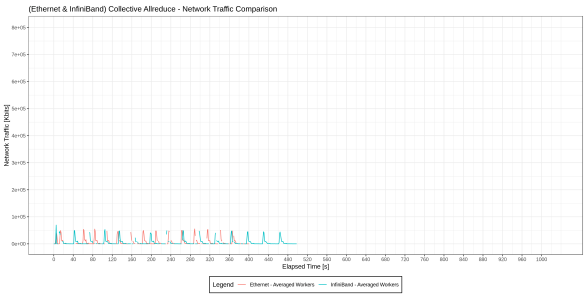


(g) Distributed Allreduce with Ringx1 reduction strategy

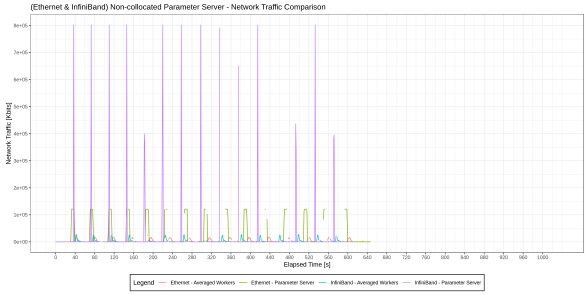
Figure 5.2: The Network Traffic, in terms of transmitted kbits, of the parameter update strategies for a cluster of 8 nodes and a cluster of 16 nodes, both using InfiniBand interconnects. The x axis represents the elapsed time in seconds. The y axis represents the amount of transmitted kbits. Both the x axes and the y axes span the same range in each plot. The time series of all the worker processes are averaged together into one time series. The parameter server's time series is not averaged into the aforementioned time series.



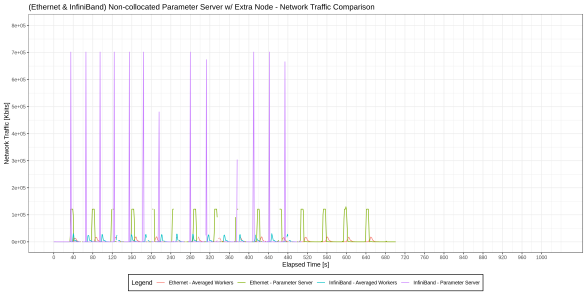
(a) Caffe2



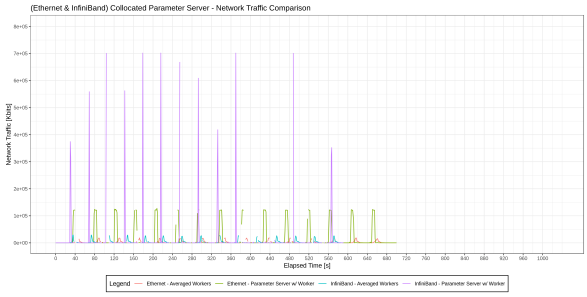
(b) Collective Allreduce



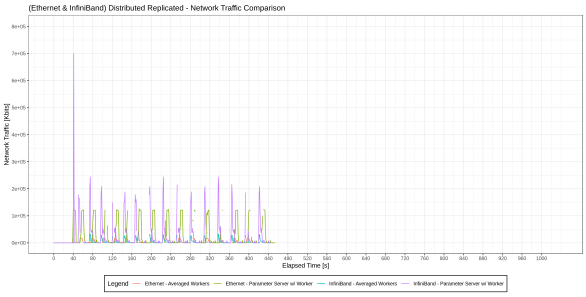
(c) Non-collocated Parameter Server



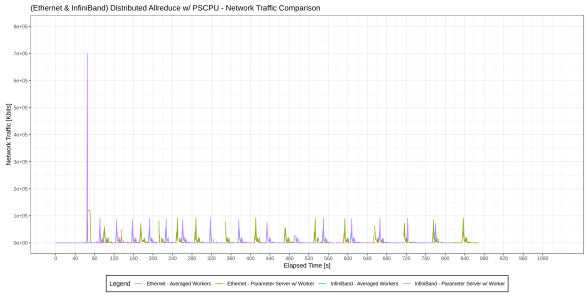
(d) Non-collocated Parameter Server with extra worker



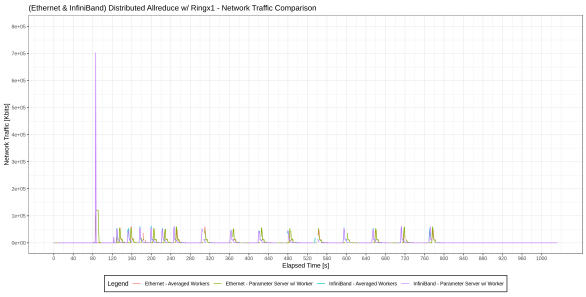
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure 5.3: The Network Traffic of the parameter update strategies for two clusters of 8 nodes using Ethernet and InfiniBand interconnects respectively. The x axis represents the elapsed time in seconds. The y axis represents the normalized Network Traffic in kbits. Both the x axes and the y axes span the same range in each plot.

traffic remains the same, as seen in figure 5.3b. This is not surprising, as the performance of Caffe2 is constant across Ethernet and InfiniBand, as proved in chapter 3. Hence, the low communication overhead of the architecture is one of the core reasons for its high efficiency across both communcation mediums.

The Parameter Server strategies benefit greatly from the addition of InfiniBand connections, as seen in figures 5.3c, 5.3d and 5.3e. It should be noted that this applies particularly to the parameter server processes, which employ a great deal of communcation between the parameter server and each individual worker. The worker processes continue to use low amounts of bandwidth. This reinforces the evidence seen in chapters 3 and 4, where Parameter Server was shown to only be viable in InfiniBand based mediums. Judging by the high bandwidth consumption of the parameter server process, it can be hypothesized that the global gradient update communicated by the parameter server to each of its workers is the core cause of the high bandwidth demands. This in turn implies limited scalability when the bandwidth does not meet or exceed the needs of the cluster.

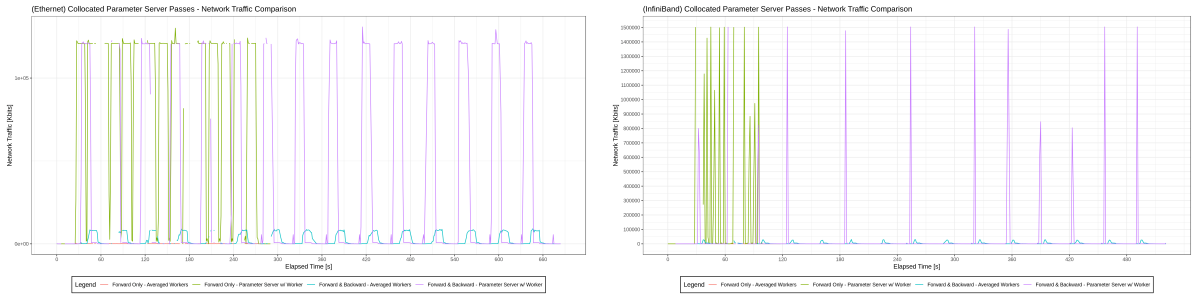
Distributed Replicated, seen in figure 5.3f, is shown to benefit from the added bandwidth, as both its parameter server process and worker processes show signs of greater network traffic than in the Ethernet case. This confirms the results seen in chapter 3, where the strategy benefited from the addition of InfiniBand interconnects, as it was unable to scale otherwise.

Although not immediately visible, both Distributed Allreduce strategies show greater bandwidth consumption when InfiniBand is added. This is more visible for PSCPU in figure 5.3g, than for Ringx1, in figure 5.3h. Nonetheless, the architecture sees slight improvements when InfiniBand is added. This conclusion is further reinforced by the results obtained for the other clusters, and by the architecture's behavior described in chapter 3.

It is thus possible to conclude that bar Caffe2 and Collective Allreduce, all architectures benefit from the addition of InfiniBand. This fact especially visible for the Parameter Server strategy, whose parameter server processes are able to send and receive the same amount of data in a shorter amount of time.

Forward Pass vs. Forward and Backward Passes

Figure 5.4 shows the Network Traffic of the Collocated Parameter Server using both pass combinations in both Ethernet and InfiniBand based mediums. The figure shows the differences between the two types of passes. In the forward pass case, it is immediately visible that the strategy's parameter server process is also dependent on the available bandwidth. This behavior is consistent with what was seen for the forward and backward pass. Moreover, this phenomenon occurs for both communication mediums.



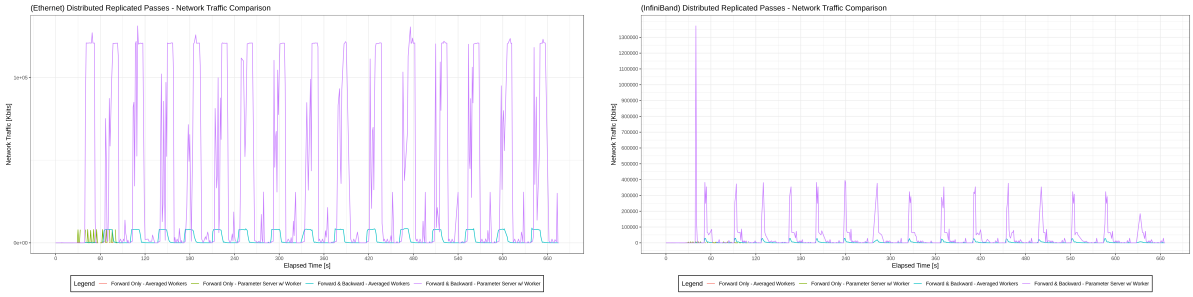
(a) Ethernet connections

(b) InfiniBand connections

Figure 5.4: The Network Traffic patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Collocated Parameter Server strategy. The x axis is the elapsed experiment time. The y axis represents the amount of data transmitted in a second. The two plots are not adjusted to represent the same spans on both axes. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one, bar the parameter server process which is plotted separately.

Figure 5.5 shows the same comparison for the Distributed Replicated strategy. The figure reveals that the architecture's forward pass depends much less on the available bandwidth than its forward and backward counterpart. This applies to both the Ethernet and InfiniBand

case. Moreover, the forward pass shows no signs of using more bandwidth when InfiniBand is added. This correlates with the results for the number of processed images per second for the forward pass, which were presented in sections 3.4.2 and 3.4.3. There, the forward pass showed the same performance across both mediums. Moreover, in the forward and backward pass combination, the parameter server process is highly dependent on bandwidth. This further correlates with what was seen in chapter 3, where the Distributed Replicated architecture excelled for the forward pass, but failed to reach the same performance when the backward pass was added.



(a) Ethernet connections

(b) InfiniBand connections

Figure 5.5: The Network Traffic patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Distributed Replicated strategy. The x axis is the elapsed experiment time. The y axis represents the amount of data transmitted in a second. The two plots are not adjusted to represent the same spans on both axes. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one, bar the parameter server process which is plotted separately.

In conclusion to this subset of experiments, for the Parameter Server, both pass combinations are highly dependent on the available bandwidth. This is specifically true for the parameter server processes themselves, as the workers do not require a considerable amount of bandwidth in either case. In contrast, the Distributed Replicated strategy shows very little communication during the forward pass. This applies to both process types, i.e. parameter server and worker. The Distributed Replicated strategy, however, makes liberal use of the bandwidth when the backward pass is employed. The results presented here correlate with what was seen in chapters 3 and 4. More specifically, in those cases where the two architectures are limited by the available bandwidth, performance stagnates and scalability is limited. This further supports the hypothesis that parameter server based architectures are highly dependent on the available bandwidth, and are unable to scale well in mediums where this resource is limited.

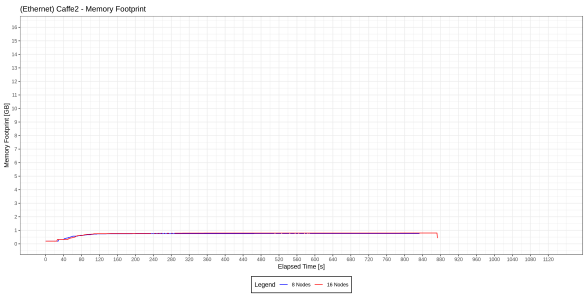
5.4.3. Memory Footprint Results

This section presents the results of the Memory Footprint evaluations for the tested parameter update strategies. For the sake of brevity, only a part of the results are presented: a comparative look at the footprints for two differently sized clusters, a survey of the footprints across equally sized clusters in different communication mediums, and an examination of the footprints across different pass combinations, for the Distributed Replicated and Collective Allreduce strategies.

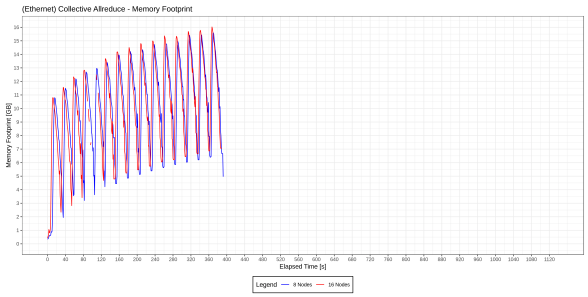
8 Nodes vs. 16 Nodes

Figure 5.6 provides a comparative look at the memory usage patterns of the different parameter update strategies. The tests focus on two clusters of different sizes: 8 and 16 nodes respectively. Both clusters make use of the Ethernet interconnects.

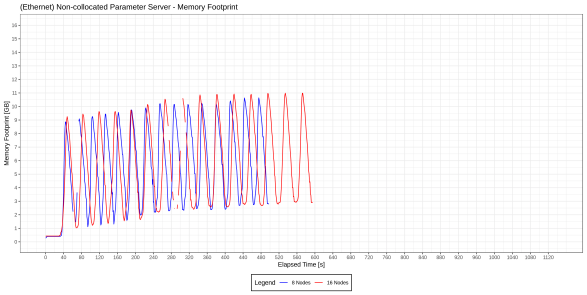
It is interesting to point out that Caffe2 uses very little memory for the ResNet50 model, as seen in figure 5.6a. Moreover, the memory usage remains the same regardless of the cluster size for each batch size. This pattern is constant across all the other tested clusters. With less than 1GB of used RAM, Caffe2 makes the most efficient use of memory out of all the tested strategies.



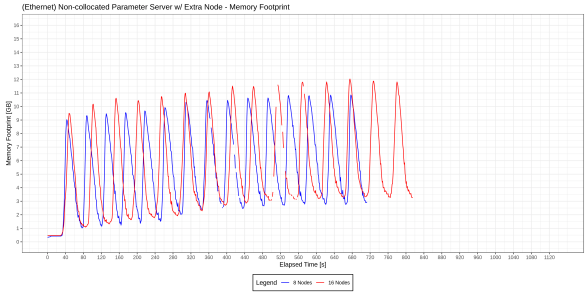
(a) Caffe2



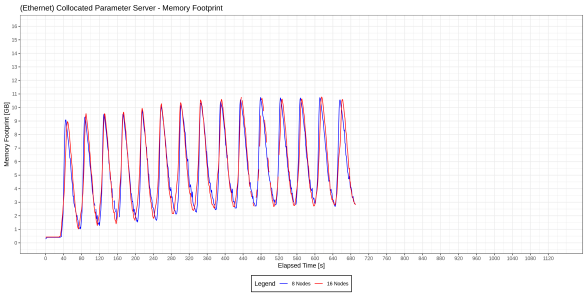
(b) Collective Allreduce



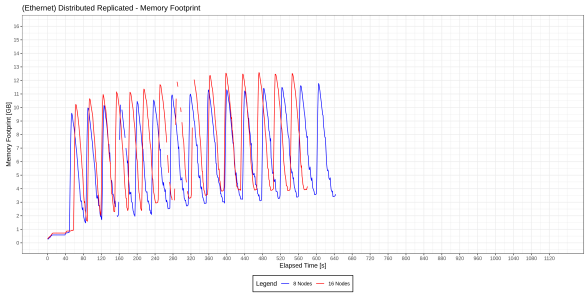
(c) Non-collocated Parameter Server



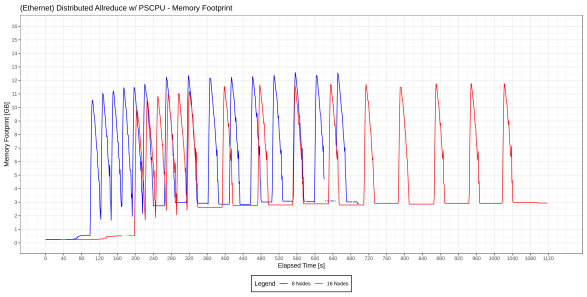
(d) Non-collocated Parameter Server with extra worker



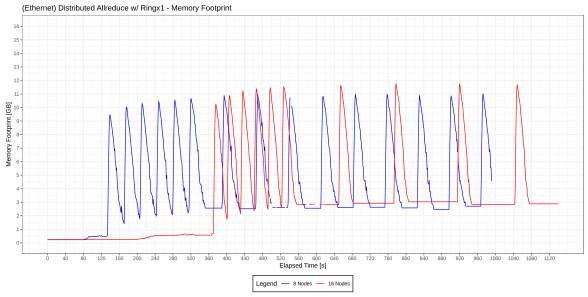
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure 5.6: The Memory Footprint of the parameter update strategies for a cluster of 8 nodes and a cluster of 16 nodes, both using Ethernet interconnects. The x axis represents the elapsed time in seconds. The y axis represents the Memory Footprint in GBs. Both the x axes and the y axes span the same range in each plot.

Collective Allreduce, shows a similar pattern: its memory footprint remains virtually the same regardless of cluster size. This is visible in figure 5.6b. However, the architecture has the greatest memory consumption out of all the tested alternatives. Furthermore, it shows quite an aggressive growth for each of the batches, of around 750 MB.

The Parameter Server strategies show different behaviors relative to which version is being used. The addition of more nodes does not produce a significant increase in the amount of memory used, as all versions use around 10 GB of RAM. This is consistent with the 12 and 4 node clusters, although they are not presented here.

Distributed Replicated shows slight increases in memory consumption for each batch. The increase seems to stabilize in the later iterations. This can be seen in figure 5.6f. Doubling the size of the cluster increases the memory consumption, however, by a relatively small margin of around 1 GB.

The Distributed Replicated strategies both show similar memory patterns. The addition of more nodes to the cluster does not produce any significant change in memory consumption, as it stays relatively constant at 11 GB.

As a conclusion to this subset of experiments, the addition of more nodes does not appear to produce a significant change in memory consumption, regardless of the parameter update architecture.

Ethernet vs. InfiniBand

Figure 5.7 provides a comparative look at the memory usage patterns of the different parameter update strategies. The tests focus on two clusters of 8 nodes using two types of interconnects: Ethernet and InfiniBand.

Figure 5.7a shows that Caffe2's memory consumption remains the same across InfiniBand and Ethernet connections. This fact is supported by all tested Caffe2 clusters using InfiniBand. Once more, its memory consumption is the lowest among those tested, with less than 1 GB of used RAM.

Collective Allreduce shows no change in memory footprint when InfiniBand is added, following the same pattern as Ethernet. Moreover, it continues to be the strategy with the highest RAM demands, having a memory consumption that appears to be a function of the elapsed time. This pattern is consistent in all the tested Collective Allreduce clusters.

In the case of the Parameter Server strategies, InfiniBand does not appear to have a significant effect on the amount of used memory. While figures 5.7c and 5.7e do show an increase in the used RAM, this is arguably not sufficient proof to conclude that InfiniBand leads to increased memory consumption. Moreover, this is further reinforced by figure 5.7d and the experiments executed for clusters of size 4, 12 and 16 over both communication mediums, where there is no noticeable different in RAM consumption between Ethernet and InfiniBand based mediums.

Distributed Replicated shows a slight increase in RAM usage in figure 5.7f. This behavior, however, is not exhibited in the results for the other tested clusters. Hence, InfiniBand is unlikely to exert any change in the amount of consumed memory.

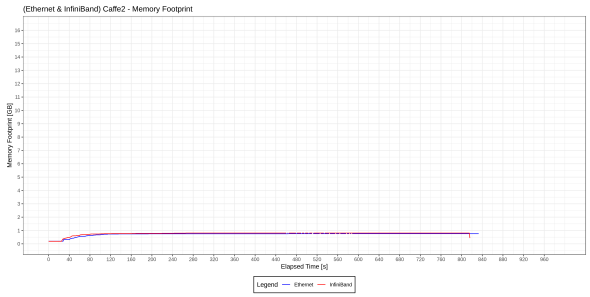
The Distributed Allreduce strategies are no exception from the other architectures when it comes to the lack of impact InfiniBand has on the memory footprint. Figures 5.7g and 5.7h both show no significant change in the amount of memory employed by this strategy, a point which is reinforced by the homologous experiments performed on the other cluster sizes.

As a conclusion to this subsection, InfiniBand does not appear to produce any significant effect on the memory footprint patterns of the tested parameter update strategies.

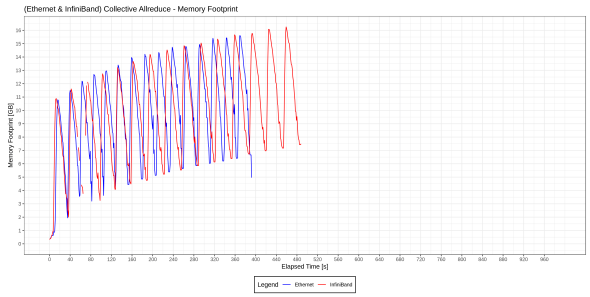
Forward Pass vs. Forward and Backward Pass

Figure 5.8 shows the Collocated Parameter Server's Memory Footprint for both pass combinations using both Ethernet and InfiniBand. From the figure, it is immediately obvious that the forward pass uses up a small fraction of the available memory in comparison to its backward counterpart. This applies for both the InfiniBand and Ethernet cases.

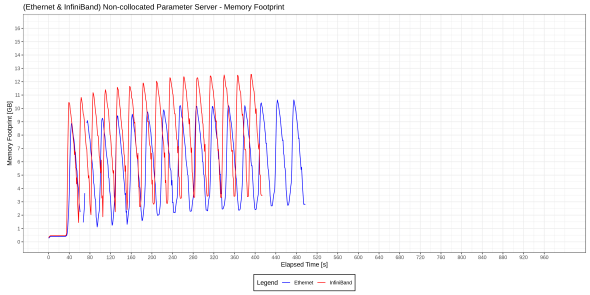
Figure 5.9 shows the Memory Footprint of the Distributed Replicated strategy for both pass combinations, given both Ethernet and InfiniBand connections. As in the Parameter Server case, the memory footprint for the forward pass is considerably lower than the backward pass, a phenomenon which is visible in both communication mediums.



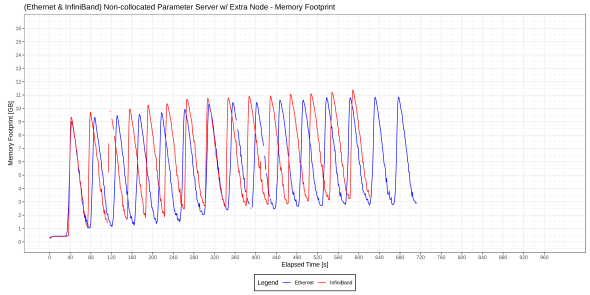
(a) Caffe2



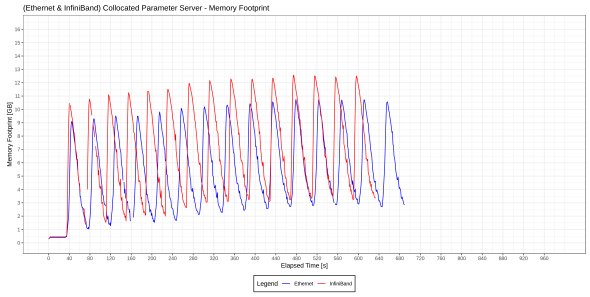
(b) Collective Allreduce



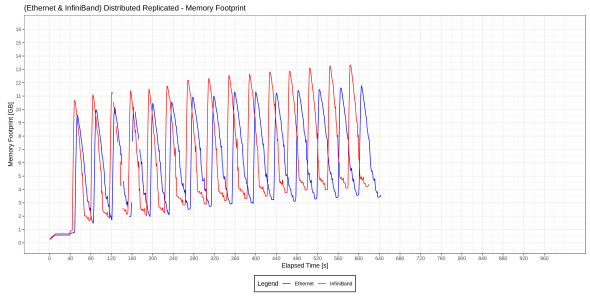
(c) Non-collocated Parameter Server



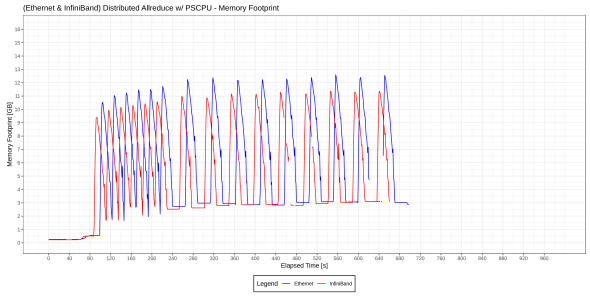
(d) Non-collocated Parameter Server with extra worker



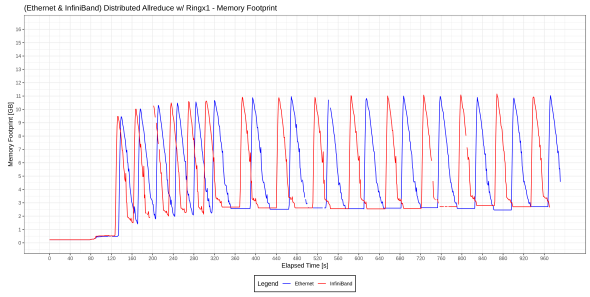
(e) Collocated Parameter Server



(f) Distributed Replicated

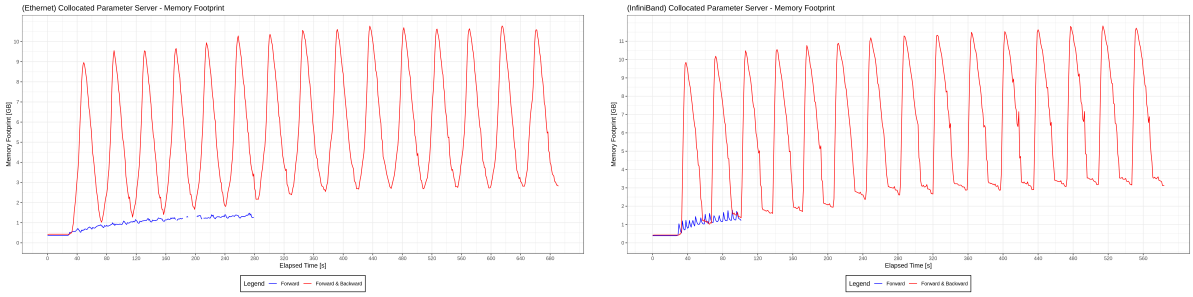


(g) Distributed Allreduce with PSCPU reduction strategy



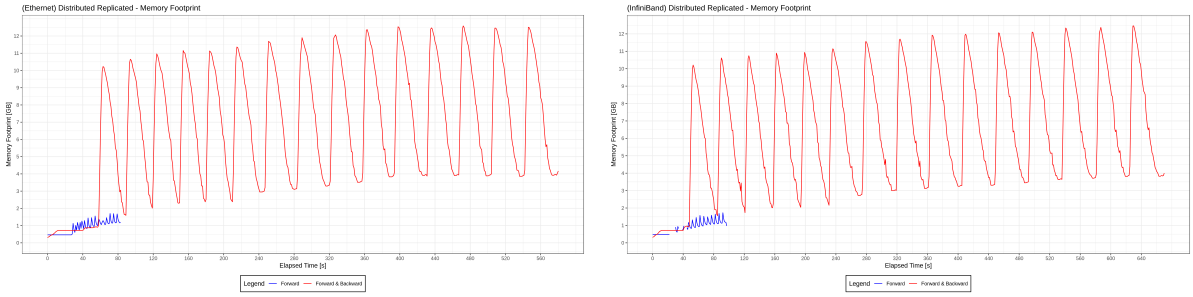
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure 5.7: The Memory Footprint of the parameter update strategies for two clusters of 8 nodes, one using Ethernet and the other InfiniBand. The x axis represents the elapsed time in seconds. The y axis represents the Memory Footprint in GBs. Both the x axes and the y axes span the same range in each plot.



(a) Ethernet connections (b) InfiniBand connections

Figure 5.8: The Memory Footprint patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Collocated Parameter Server strategy. The x axis is the elapsed experiment time. The y axis represents the amount of used memory in GBs. The two plots are not adjusted to represent the same spans on both axes. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one.



(a) Ethernet connections (b) InfiniBand connections

Figure 5.9: The Memory Footprint patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Distributed Replicated strategy. The x axis is the elapsed experiment time. The y axis represents the amount of used memory in GBs. The two plots are not adjusted to represent the same spans on both axes. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one.

Consequently, it is straight-forward to conclude that the backward pass incurs the bulk of the memory utilization, whereas the forward pass makes very limited use of it. This phenomenon occurs for both Ethernet and InfiniBand.

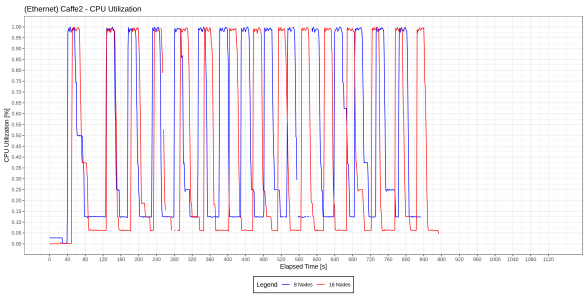
5.4.4. CPU Usage Results

This section presents the results of the CPU utilization evaluations for the tested parameter update strategies. For the sake of brevity, only a part of the results are presented: a comparative look at the footprints for two different sized clusters, a survey on the CPU consumption patterns across equally sized clusters in different communication mediums, and an examination of the CPU consumption patterns of both pass combinations for the Parameter Server and Distributed Replicated strategies.

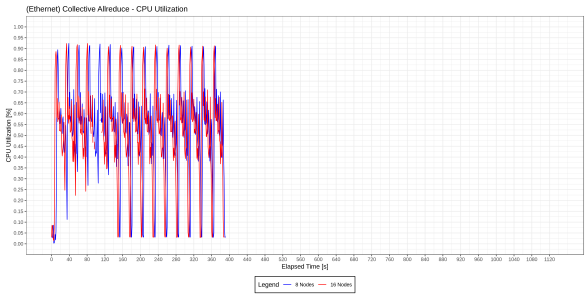
8 Nodes vs. 16 Nodes

Figure 5.10 provides a comparative look at the CPU usage patterns of the different parameter update strategies. The tests focus on two clusters of different sizes: 8 and 16 nodes respectively. Both clusters make use of the Ethernet interconnects.

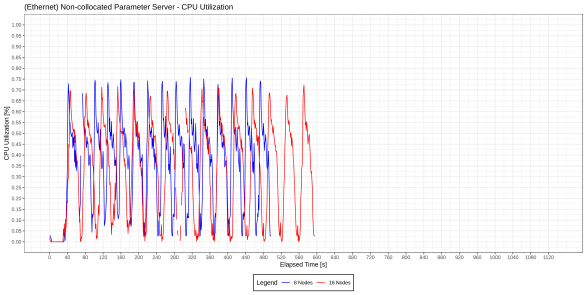
Caffe2 is visibly highly dependent on computational resources, as is shown in 5.10a. During each batch it uses up all the available CPU, a phenomenon which is consistent across all the tested Caffe2 clusters. This would suggest that computational resources are one of the sources of bottleneck in the framework when distribution is employed. Moreover, repeatedly overusing the CPU cores can lead to throttling, which can decrease performance for long running training processes. The cluster based CPU usage contrasts significantly with the single node case, where the usage of CPU resources is marginal, as seen in figure 5.11. This behavior seems to correlate with the performance of the framework, and implicitly its



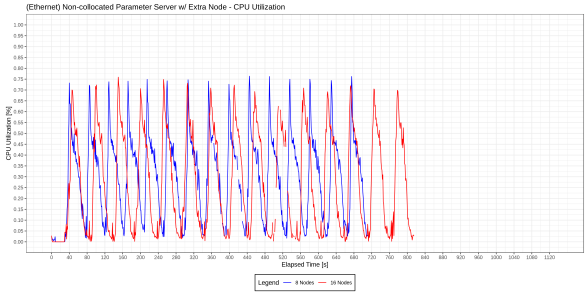
(a) Caffe2



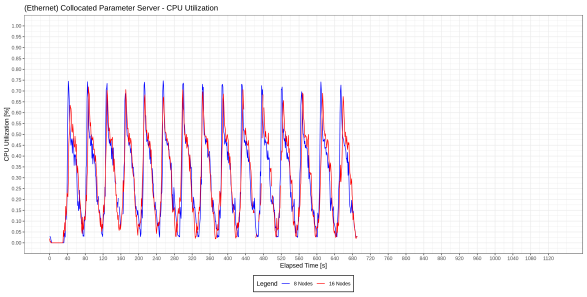
(b) Collective Allreduce



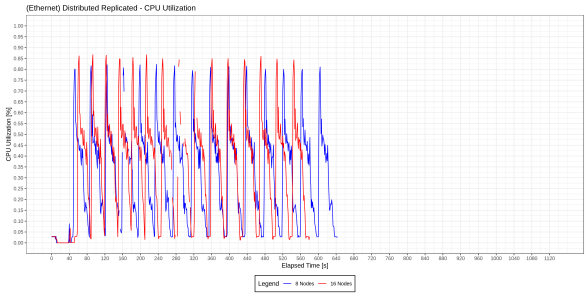
(c) Non-collocated Parameter Server



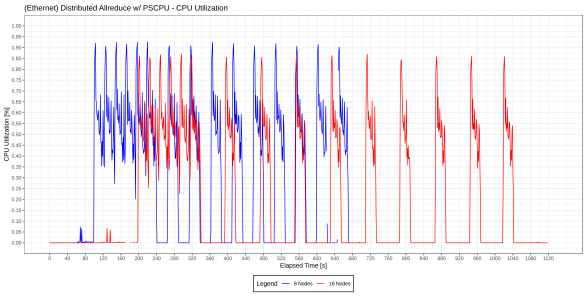
(d) Non-collocated Parameter Server with extra worker



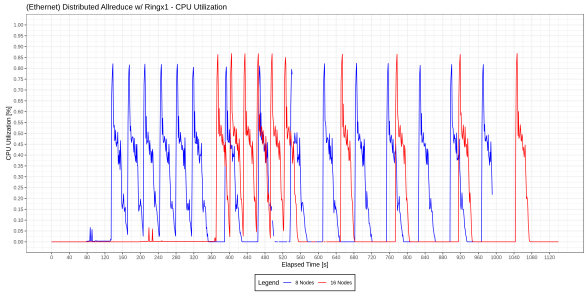
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure 5.10: The CPU utilization of the parameter update strategies for a cluster of 8 nodes and a cluster of 16 nodes, both using Ethernet interconnects. The x axis represents the elapsed time in seconds. The y axis represents the normalized CPU utilization. Both the x axes and the y axes span the same range in each plot.

distributed architecture: when the CPU consumption is low, i.e. in the single node case, the framework performs well, when the CPU consumption is very high, i.e. in the distributed case, the framework does not perform well, nor it does not scale well. This continues to support the hypothesis forwarded in chapter 3, that CPU consumption is a likely factor in the limited scalability of Caffe2. Considering that other similar Ring Allreduce strategies see no CPU bottleneck, it is possible to speculate that the underlying implementation of the Ring Allreduce algorithm in Caffe2 is unsuitable. It should be noted, however, that the CPU usage of Caffe2 is not influenced by the size of the cluster.

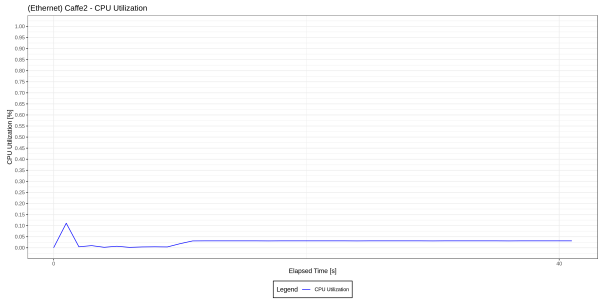


Figure 5.11: Caffe2 CPU utilization pattern on a single node execution context.

All of TensorFlow's architectures show similar CPU utilization patterns: a sharp peak, followed by a decrease in usage. There is one peak for each processed batch. None of the framework's architectures show signs of overusing the CPU for long periods of time, or indeed of ever completely using the available computational resources. This contrasts with Caffe2. This CPU utilization pattern is maintained across all the tested cluster configurations, which leads to the conclusion that the size of the cluster does not have any significant effect on the CPU utilization.

As a conclusion to this subset of experiments, the size of the underlying cluster does not influence the CPU utilization habits of the parameter update strategies.

Ethernet vs. InfiniBand

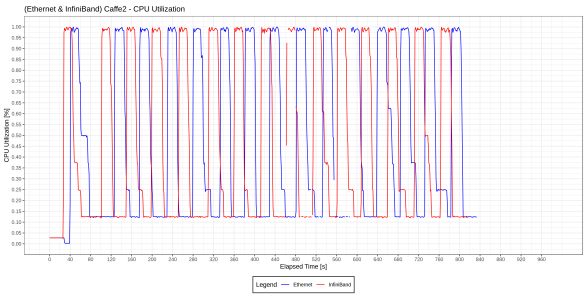
Figure 5.12 provides a comparative look at the CPU usage patterns of the different parameter update strategies. The tests focus on two clusters, both having 8 nodes, however, using a different type of interconnections: Ethernet and InfiniBand.

For InfiniBand interconnections, Caffe2 does not appear to change its CPU consumption pattern with the addition of InfiniBand interconnects. This however, is intuitive, since its network traffic does not change with the addition of InfiniBand, a fact which has been observed both in this chapter and chapter 3. It should be mentioned, however, that if InfiniBand had a detrimental effect on the CPU usage, this would be hard to see in figure 5.12a, since CPU utilization is at maximal capacity in both time series. Nonetheless, this is unlikely to be the case, since as the plateaus of peak CPU usage are of equal duration in both cases. This would suggest that in either case, the amount of computation to be done is independent of the underlying bandwidth, a fact which is further supported by Caffe2's Network Traffic patterns.

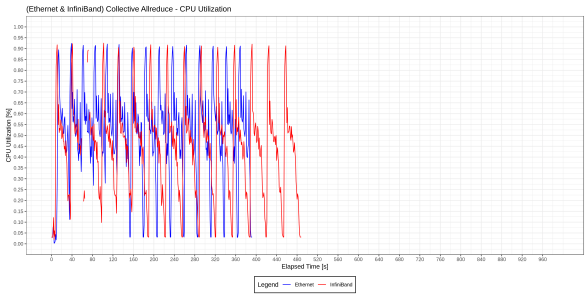
While in figure 5.12f Distributed Replicated appears to be influenced by InfiniBand, this is unlikely to be the case, as comparisons for clusters at sizes 4, 12 and 16 reveal that there is little to no difference between Ethernet and InfiniBand in terms of CPU utilization.

Both Collective Allreduce and Distributed Allreduce do not show any signs of being affected by the change to InfiniBand, as CPU utilization remains the same.

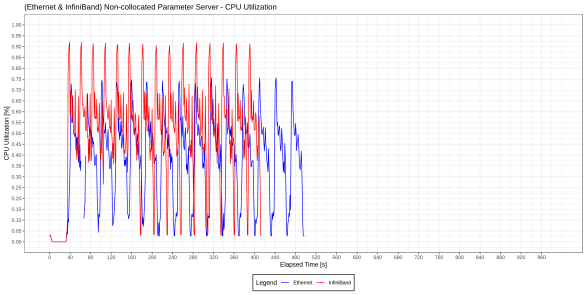
The Parameter Server strategies appear to be influenced by the addition of InfiniBand connections, as clusters with Ethernet interconnections show lower CPU utilization than their InfiniBand counterparts. This is consistent to the evidence produced by the clusters of size 4, 12 and 16. The network patterns of the Parameter Server architecture reinforce this hypothesis, as the strategy has shown to be highly dependent on bandwidth. A possible explanation of why bandwidth affects CPU consumption in the aforementioned architecture



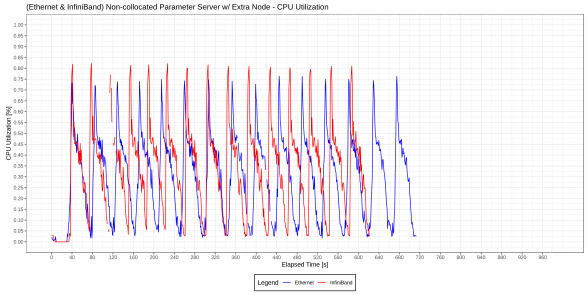
(a) Caffe2



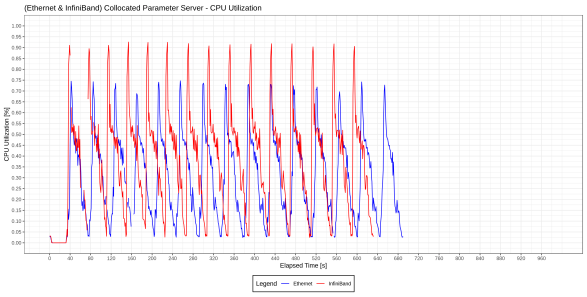
(b) Collective Allreduce



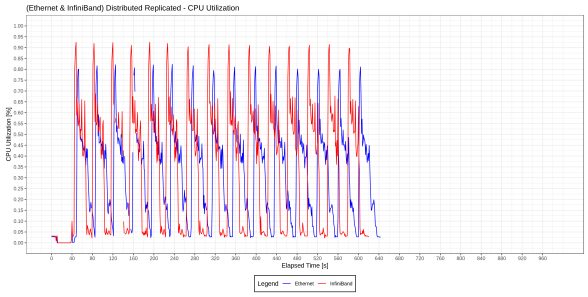
(c) Non-collocated Parameter Server



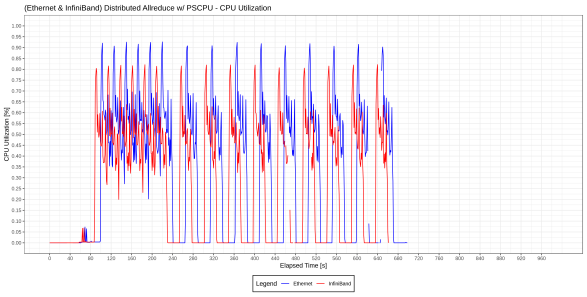
(d) Non-collocated Parameter Server with extra worker



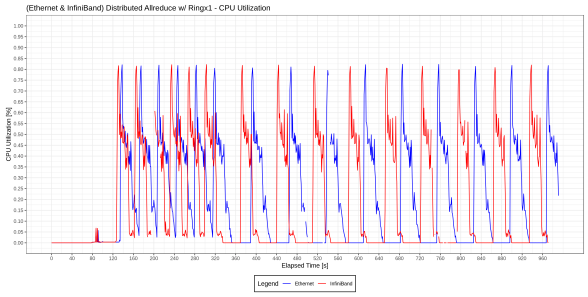
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

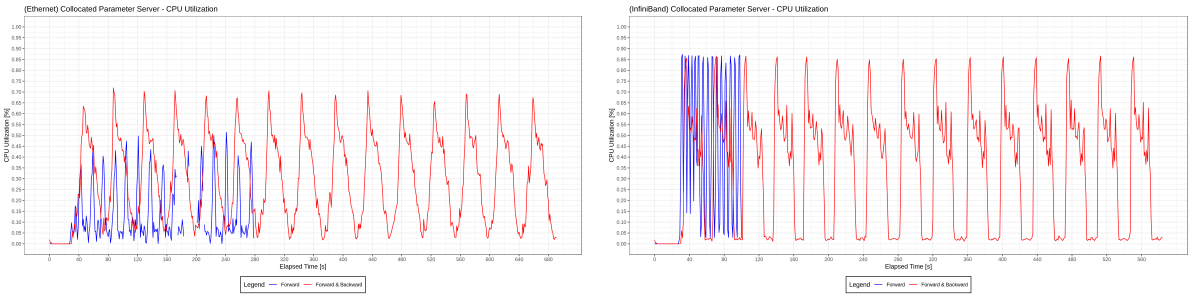
Figure 5.12: The CPU utilization of the parameter update strategies for two clusters of 8 nodes using Ethernet and InfiniBand interconnects respectively. The x axis represents the elapsed time in seconds. The y axis represents the normalized CPU utilization. Both the x axes and the y axes span the same range in each plot.

refers to the fact that the greater bandwidth allows the parameter server process to deliver more data per unit of time to the workers, which in turn means that more data can be processed at that point.

In conclusion to this subset of experiments, all strategies, bar Parameter Server, show no significant signs of being influenced by the addition of InfiniBand. Parameter Server, on the other hand, shows a greater CPU consumption as InfiniBand is added. This further reinforces the idea that Parameter Server is highly reliant on the underlying bandwidth. As a further point, it should be noted that other than bandwidth, Parameter Server does not exhibit any other bottlenecks across the evaluated dimensions.

Forward Pass vs. Forward and Backward Pass

Figure 5.13 reveals the CPU consumption habits of the Parameter Server for its two different pass combinations. While for Ethernet connections, as seen in figure 5.13a, the forward pass uses less CPU resources, in the InfiniBand case, it uses as many as the forward and backward pass. This is consistent to what was seen earlier in this section regarding the architecture's increased CPU consumption with the addition of more bandwidth, a phenomenon likely correlated with the high bandwidth demands of the parameter server process. For the forward pass, the CPU utilization pattern is significantly different than the forward and backward pass. More specifically, in the forward pass, the CPU usage shows sudden spikes, followed by abrupt drops. When the backward pass is added, the drops become much more gradual, which is likely due to the backward propagation of the gradients through the network. In neither case, however, is the CPU used at high capacity for extended periods of time, nor is it ever used at 100% capacity.



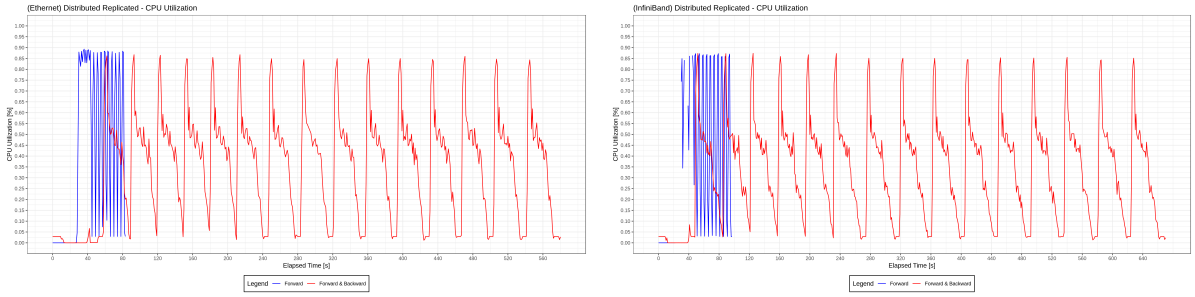
(a) Ethernet connections

(b) InfiniBand connections

Figure 5.13: The CPU Utilization patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Collocated Parameter Server strategy. The x axis is the elapsed experiment time. The y axis represents the normalized consumption of CPU resources. The two plots are not adjusted to represent the same spans on the x axis. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one.

Figure 5.14 shows the CPU utilization patterns of the Distributed Replicated strategy for both pass combinations. Unlike the Parameter Server strategy, the CPU consumption peaks for the forward pass are the same as when the backward pass is added. This phenomenon continues to be true in the InfiniBand case as well. As in the Parameter Server case, in the forward pass case, the CPU consumption peaks are sudden and short lived, as they are followed by sudden drops. When the backward pass is added, the drops are much more gradual. This behavior is due to the backpropagation which occurs during the backward pass.

In conclusion to this subsection, for Parameter Server, CPU consumption is lower for the forward pass relative to its counterpart when Ethernet is used. When InfiniBand is added, both pass combinations show the same peak CPU usage. This continues to support the idea that bandwidth is important to the Parameter Server architecture. For Distributed Replicated, the peak CPU consumption is the same for both pass combinations in both types of connections. Moreover, the patterns show that the backward pass employs the bulk of CPU utilization, as the drops in CPU usage are much more gradual than the cases where only the forward pass is employed, a phenomenon which is due to backpropagation.



(a) Ethernet connections

(b) InfiniBand connections

Figure 5.14: The CPU Utilization patterns of the forward and forward and backward pass combinations, in both Ethernet and InfiniBand based mediums for the Distributed Replicated strategy. The x axis is the elapsed experiment time. The y axis represents the normalized consumption of CPU resources. The two plots are not adjusted to represent the same spans on the x axis. The results are obtained on a cluster of 16 nodes, whose individual time series are averaged into one.

5.5. Experiment Conclusions

RQ3: *What are the CPU usage, Memory footprint, and Network usage patterns of the distributed parameter update strategies?*

- Architectures which fit in the parameter server arch-type (Parameter Server, Distributed Replicated, PSCPU Distributed Allreduce) benefit from the addition of InfiniBand.
- Ring based methods (such as Caffe2 and Collective Allreduce), have low bandwidth demands, as they employ few point-to-point transmissions between pairs of workers. Consequently, such architectures do not see any changes in the network patterns given InfiniBand.
- Ring based architectures (such as Caffe2, Collective Allreduce and Ringx1 Distributed Allreduce) do not see any changes in Network Traffic as the size of the cluster changes.
- Strategies which fit in the parameter server arch-type (Parameter Server, Distributed Replicated and PSPCU Distributed Allreduce) are dependent on the number of worker processes.
- Cluster size and InfiniBand connections have no effect on the Memory Footprint of the tested architectures.
- Evidence shows that Caffe2 is bottlenecked by the CPU which is a likely explanation of its inability to scale. Considering that similar ring based architectures scale much better, it is possible to infer that the particular implementation of Caffe2's ring architecture is inadequate.
- CPU utilization is independent of cluster size in all of the tested architectures. Furthermore, bar the Parameter Server strategy, all the other architecture's CPU consumption does not change with the addition of InfiniBand. The CPU usage of the Parameter Server architecture increases as InfiniBand is added.

6

Conclusions and Further Work

Machine Learning has become an exciting field in Computer Science, which has garnered a large following from both Academia and Industry. Deep Learning has become especially popular, having received a large amount of interest due its numerous applications, that have redefined what the state of the art is in a large array of domains such as Computer Vision, Natural Language Processing or Bioinformatics. It is only natural that a large number of frameworks specifically designed to ease the production and development of Machine Learning applications have been made publicly available to the wide Computer Science community. As a large volume of data is usually an essential component of Machine Learning and of Deep Learning especially, distribution is generally essential in order to make training large models targeted at solving complex problems practical. Multi-node distribution is especially applied to very large scale tasks, which cannot be trivially handled by a single node with intra-node distribution. To this extent, many of the aforementioned frameworks support large scale distribution by employing various distributed parameter update strategies in order to successfully accomplish the learning task in such scenarios. Popular examples include: Parameter Server, Ring Allreduce, Collective Allreduce, Distributed Allreduce, and Distributed Replicated. At the time of writing, however, there is a lack of open and objective research into the efficiency, or on the contrary, into the inefficiencies of parameter update strategies. This thesis attempts to fill in this gap, by looking into the aforementioned parameter update strategies, and answering 3 research questions. Section 6.1 briefly summarizes the research questions, and the conclusions reached for each of them, while section 6.2 presents potential further research paths, which stem from this study. It should be mentioned, however, that the key findings in each experiment chapter reveal interesting discussions and insights into the aforementioned parameter server strategies, which, due to brevity reasons, are not discussed here.

6.1. Conclusions

The following is a brief summary of the research questions and the conclusions reached for each of them:

RQ1 *Which of the distributed parameter update strategies scale well as more nodes are added?*

Evaluations were performed across a large array of cluster sizes, ranging from 1 to 56 nodes, using both commodity and high performance connections (e.g. 1 Gbps Ethernet and FDR InfiniBand), and employing either the forward pass or the forward and backward passes typical to Neural Networks. Their *batch time* and *processed images per second* are measured. Collective Allreduce has been shown to scale linearly across both mediums, yielding the best performance out of all the tested strategies. Parameter Server cannot scale in commodity Ethernet based mediums exhibiting logarithmic growth. It shows top performance when high bandwidth connections (such as InfiniBand) are available via a linear growth comparable to Collective Allreduce. Caffe2

performs better than any architecture on a single node, however it's underlying Ring Allreduce strategy is unable to scale well when distribution is employed, in either of the communication mediums. Distributed Replicated fares well when only the forward pass is employed, however, it performs significantly worse when the backward pass is added, being unable to scale in the commodity network case, and showing limited scalability in the high performance interconnects context. Distributed Allreduce has shown that it is unable to scale in any of the tested contexts, in certain cases showing decreased performance as more nodes are added.

RQ2 *How well does the Parameter Server strategy scale relative to the batch size and the number of nodes?*

Parameter Server was tested across a large range of cluster sizes, ranging from 1 to 40 nodes, using both commodity and high performance connections (e.g. 1 Gbps Ethernet and FDR InfiniBand). Moreover, the architecture was tested for both its forward pass and its forward and backward pass combination. Their *batch time* and *processed images per second* are measured. Results reveal that the time per batch scales linearly in the batch size in both the communication mediums, where the growth is inversely proportional to the size of the cluster. Parameter Server is unable to scale in commodity Ethernet based mediums, as the limited bandwidth, and communication overhead disallows large clusters to reap in the benefits of additional resources. High-performance connections such as InfiniBand yield great benefits, and allow Parameter Server to scale linearly relative to the number of nodes in the clusters. Parameter Server, however, shows a logarithmic growth in the batch size, for both communication mediums, for both pass combinations. This is especially visible when high bandwidth connections are employed, as the clusters derive marginal performance benefits past the 64 images per batch size. This challenges the common practice which is often employed in large scale machine learning, where the size of the batch is set to large values in order to decrease communication frequency. Moreover, when paired with research which shows that large batch sizes are detrimental to the quality of the trained model, this result proposes an interesting question: *are large batch sizes actually useful for large scale Machine Learning in the context of the Parameter Server architecture?*.

RQ3 *What are the CPU usage, Memory footprint, and Network usage patterns of the distributed parameter update strategies?*

The distributed parameter update strategies are tested across clusters of size 4, 8, 12 and 16, for both commodity and high performance mediums. Their *CPU utilization*, *Memory Footprint* and *Network Traffic* are measured. Results reveal that Caffe2 has a CPU bottleneck, which is likely one of the reasons why it unable to scale when the workload is CPU bound. Its CPU usage for the single node case is negligible, which suggests that the architecture's performance is directly correlated to its CPU consumption. Additionally, Caffe2 has a very low memory consumption, which remains constant across all the tested contexts. All of TensorFlow's architectures make efficient use of the available CPU and Memory, and are generally not affected by the addition of InfiniBand (with the exception of Parameter Server) or the cluster size. Parameter Server is highly dependent on the underlying bandwidth due to the frequent point-to-point exchanges between parameter server and workers, which explains the architecture's inability to scale in commodity Ethernet based mediums. This is an issue faced by the other parameter server based architectures like Distributed Replicated, and PSCPU Distributed Allreduce. Analyses on the individual training passes reinforce the hypothesis that bandwidth limited parameter server processes are a core cause behind scalability issues. Ring based methods, which generally employ unicast communication, are less affected by the available bandwidth, and generally do not benefit to a large extent from the addition of larger bandwidth connections, nor are they affected by the size of the cluster. Examples include Caffe2 and Collective Allreduce.

6.2. Future Work

This study has revealed many further research paths, which can either build on top of the results obtained in this thesis, address the identified limitations of some of the parameter update strategies, or look deeper into certain interesting issues surrounding large scale distributed Machine Learning. Although not exhaustive by any means, the following list presents a few ideas for further research:

1. The results concerned with Parameter Server's scalability relative to the batch size presented in this study (see chapter 4), paired up with the research in [37], which suggests that large batch sizes are detrimental to the quality of a trained Neural Network, put forward an interesting question: *do large batch sizes improve the training time in large scale Parameter Server based Machine Learning, or on the contrary are they detrimental from both a model quality and training time perspective?*. Further research is required to better understand how the batch size affects the Parameter Server architecture, and if indeed batch sizes past a certain point yield no noticeable benefit to the training time.

2. Although not concrete, certain evidence in chapter 3 has shown that the Soft Placement heuristics can have a lower performance than their Hard Placement counterparts. Further research is needed to better understand if this is indeed the case, and if so, when does this happen, and how can it be fixed, or at least improved?

3. The evaluations performed in this study have been done exclusively using synthetic, randomly generated data. It would be highly interesting to see if real world data, in the form of canonical datasets, such as ImageNet, CIFAR, MS-COCO, and others, imply any changes to the results obtained here. This could provide valuable insight into how the nature of the data influences the performance of a parameter update strategy, whether this refers to scalability or hardware consumption. If indeed there is a discrepancy, it might then be possible to understand how the parameter update strategies can adapt to the nature of the data, or, on the contrary, how the data itself can be modified, in order to ensure that parameter update strategies yield peak performance.

4. The set of parameter update strategies evaluated here is not exhaustive. There are other update strategies which have not been evaluated in this study. It is likely worth looking into these other strategies, in order to build a more complete picture of the performance hierarchy in large scale distributed Machine Learning.

5. The study in this paper was conducted against the ResNet50 Neural Network. There is an abundance of other Neural Networks published in literature, for which a similar study is worth performing. This would diversify the results obtained here, and would either reinforce the conclusions reached in this paper or challenge them. Regardless, such a study would be very useful, and would further build on the current understanding of large scale distributed Machine Learning.

6. Caffe2 has shown to be very efficient for the single node case, however, when distribution is used, its performance is lacking. Moreover, in the single node case, the framework uses little computation power, unlike the distributed case, where it is bottlenecked by CPU resources, which would indicate that scalability and CPU consumption are correlated. It would be interesting to further investigate this issue, and forward a concrete hypothesis of why Caffe2 is unable to scale particularly well.

7. The parameter server arch-type is highly dependent on the available bandwidth, as it relies on frequent point-to-point communication between the parameter server and the individual worker processes. This phenomenon has been seen in many instances of the arch-type, such as Parameter Server or Distributed Replicated. Looking into how the arch-type could be improved in this sense, i.e. reducing the cost of communication, may yield great benefits to its performance. An ambitious goal would be to make the all parameter server based architectures viable for commodity Ethernet mediums.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Martin 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. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

[3] ASCI. Das-5 documentation, 2019. URL `https://www.cs.vu.nl/das5/`.

[4] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.

[5] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[6] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[8] Google Brain. Tensorflow documentation, 2019. URL `https://www.tensorflow.org/api_docs`.

[9] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016. URL `http://arxiv.org/abs/1604.00981`.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[12] Daniel Crevier. *AI: The Tumultuous History of the Search for Artificial Intelligence*. Basic Books, Inc., New York, NY, USA, 1993. ISBN 0-465-02997-3.

[13] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 37–48, 2014.

[14] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

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

[18] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

[19] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL http://arxiv.org/abs/1706.02677.

[20] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[21] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014. URL http://arxiv.org/abs/1412.5567.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. URL http://arxiv.org/abs/1603.05027.

[24] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning Workshop*, 03 2015.

[25] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[26] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, April 1998. ISSN 0218-4885. doi: 10.1142/S0218488598000094. URL http://dx.doi.org/10.1142/S0218488598000094.

[27] Facebook Incubator. Gloo repository, 2019. URL https://github.com/facebookincubator/gloo.

[28] Genlin Ji and Xiaohan Ling. Ensemble learning based distributed clustering. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 312–321. Springer, 2007.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[30] Jonathan Katzy Jeroen Klop-Penburg Tim Verbelen Joost Verbraeken, Matthijs Wolting and Jan S. Rellermeyer. A survey on distributed machine learning. 2019.

[31] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. URL http://arxiv.org/abs/cs.AI/9605103.

[32] Tim Kraska, Ameet Talwalkar, and John Duchi. Mlbase: A distributed machine-learning system. In *In CIDR*, 2013.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

[34] Stanford University. Stanford Electronics Laboratories, B. Widrow, United States. Office of Naval Research, United States. Army Signal Corps, United States. Air Force, and United States. Navy. *Adaptive "adaline" neuron using chemical "memistors."*. 1960. URL https://books.google.ch/books?id=Yc4EAAAAIAAJ.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436, 2015.

[36] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[37] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018. URL http://arxiv.org/abs/1804.07612.

[38] John McCarthy. Professor sir james lighthill, frs. artificial intelligence: A general survey. *Artif. Intell.*, 5(3):317–322, 1974. URL http://dblp.uni-trier.de/db/journals/ai/ai5.html#McCarthy74.

[39] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL http://jmlr.org/papers/v17/15-237.html.

[40] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL http://dl.acm.org/citation.cfm?id=3104322.3104425.

[41] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[42] Rolf Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, pages 1–9. Springer, 2004.

[43] Berkeley AI Research. Caffe documentation, 2019. URL https://caffe.berkeleyvision.org/.

[44] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

[45] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6. URL http://dl.acm.org/citation.cfm?id=65669.104451.

[46] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

[47] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. pages 2135–2135, 08 2016. doi: 10.1145/2939672.2945397.

[48] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[50] Facebook Open Source. Caffe2 documentation, 2019. URL https://caffe2.ai/docs.

[51] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Dipankar Das, Mikhail E. Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, and Pradeep Dubey. On scale-out deep learning training for cloud and HPC. *CoRR*, abs/1801.08030, 2018. URL http://arxiv.org/abs/1801.08030.

[52] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[53] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, Inc., Orlando, FL, USA, 4th edition, 2008. ISBN 1597492728, 9781597492720.

[54] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *Arxiv*, 2016. URL https://arxiv.org/abs/1609.03499.

[55] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 84–97. ACM, 2016.

[56] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1335–1344, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2783323. URL http://doi.acm.org/10.1145/2783258.2783323.

[57] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015. URL http://arxiv.org/abs/1502.03044.

[58] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[59] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855742.

[60] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[61] Kuo Zhang, Salem Alqahtani, and Murat Demirbas. A comparison of distributed machine learning platforms. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[62] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. *CoRR*, abs/1511.05950, 2015. URL http://arxiv.org/abs/1511.05950.

A

Results of the Parameter Update Strategies' Scalability Experiments

This annex presents the concrete results obtained during the experiments focusing on the parameter update architectures' ability to scale relative to the size of the cluster and in different communication mediums. The batch sizes are kept fixed for all experiments at 32 images. These experiments are discussed in detail in chapter 3. The exact values of the results are presented here as tables. If not specified otherwise, the results are obtained for the Hard Placement heuristics.

A.1. Ethernet Results

Node Count	Forwd. Seconds per Batch [sec / batch]	Forwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]
1	0.525107	60.9399608079877	1.97693	16.1867137430258
2	4.8403275	6.61112290439025	23.25	1.37634408602151
4	2.4817775	12.893984251207	12.02365	2.66142144856179
8	1.2248525	26.1255947144656	6.3431375	5.04482206163748
16	0.5716589375	55.9774332225848	3.32293125	9.63005178033701
24	0.426597916666667	75.0120869085351	2.19693333333333	14.565758329793

Table A.1: Caffe2 (Ring Allreduce)

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
2	26.615	1.20232951343228	6.97	4.59110473457676
4	55	0.581818181818182	10.5175	3.04254813406228
8	110.85625	0.288662118734848	20.53375	1.55840993486333
16	220.549375	0.145092227080671	54.25875	0.589766627502477
24	333.12875	0.096058956184358	62.555416667	0.511546428830375
32	442.8765625	0.072254896080666	83.1828125	0.384694855081992
40	545.26675	0.058686872067662	91.28275	0.350559114400037
48	654.425	0.048897887458456	109.82125	0.291382587613964

Table A.2: Collective Allreduce

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	14.6	2.19178082191781	3.62	8.83977900552486
2	20.555	1.55679883240088	6.455	4.95739736638265
4	25.872	1.23685837971552	11.23	2.84951024042743
8	30.645	1.04421602218959	16.92	1.89125295508274
16	33.4125	0.957725402169847	20.65	1.54963680387409
24	34.451666667	0.928837501805149	22.49	1.42285460204535
32	34.8703125	0.917686068916073	22.448125	1.42550881198318
40	35.291891892	0.906723847447062	23	1.39130434782609
48	35.4525	0.90261617657429	23.69	1.3507809202195
56	35.651785714	0.897570748817611	25.19	1.27034537514887

Table A.3: Collocated Parameter Server

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	5.13	6.23781676413255	4.06	7.88177339901478
2	8.98	3.56347438752784	7.49	4.27236315086782
4	14.43	2.21760221760222	12.2925	2.60321334146838
8	20.48875	1.56183271307425	17.652125	1.81281290496187
16	26.65	1.20075046904315	21.304	1.50206533984228
24	29.368	1.08962135657859	22.721	1.40838871528542
32	30.771666667	1.03991767317294	22.448709677	1.42547168458354
40	35.206111111	0.908933108206937	22.51	1.42159040426477
48	35.457021277	0.902501080110684	23.663023256	1.3523208617008

Table A.4: (Soft Placement) Collocated Parameter Server

Node Count	RINGx1 - Forwd. + Backwd. Images per Second [img / sec]	RINGx1 - Forwd. + Backwd. Seconds per Batch [sec / batch]	PSCPU - Forwd. + Backwd. Images per Second [img / sec]	PSCPU - Forwd. + Backwd. Seconds per Batch [sec / batch]
2	4.98	6.42570281124498	5.04	6.34920634920635
4	5.51	5.80762250453721	6.84	4.67836257309942
8	6.23	5.13643659711075	7.99	4.00500625782228
16	4.91	6.5173116089613	8.48	3.77358490566038
24	-	-	8.71	3.673938002

Table A.5: (Forward and Backward only) Distributed Allreduce

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	14.67	2.1813224267212	3.6	8.88888888888889
2	28.57	1.12005600280014	6.205	5.15713134568896
4	56.4725	0.566647483288326	12.03	2.66001662510391
8	111.41125	0.287224135803162	17.915	1.78621267094613
16	214.1825	0.149405296884666	19.040625	1.68061710159199
24	326.197916667	0.098099952099533	19.843333333	1.61263228626932
32	412.2925	0.077614800172208	20.389666667	1.56942241982754
40	515.67525	0.062054558561808	20.088717949	1.5929339085371
48	614.101041667	0.052108688682786	21.154042553	1.51271322820809
56	708.425714286	0.045170579433656	21.170357143	1.51154748046283

Table A.6: Distributed Replicated

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	5.13	6.23781676413255	4.06	7.88177339901478
2	4.35	7.35632183908046	3.75	8.53333333333333
4	11.03	2.9011786038078	9.39	3.40788072417465
8	18.294285714	1.74918007186864	16.13	1.98388096714197
16	24.858666667	1.28727740826422	20.979333333	1.52531062317718
24	28.204285714	1.13457934458932	22.763636364	1.40575079869959
32	29.863448276	1.07154403953133	21.611153846	1.4807168663011
40	30.99	1.03259115843821	22.18	1.44274120829576
48	31.77	1.00723953415172	22.86	1.39982502187227

Table A.7: (Soft Placement) Parameter Server

A.2. InfiniBand Results

Node Count	Forwd. Seconds per Batch [sec / batch]	Forwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]
1	0.525107	60.9399608079877	1.97693	16.1867137430258
2	5.1097125	6.26258326667107	23.0458	1.38853943339446
4	2.42856625	13.1764986685181	12.20608125	2.62164402682474
8	1.22079890625	26.2123432746973	6.2302625	5.13622018333898
16	0.618524375	51.7360370801878	3.2426125	9.86858590102888
24	0.402147083333333	79.572876010332	2.15779083333333	14.8299823623621

Table A.8: Caffe2 (Ring Allreduce)

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
2	27.355	1.16980442332298	5.175	6.18357487922705
4	54.4925	0.587236775703078	13.7675	2.32431450880697
8	110.5125	0.289560004524375	27.45875	1.16538444029681
16	221.693125	0.144343673264563	54.315	0.589155850133481
24	333.046666667	0.096082631062618	60.785	0.5264456691618
32	440.521875	0.072641114587102	82.09625	0.389786378793185
40	544.615	0.058757103642022	91.747	0.348785246384078
48	652.08	0.04907373328426	110.393541667	0.289872029801593

Table A.9: Collective Allreduce

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	14.6	2.19178082191781	3.62	8.83977900552486
2	26.395	1.2123508240197	6.945	4.60763138948884
4	52.16	0.613496932515337	13.8025	2.31842057598261
8	101.935	0.313925540785795	27.36	1.16959064327485
16	188.21625	0.170017200959003	42.414375	0.754461193875897
24	254.504166667	0.125734680178614	60.49625	0.5289584065128
32	299.2615625	0.106929870086473	79.4953125	0.402539457908289
40	352.144864865	0.090871692853643	95.71425	0.334328482958389
48	390.083958333	0.082033622035497	110.07875	0.290700975438039
56	427.231509434	0.074900842501982	128.347037037	0.249324026005953

Table A.10: Collocated Parameter Server

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	5.13	6.23781676413255	4.06	7.88177339901478
2	9.85	3.248730964467	7.44	4.3010752688172
4	19.345	1.65417420522099	14.5225	2.20347736271303
8	37.47	0.8540165465705 9	26.134285714	1.2244451732 9424
16	71.416875	0.448073372014667	47.2293333	0.677545028991548
24	100.52	0.318344608038201	63.30086	0.505523578321 05
32	127.6275	0.250729662494368	77.081290323	0.415146138134271
40	348.079459459	0.09193303175584	89.149487179	0.358947662096456
48	389.549318182	0.082146209751674	103.348888889	0.309630808265089

Table A.11: (Soft Placement) Collocated Parameter Server

Node Count	RINGx1 - Forwd. + Backwd. Images per Second [img / sec]	RINGx1 - Forwd. + Backwd. Seconds per Batch [sec / batch]	PSCPU - Forwd. + Backwd. Images per Second [img / sec]	PSCPU - Forwd. + Backwd. Seconds per Batch [sec / batch]
2	4.9	6.53061224489796	5.06	6.32411067193676
4	6.11	5.23731587561375	6.83	4.68521229868228
8	8.28	3.86473429951691	8.28	3.86473429951691
16	7.99	4.00500625782228	7.99	4.00500625782228
24	-	-	8.85	3.615819209
32	-	-	9.36	3.418803419

Table A.12: (Forward and Backward only) Distributed Allreduce

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	14.67	2.1813224267212	3.6	8.88888888888889
2	28.09	1.13919544321823	6.18	5.17799352750809
4	56.99	0.561502017897877	11.885	2.69246949936895
8	107.44875	0.297816400842262	22.36875	1.43056719754121
16	217.813125	0.146914929942812	31.9625	1.00117324990223
24	321.08875	0.09966091929412	44.757083333	0.714970628490574
32	430.4603125	0.074339025156936	54.40625	0.588167719701321
40	522.529	0.06124062013783	62.64027027	0.51085347911287
48	616.259583333	0.05192617018129	69.297708333	0.461775732124195
56	715.992321429	0.044693216731897	76.690178571	0.417263339273285

Table A.13: Distributed Replicated

Node Count	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
1	5.13	6.23781676413255	4.06	7.88177339901478
2	4.97	6.43863179074447	3.87	8.2687338501292
4	14.403	2.22175935568979	11.38	2.81195079086116
8	33.344285714	0.959684675043569	25.907142857	1.23518059002611
16	69.200714286	0.462422972510761	52.244	0.612510527524692
24	99.866086957	0.320429096353585	75.728636364	0.422561418459823
32	130.217	0.245743643303102	95.441333333	0.335284502872044
40	146.936571429	0.21778104449281	81.954473684	0.390460685811803
48	170.7896	0.187365038620619	92.820222222	0.344752460551807

Table A.14: (Soft Placement) Parameter Server

B

Results of the Parameter Server Dedicated Scalability Experiments

This annex presents the results obtained by extensively evaluating the scalability of the Collocated Parameter Server architecture across different sizes of clusters and batches in different communication mediums. The results of these experiments are discussed in chapter 4. The results are presented here as tables.

B.1. Ethernet Results

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	4.26	1.877934272	1.84	4.347826087
16	7.48	2.139037433	2.46	6.504065041
32	13.31	2.404207363	3.53	9.065155807
64	21.29	3.006106153	3.75	17.06666667
128	30.46	4.202232436	3.16	40.50632911
256	39.62	6.46138314	5.38	47.58364312

Table B.1: 1 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	5.965	1.34115674769489	3.115	2.56821829855538
16	11.31	1.41467727674624	4.955	3.22906155398587
32	20.555	1.55679883240088	6.455	4.95739736638265
64	35.775	1.78895877009085	8.31	7.70156438026474
128	59.99	2.13368894815803	9.8	13.0612244897959
256	71.43	3.58392832143357	10.865	23.5618959963185

Table B.2: 2 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	7.0325	1.13757554212584	4.38	1.82648401826484
16	13.945	1.14736464682682	7.8	2.05128205128205
32	25.872	1.23685837971552	11.23	2.84951024042743
64	47.395	1.35035341280726	14.585	4.38806993486459
128	78.1375	1.63813789793633	18.28	7.00218818380744
256	115.4075	2.21822671836752	21.16	12.0982986767486

Table B.3: 4 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.02125	0.997350786972105	4.81	1.66320166320166
16	15.81	1.01201771030993	9.64	1.6597510373444
32	30.645	1.04421602218959	16.92	1.89125295508274
64	57.31	1.11673355435352	24.84125	2.57635988527147
128	101.3375	1.26310595781423	31.98	4.00250156347717
256	157.9475	1.62079171876731	26.37375	9.70662116687995

Table B.4: 8 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.5675	0.93376130726583	5.1925	1.54068367838228
16	16.975625	0.942527889252973	10.45625	1.53018529587567
32	33.4125	0.957725402169847	20.65	1.54963680387409
64	64.5575	0.991364287650544	33.355625	1.91871685810114
128	119.85375	1.06796825297498	44.703125	2.86333449842712
256	207.09625	1.23614020051063	50.724375	5.04688327850269

Table B.5: 16 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.78	0.911161731207289	5.48	1.45985401459854
16	17.408695652	0.919080919090101	11.78	1.35823429541596
32	34.451666667	0.928837501805149	22.49	1.42285460204535
64	66.8925	0.956758978958777	39.748333333	1.61013040380402
128	126.315833333	1.01333298148428	55.69	2.29843778057102
256	227.275416667	1.12638667108941	69.659583333	3.67501480415437

Table B.6: 24 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.600625	0.930164958941937	5.400645161	1.48130450372316
16	17.670357143	0.905471229048605	11.11	1.44014401440144
32	34.8703125	0.917686068916073	22.448125	1.42550881198318
64	68.4028125	0.935634042825359	43.8259375	1.4603224403357
128	131.579375	0.972796838410275	78.017741935	1.64065245706089
256	245.5084375	1.04273402008841	111.418125	2.29765130224548

Table B.7: 32 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.93	0.895856662933931	5.68	1.40845070422535
16	17.811935484	0.898274082250769	11.34	1.41093474426808
32	35.291891892	0.906723847447062	23	1.39130434782609
64	69.26	0.924054288189431	45.56	1.40474100087796
128	133.67125	0.957573150546584	72.0355	1.77690166653941
256	247.78175	1.03316729339429	95.186	2.68947114071397

Table B.8: 40 Node Cluster

B.2. InfiniBand Results

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	4.24	1.886792453	1.88	4.255319149
16	7.35	2.176870748	2.4	6.666666667
32	13.23	2.418745276	3.48	9.195402299
64	19.14	3.343782654	3.04	21.05263158
128	30.79	4.157193894	4.94	25.91093117
256	39.5	6.481012658	4.93	51.92697769

Table B.9: 1 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	8.285	0.965600482800241	3.67	2.17983651226158
16	14.945	1.07059217129475	5.37	2.97951582867784
32	26.395	1.2123508240197	6.945	4.60763138948884
64	42.97	1.48941121712823	8.61	7.43321718931475
128	61.305	2.08792105048528	9.86	12.9817444219067
256	79.48	3.22093608454957	10.92	23.4432234432234

Table B.10: 2 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	15.16	0.527704485488127	7.2225	1.10764970578055
16	29.98	0.533689126084056	10.755	1.48768014876801
32	52.16	0.613496932515337	13.8025	2.31842057598261
64	84.99	0.753029768208025	17.095	3.74378473237789
128	118.81	1.0773503913812	19.59	6.53394589076059
256	158.0425	1.61981745416581	21.77	11.7593017914561

Table B.11: 4 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	30.31375	0.26390664302503	14.12375	0.566421807239579
16	58.3025	0.274430770550148	21.12375	0.757441268714125
32	101.935	0.313925540785795	27.36	1.16959064327485
64	166.9475	0.383354048428398	33.92125	1.88672292442053
128	239.3875	0.534697927001201	38.78625	3.30013858003803
256	313.21125	0.817339734763678	43.40875	5.8974285138365

Table B.12: 8 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	56.549375	0.141469291216746	26.86625	0.297771367421951
16	105.690625	0.151385234026197	40.745	0.392686219167996
32	188.21625	0.170017200959003	42.414375	0.754461193875897
64	310.451875	0.206151114403803	66.755	0.9587296831698
128	463.198125	0.276339633283274	76.81625	1.66631409369762
256	609.44625	0.420053450160699	85.98125	2.97739332703351

Table B.13: 16 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	46.233043478	0.173036412880904	37.99875	0.21053324122504
16	143.743333333	0.111309510006519	57.68625	0.277362456391254
32	254.504166667	0.125734680178614	60.49625	0.5289584065128
64	426.937916667	0.14990469925846	85.91625	0.744911469017793
128	653.5125	0.195864654463381	89.457916667	1.43084038583717
256	841.480833333	0.304225586441483	87.2175	2.93519075873535

Table B.14: 24 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	46.5090625	0.172009487398289	46.0821875	0.173602869872443
16	173.668387097	0.092129605551432	70.7409375	0.226177381378357
32	299.2615625	0.106929870086473	79.4953125	0.402539457908289
64	517.0103125	0.123788633326342	101.2015625	0.632401303092529
128	800.10625	0.159978752821891	110.209375	1.16142569540931
256	963.454333333	0.265710569918126	113.7475	2.25059891426185

Table B.15: 32 Node Cluster

Batch Size	Forwd. Images per Second [img / sec]	Forwd. Seconds per Batch [sec / batch]	Forwd. + Backwd. Images per Second [img / sec]	Forwd. + Backwd. Seconds per Batch [sec / batch]
8	107.9205128	0.07412863218	54.317	0.1472835392
16	203.3465	0.07868342952	82.84975	0.1931206793
32	352.1448649	0.09087169285	95.71425	0.334328483
64	565.6193939	0.1131502927	116.0835	0.5513272773
128	814.7869444	0.1570962825	124.6695	1.026714633
256	1128.427179	0.2268644399	132.095	1.937999167

Table B.16: 40 Node Cluster

C

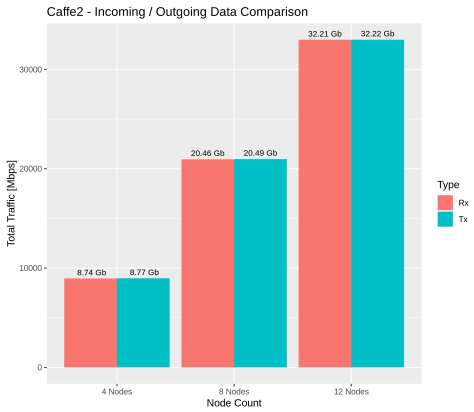
Results of the Hardware Consumption Patterns of the Parameter Update Architectures

This annex presents the results obtained by surveying the CPU utilization, Memory Footprint and Network Traffic of the parameter update architecture across different sized clusters and communication mediums. The results of these experiments are discussed in chapter 5. The results are presented here as plots.

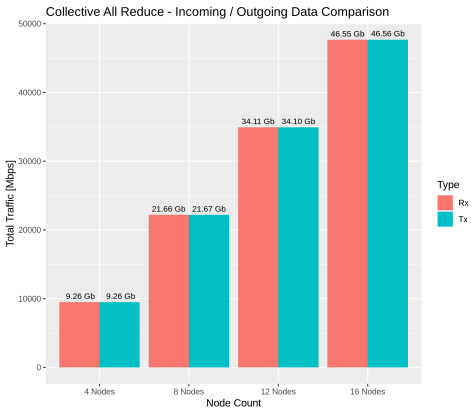
C.1. Total Traffic

This section describes the total incoming and outgoing traffic for each evaluated architecture, across an array of cluster sizes, in Ethernet and InfiniBand communication mediums. The plots in figure C.1 presents the Ethernet results while those in figure C.2 show the InfiniBand results.

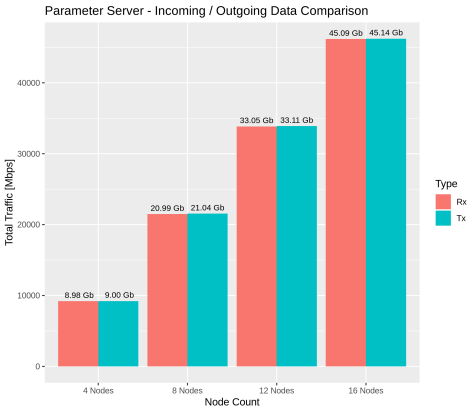
C.1.1. Ethernet Results



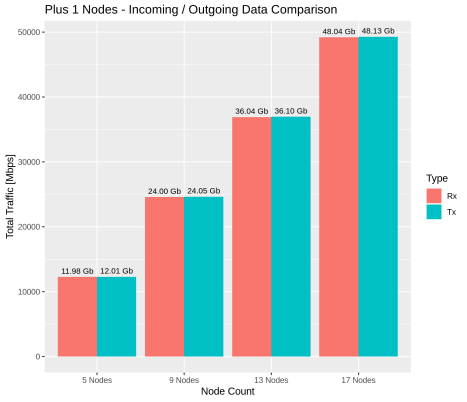
(a) Caffe2



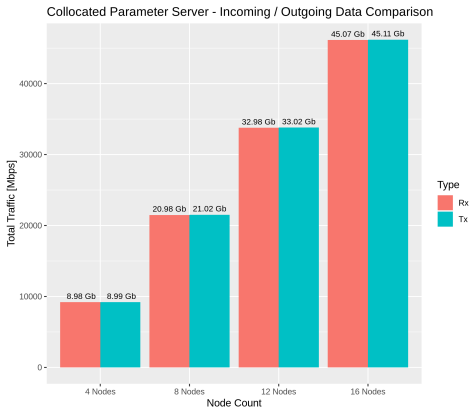
(b) Collective Allreduce



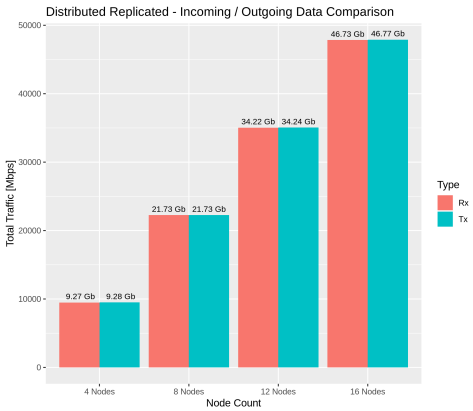
(c) Non-collocated Parameter Server



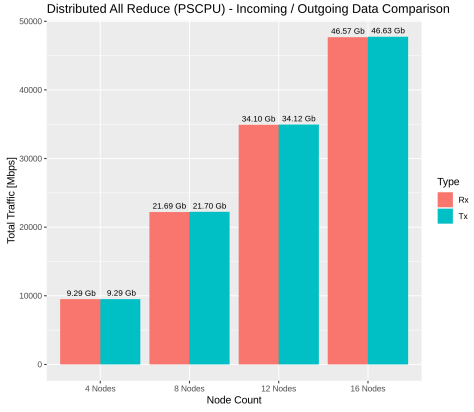
(d) Non-collocated Parameter Server with extra worker



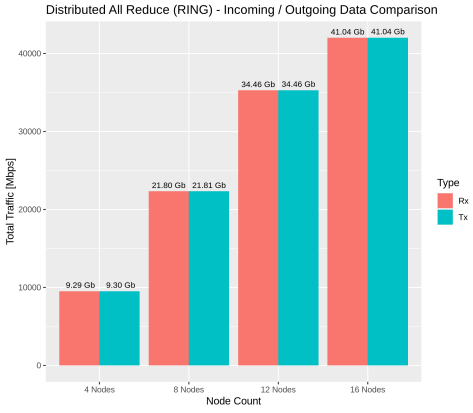
(e) Collocated Parameter Server



(f) Distributed Replicated



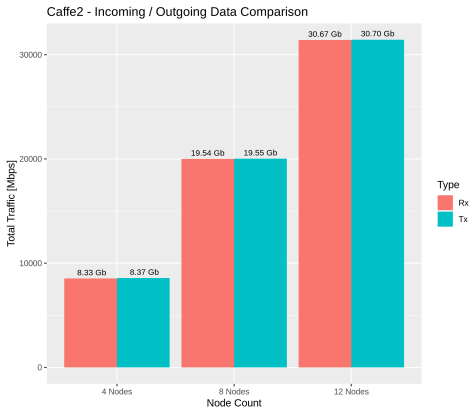
(g) Distributed Allreduce with PSCPU reduction strategy



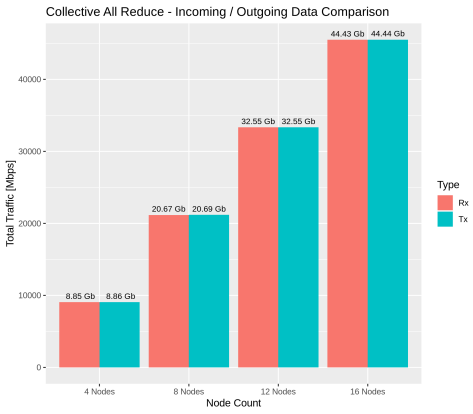
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.1: The cumulative Network Traffic of all the nodes in the clusters. Each plot presents both the cumulative incoming and outgoing traffic across a range of cluster sizes in an Ethernet medium. The y axis is unnormalized across plots.

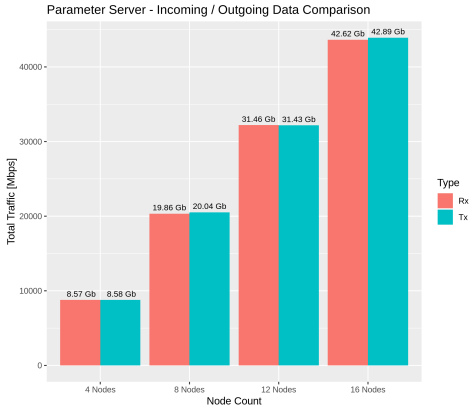
C.1.2. InfiniBand Results



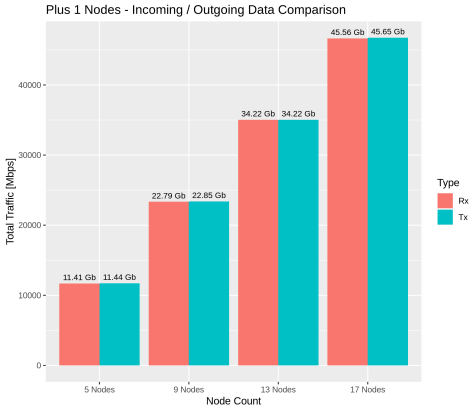
(a) Caffe2



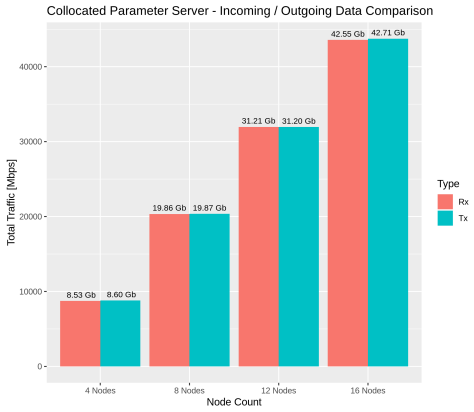
(b) Collective Allreduce



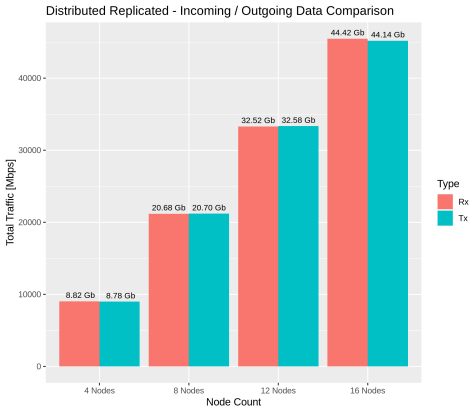
(c) Non-collocated Parameter Server



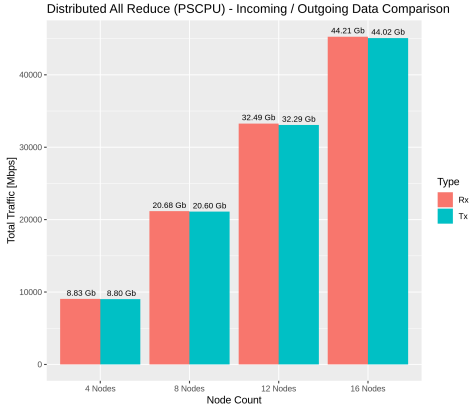
(d) Non-collocated Parameter Server with extra worker



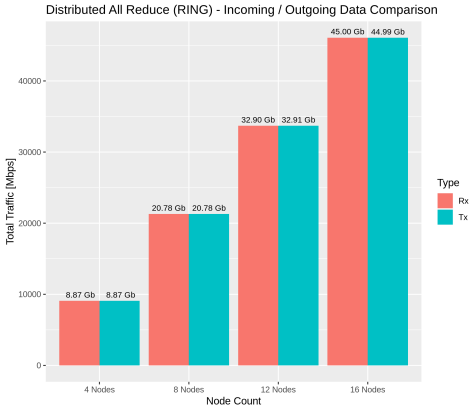
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.2: The cumulative Network Traffic of all the nodes in the clusters. Each plot presents both the cumulative incoming and outgoing traffic across a range of cluster sizes in an InfiniBand medium. The y axis is unnormalized across plots.

C.2. Per Process Resource Consumption

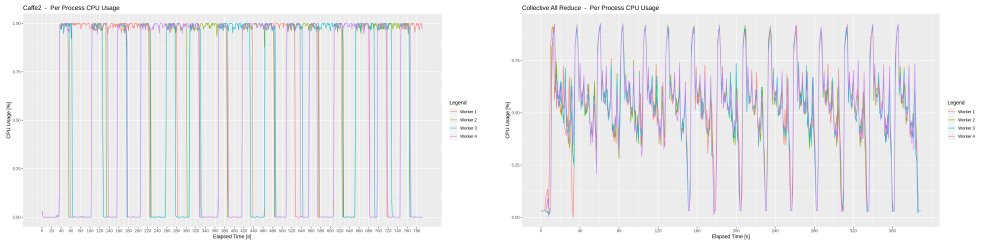
The plots in this section show the per process hardware resource utilization patterns across three dimensions: CPU usage, Network Traffic, and Memory Footprint. The results are presented for 4, 8, 12 and 16 node clusters. Section C.2.1 shows the results for commodity Ethernet interconnections, while section C.2.2 presents the results for InfiniBand interconnections. Results are shown for the clusters of size 4, 8, 12 and 16.

C.2.1. Ethernet Results

The following subsections present the results for the CPU usage, Memory Footprint and Network Traffic, in the case when Ethernet interconnections are used. Plots are presented for all the tested cluster sizes.

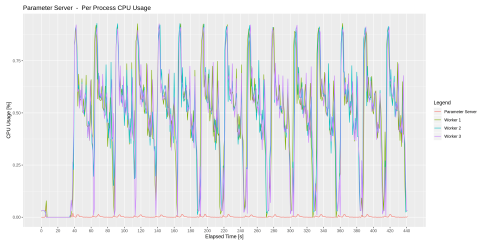
4 Nodes

CPU Utilization

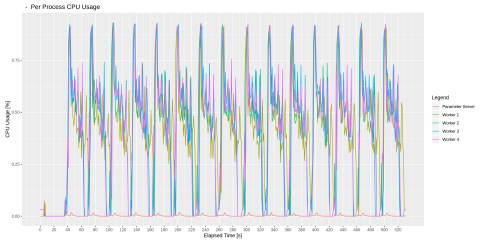


(a) Caffe2

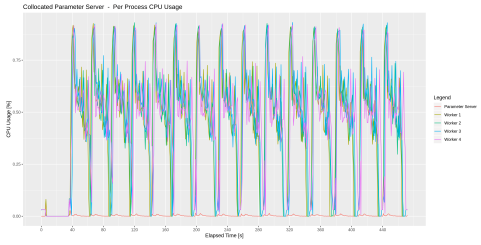
(b) Collective Allreduce



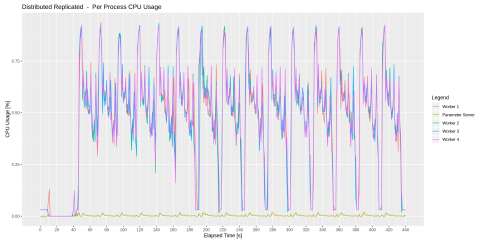
(c) Non-collocated Parameter Server



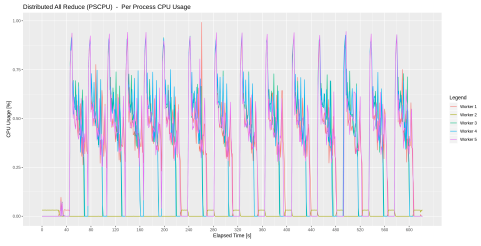
(d) Non-collocated Parameter Server with extra worker



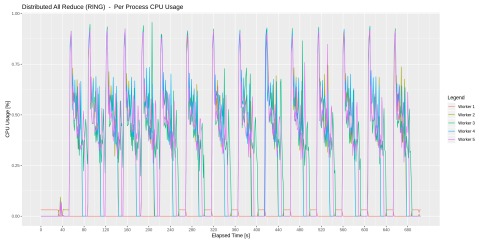
(e) Collocated Parameter Server



(f) Distributed Replicated



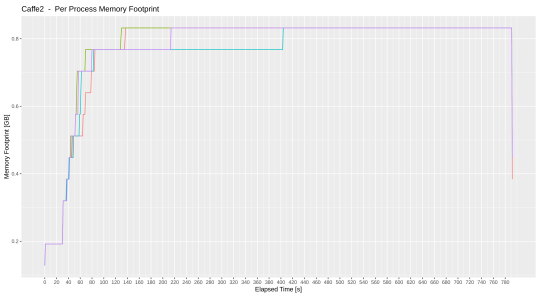
(g) Distributed Allreduce with PSCPU reduction strategy



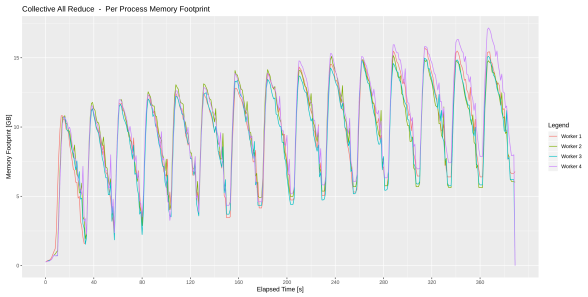
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.3: Each plot presents the per process CPU consumption in an Ethernet medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 4 node cluster.

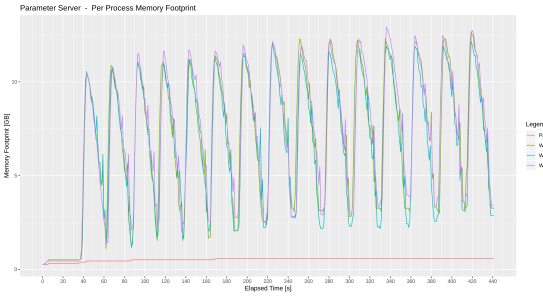
Memory Footprint



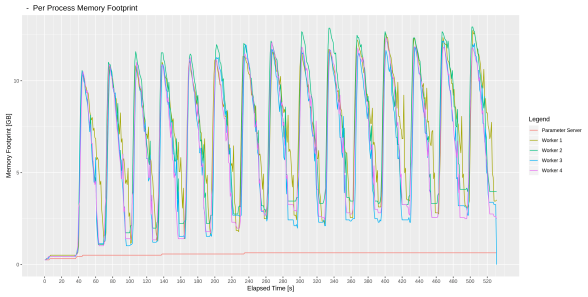
(a) Caffe2



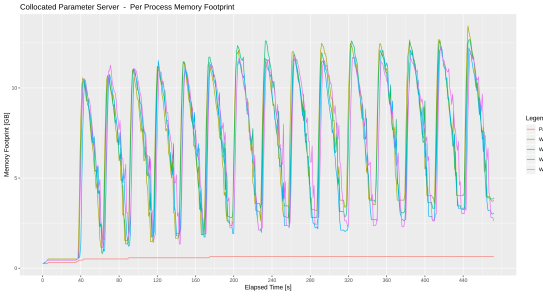
(b) Collective Allreduce



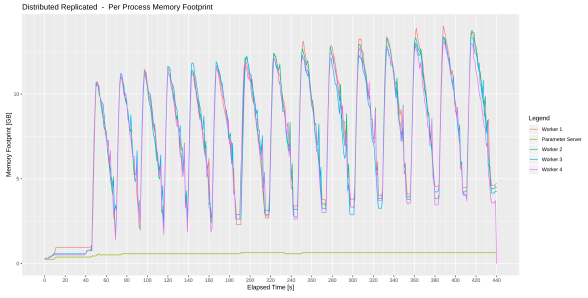
(c) Non-collocated Parameter Server



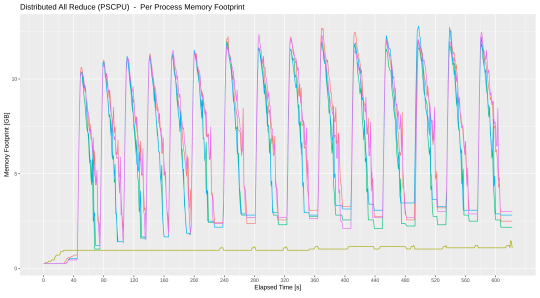
(d) Non-collocated Parameter Server with extra worker



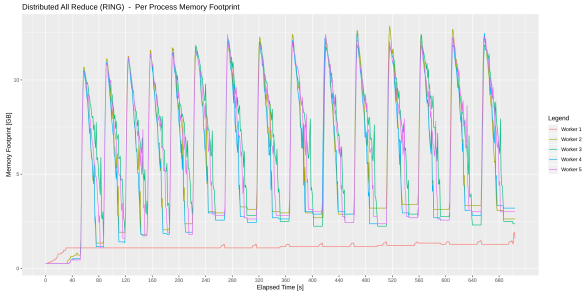
(e) Collocated Parameter Server



(f) Distributed Replicated



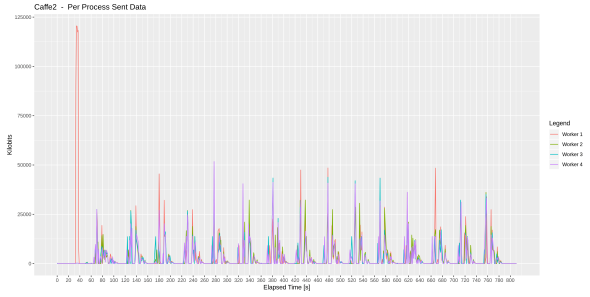
(g) Distributed Allreduce with PSCPU reduction strategy



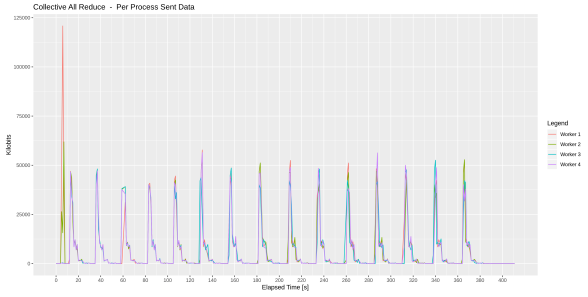
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.4: Each plot presents the per process Memory Footprint in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 4 node cluster.

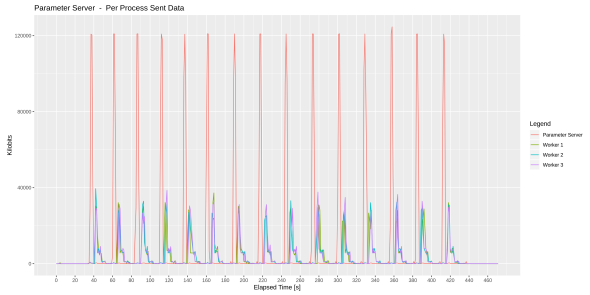
Network Traffic



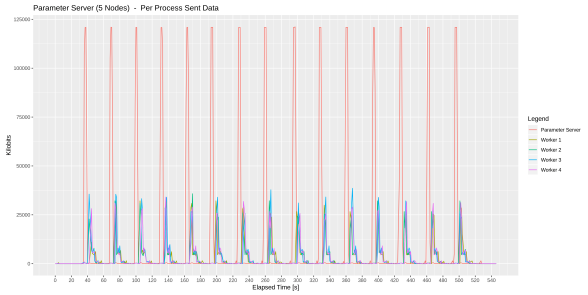
(a) Caffe2



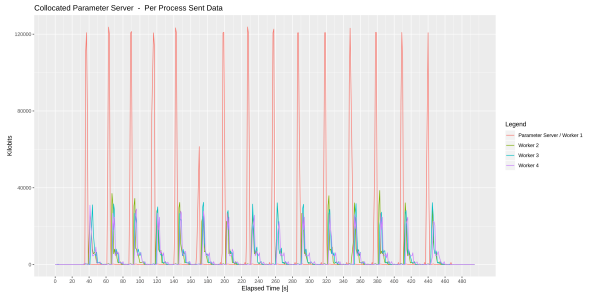
(b) Collective Allreduce



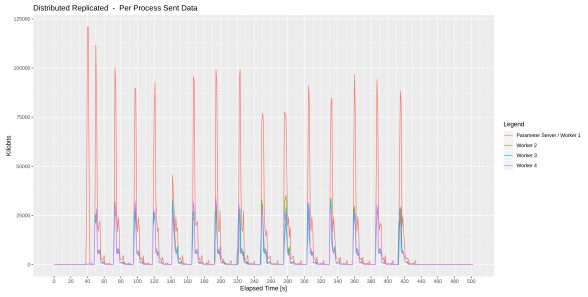
(c) Non-collocated Parameter Server



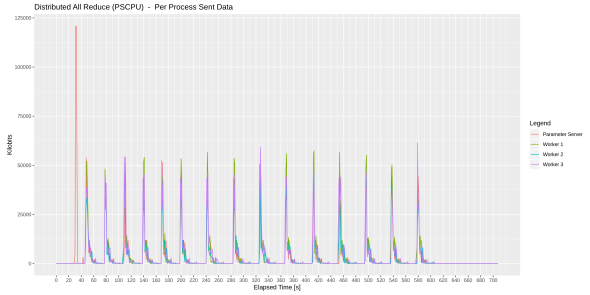
(d) Non-collocated Parameter Server with extra worker



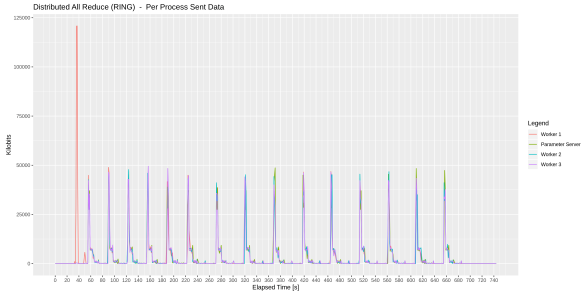
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy

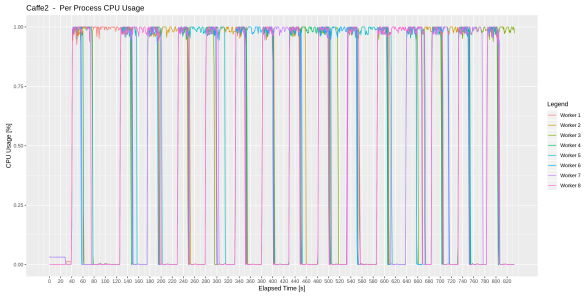


(h) Distributed Allreduce with Ringx1 reduction strategy

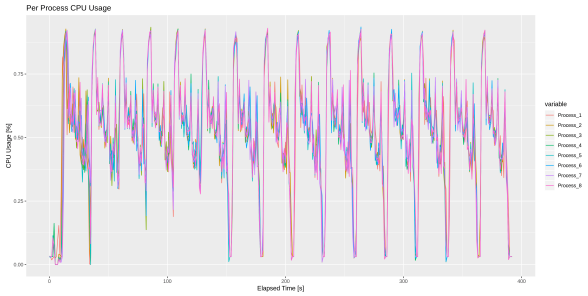
Figure C.5: Each plot presents the per process outgoing Network Traffic in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 4 node cluster.

8 Nodes

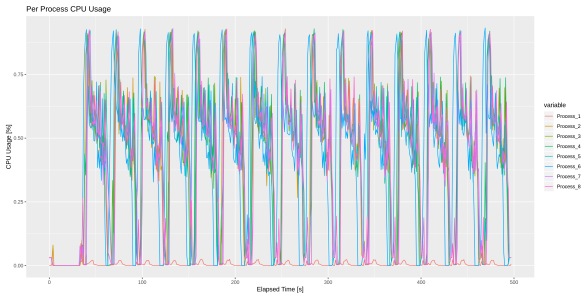
CPU Utilization



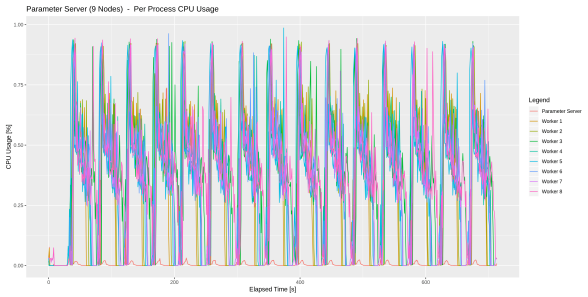
(a) Caffe2



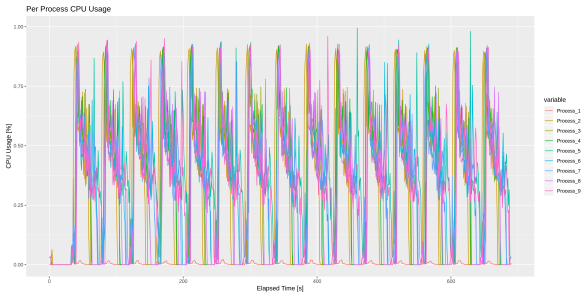
(b) Collective Allreduce



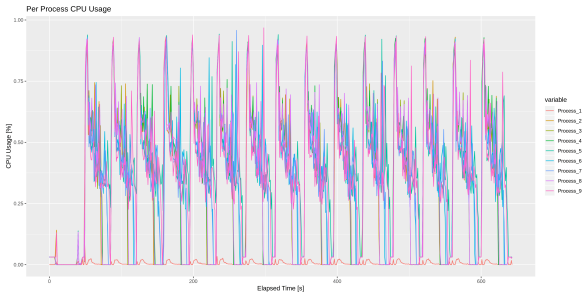
(c) Non-collocated Parameter Server



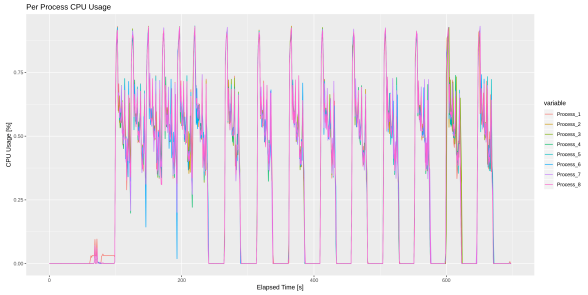
(d) Non-collocated Parameter Server with extra worker



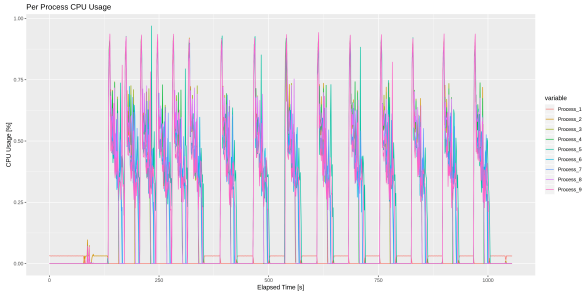
(e) Collocated Parameter Server



(f) Distributed Replicated



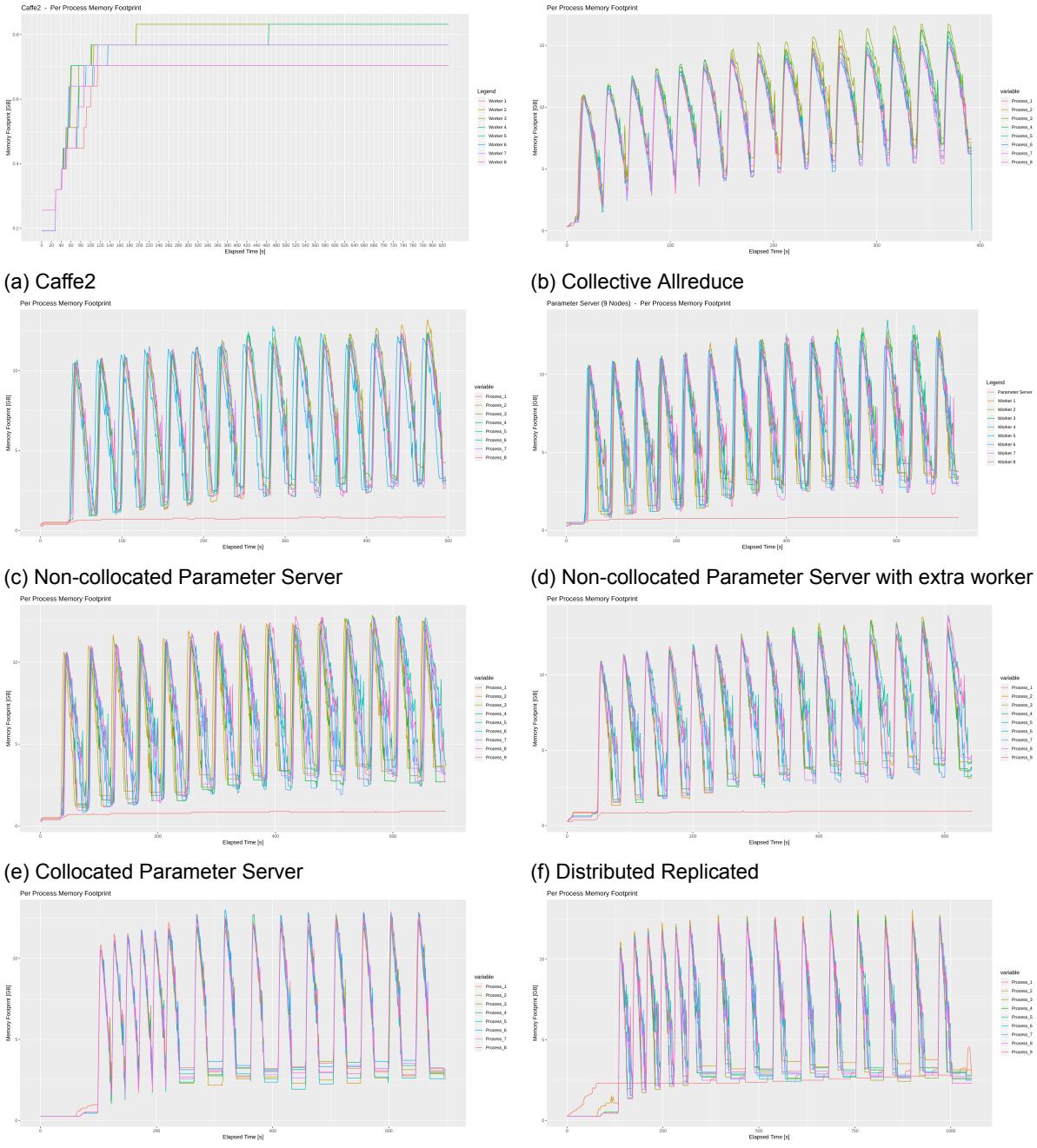
(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.6: Each plot presents the per process CPU consumption in an Ethernet medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 8 node cluster.

Memory Footprint



(a) Caffe2

(b) Collective Allreduce

(c) Non-collocated Parameter Server

(d) Non-collocated Parameter Server with extra worker

(e) Collocated Parameter Server

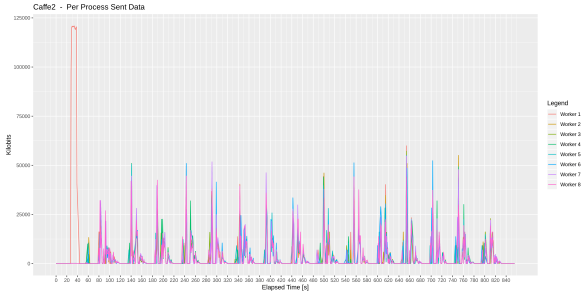
(f) Distributed Replicated

(g) Distributed Allreduce with PSCPU reduction strategy

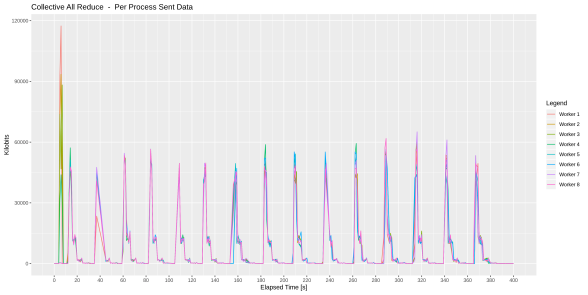
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.7: Each plot presents the per process Memory Footprint in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 8 node cluster.

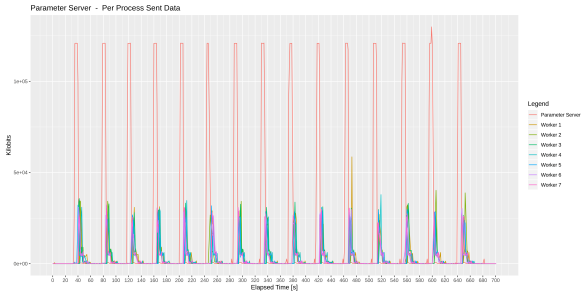
Network Traffic



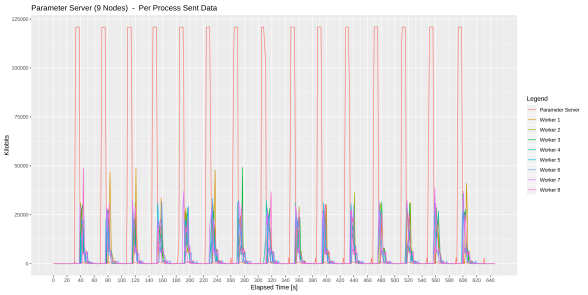
(a) Caffe2



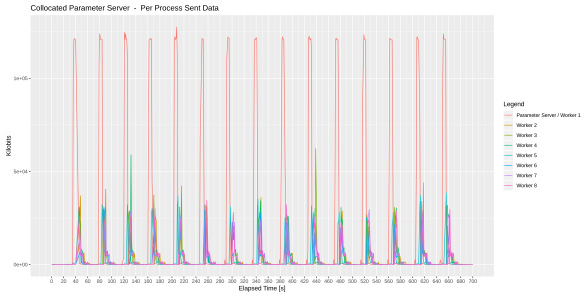
(b) Collective Allreduce



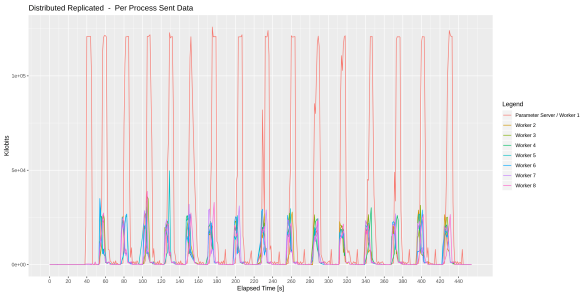
(c) Non-collocated Parameter Server



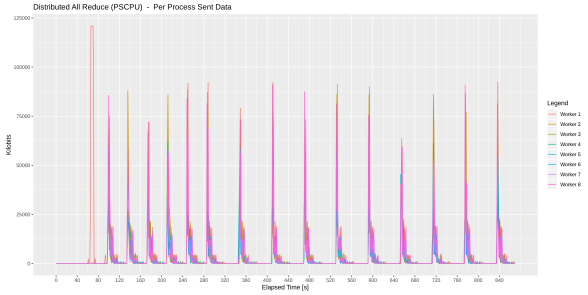
(d) Non-collocated Parameter Server with extra worker



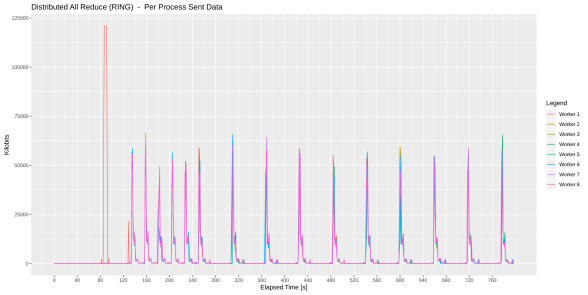
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy

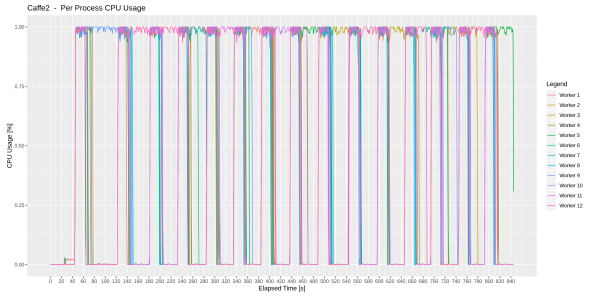


(h) Distributed Allreduce with Ringx1 reduction strategy

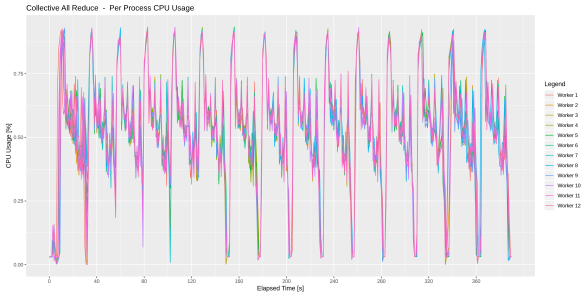
Figure C.8: Each plot presents the per process outgoing Network Traffic in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 8 node cluster.

12 Nodes

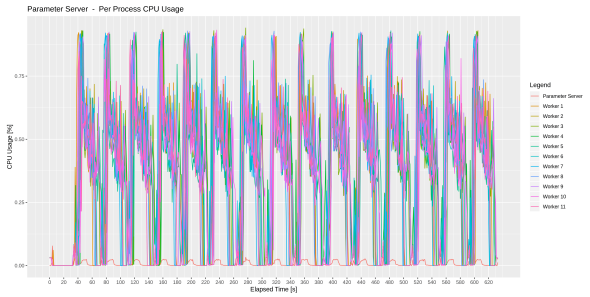
CPU Utilization



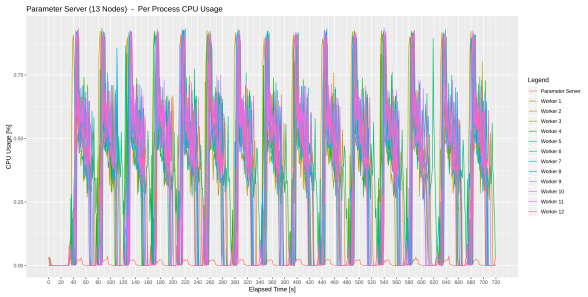
(a) Caffe2



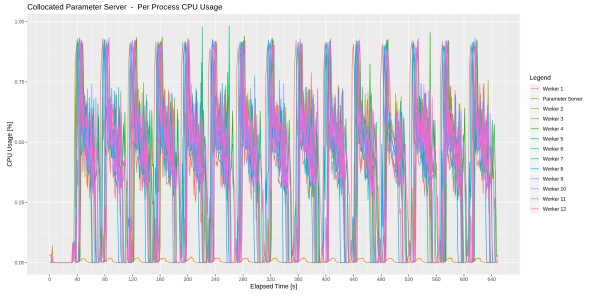
(b) Collective Allreduce



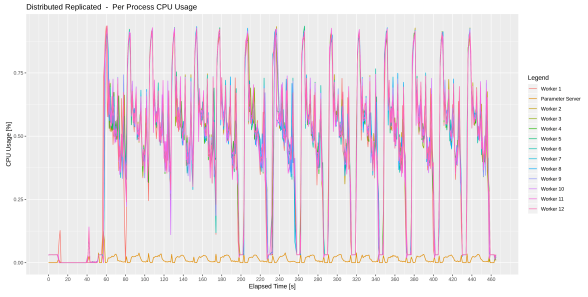
(c) Non-collocated Parameter Server



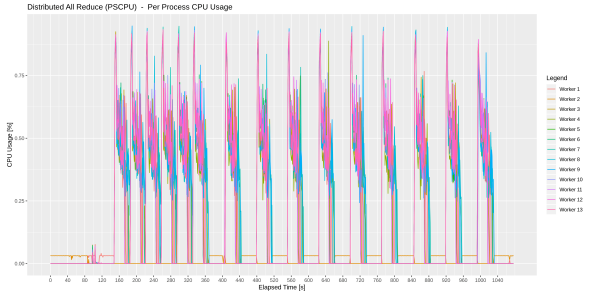
(d) Non-collocated Parameter Server with extra worker



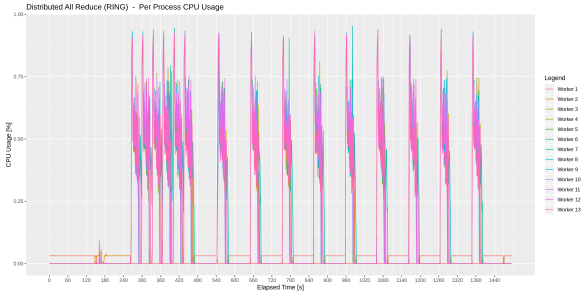
(e) Collocated Parameter Server



(f) Distributed Replicated



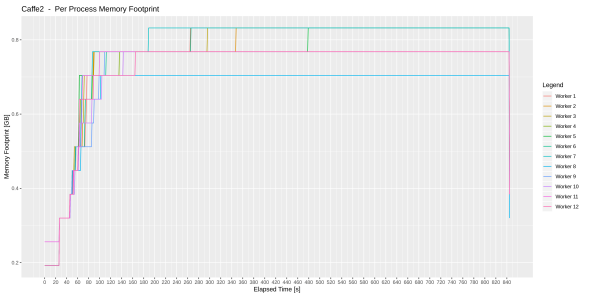
(g) Distributed Allreduce with PSCPU reduction strategy



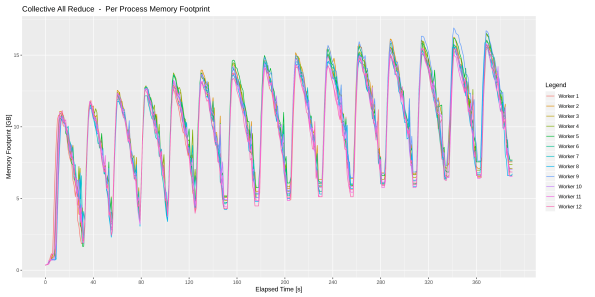
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.9: Each plot presents the per process CPU consumption in an Ethernet medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 12 node cluster.

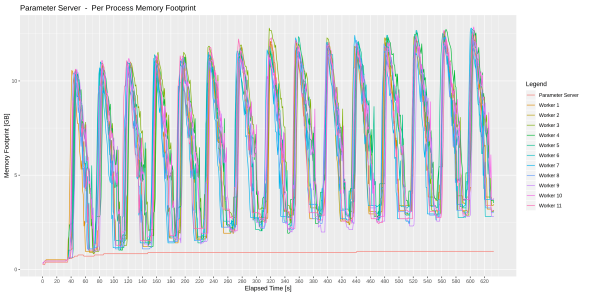
Memory Footprint



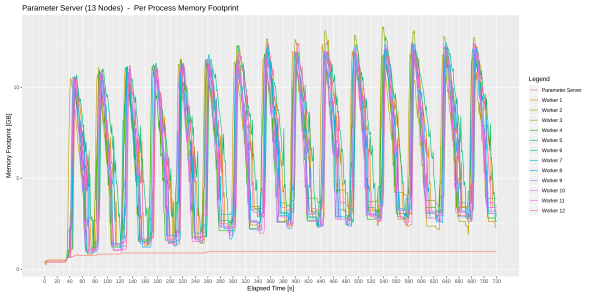
(a) Caffe2



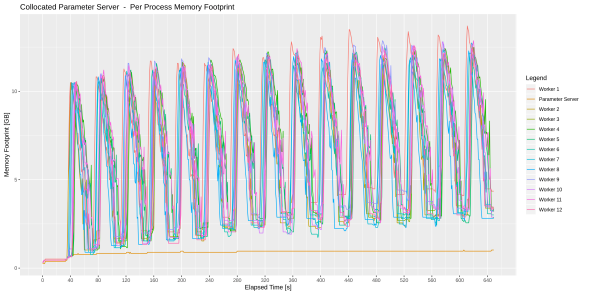
(b) Collective Allreduce



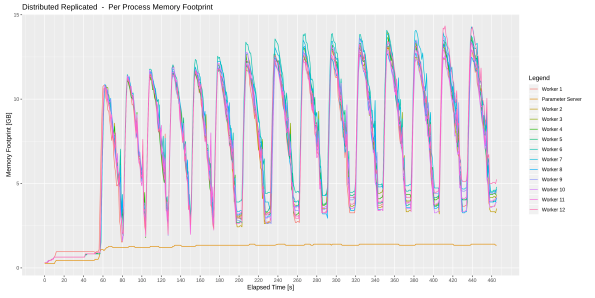
(c) Non-collocated Parameter Server



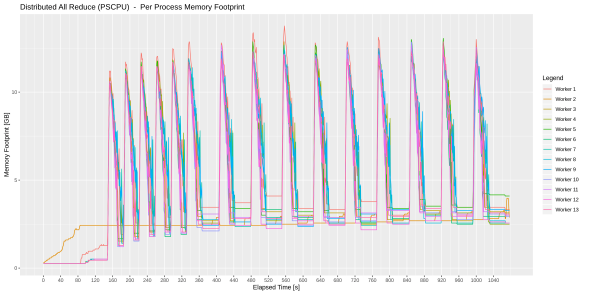
(d) Non-collocated Parameter Server with extra worker



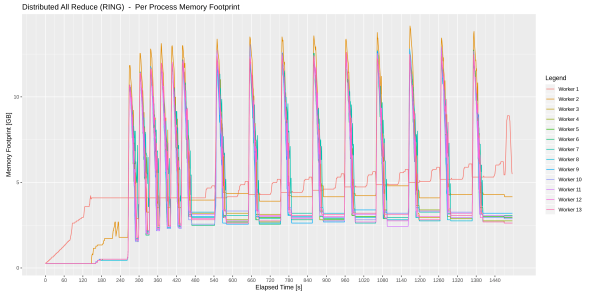
(e) Collocated Parameter Server



(f) Distributed Replicated



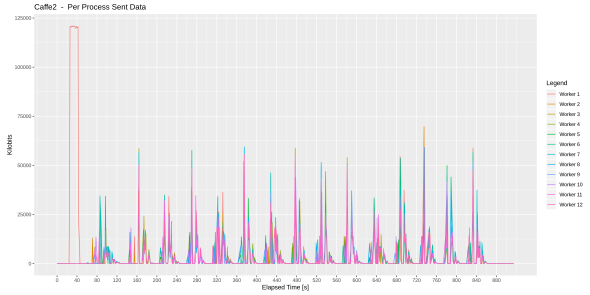
(g) Distributed Allreduce with PSCPU reduction strategy



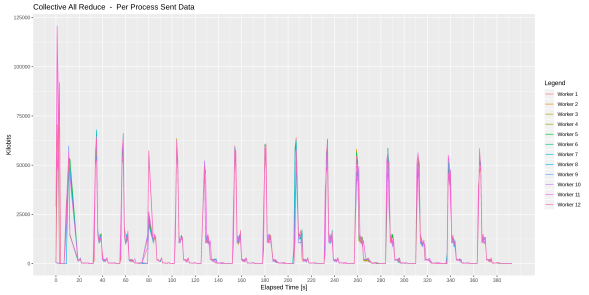
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.10: Each plot presents the per process Memory Footprint in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 12 node cluster.

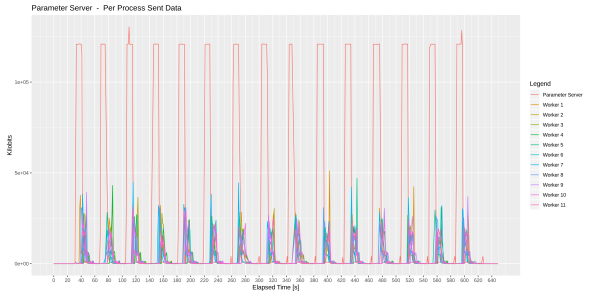
Network Traffic



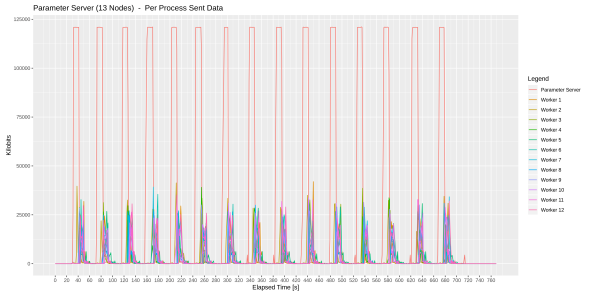
(a) Caffe2



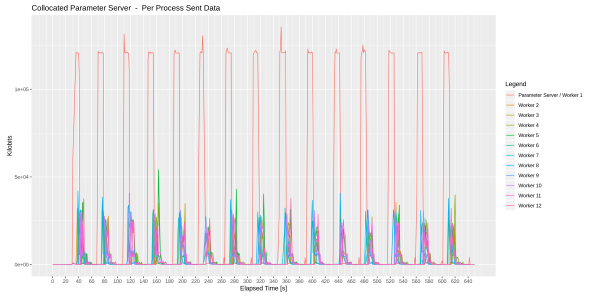
(b) Collective Allreduce



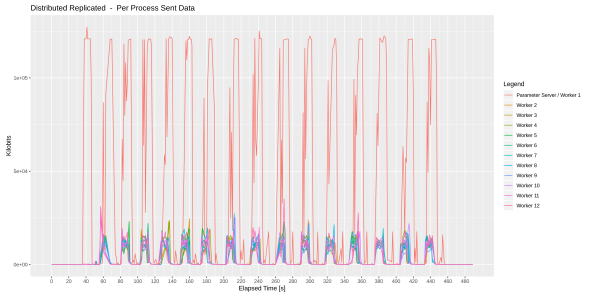
(c) Non-collocated Parameter Server



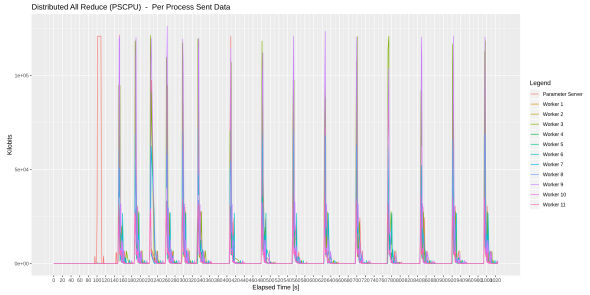
(d) Non-collocated Parameter Server with extra worker



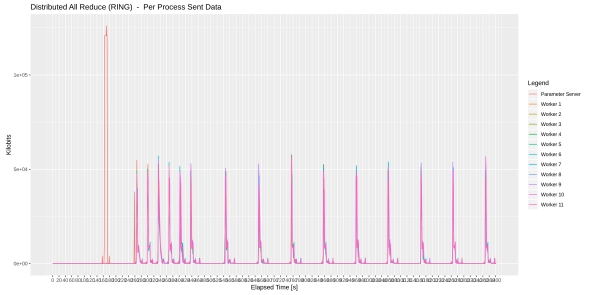
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy

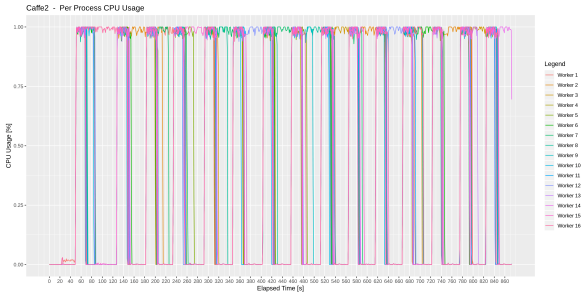


(h) Distributed Allreduce with Ringx1 reduction strategy

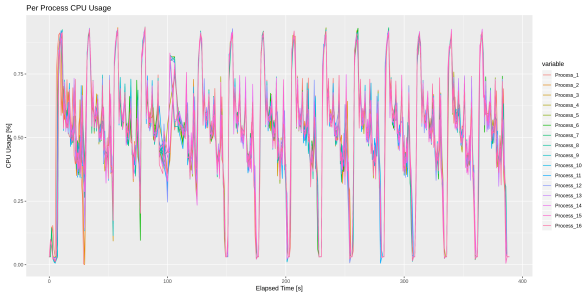
Figure C.11: Each plot presents the per process outgoing Network Traffic in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 12 node cluster.

16 Nodes

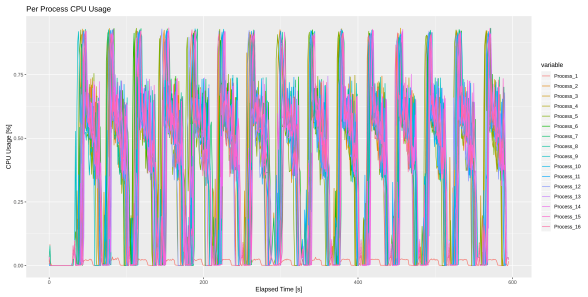
CPU Utilization



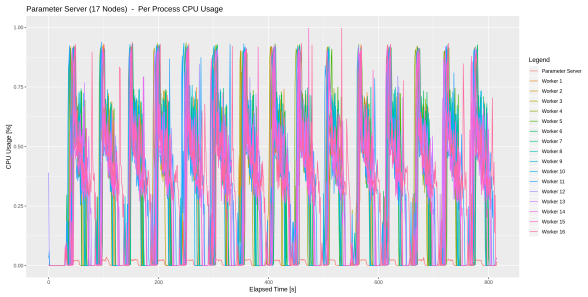
(a) Caffe2



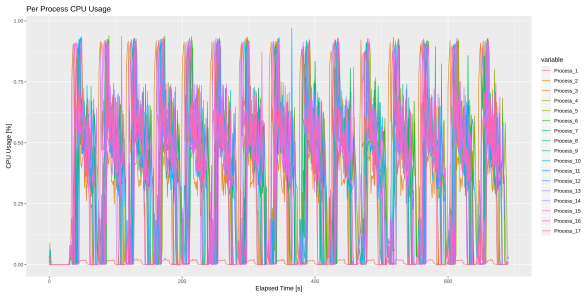
(b) Collective Allreduce



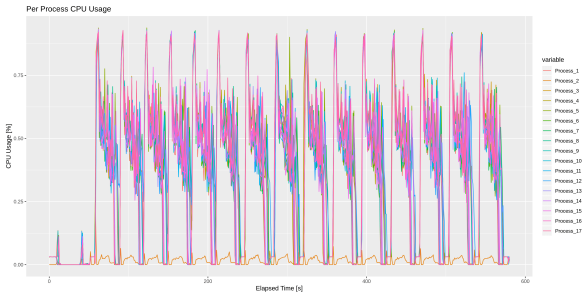
(c) Non-collocated Parameter Server



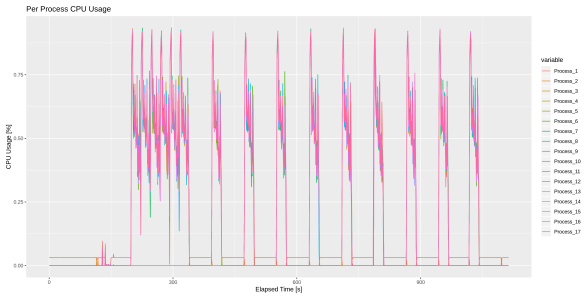
(d) Non-collocated Parameter Server with extra worker



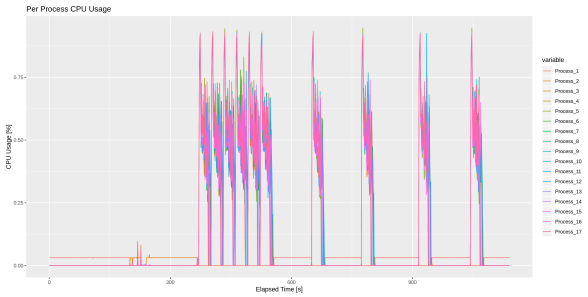
(e) Collocated Parameter Server



(f) Distributed Replicated



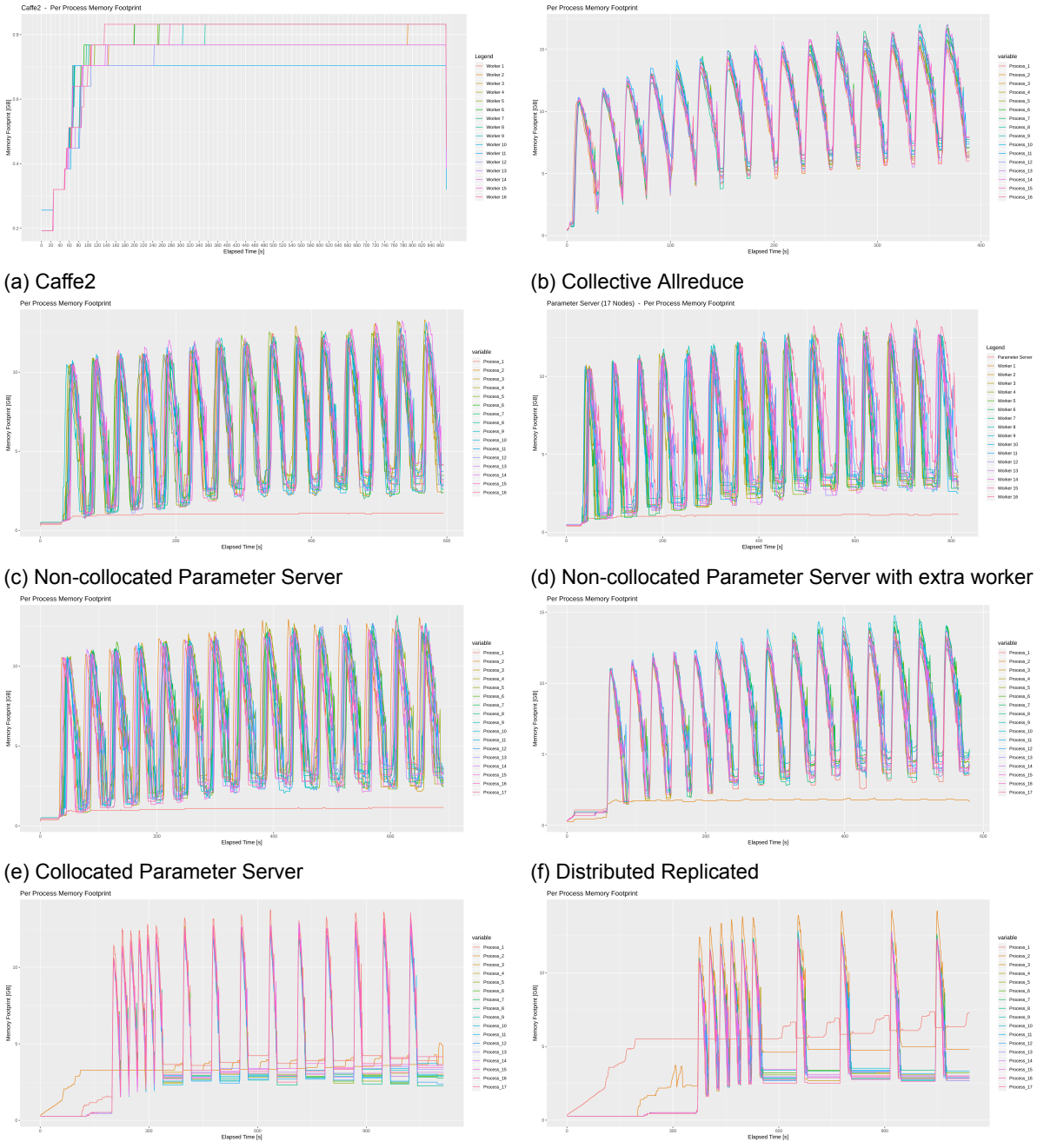
(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.12: Each plot presents the per process CPU consumption in an Ethernet medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 16 node cluster.

Memory Footprint



(a) Caffe2

(b) Collective Allreduce

(c) Non-collocated Parameter Server

(d) Non-collocated Parameter Server with extra worker

(e) Collocated Parameter Server

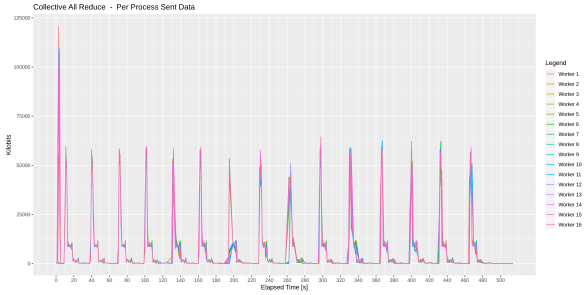
(f) Distributed Replicated

(g) Distributed Allreduce with PSCPU reduction strategy

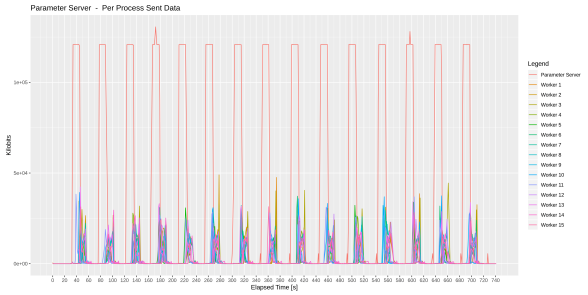
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.13: Each plot presents the per process Memory Footprint in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 16 node cluster.

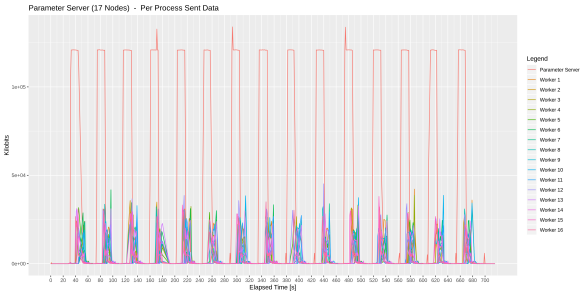
Network Traffic



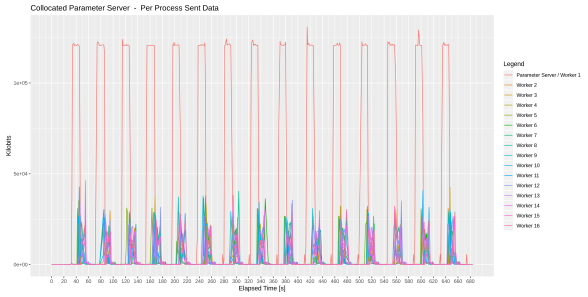
(a) Collective Allreduce



(b) Non-collocated Parameter Server

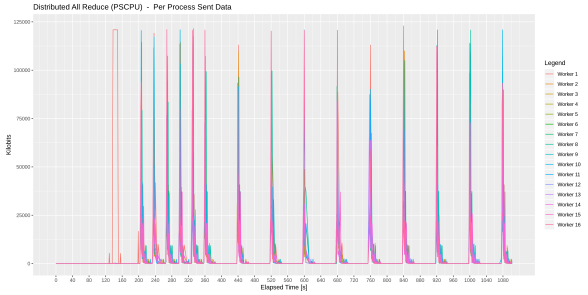


(c) Non-collocated Parameter Server with extra worker

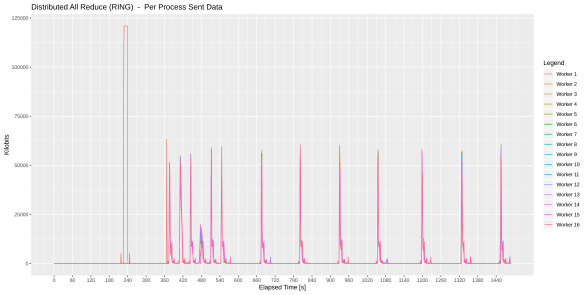


(d) Collocated Parameter Server

(e) Distributed Replicated



(f) Distributed Allreduce with PSCPU reduction strategy



(g) Distributed Allreduce with Ringx1 reduction strategy

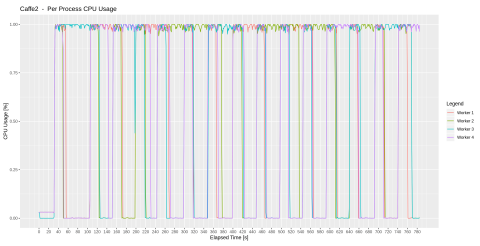
Figure C.14: Each plot presents the per process outgoing Network Traffic in an Ethernet medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 16 node cluster.

C.2.2. InfiniBand Results

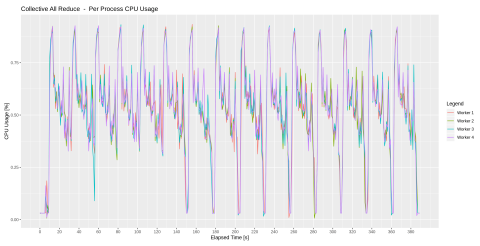
The following subsections present the results for the CPU usage, Memory Footprint and Network Traffic, in the case when InfiniBand interconnections are used. Plots are presented for all the tested cluster sizes.

4 Nodes

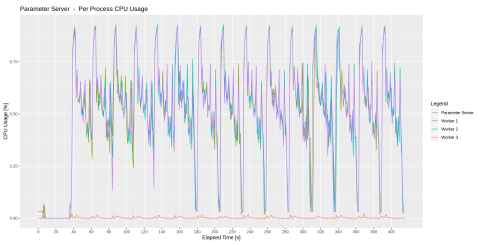
CPU Utilization



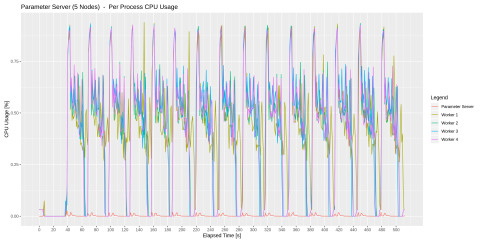
(a) Caffe2



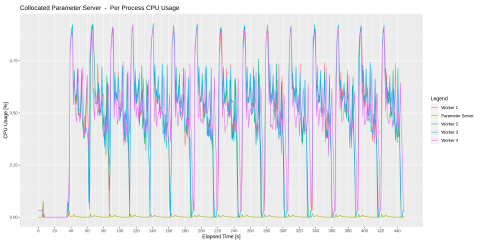
(b) Collective Allreduce



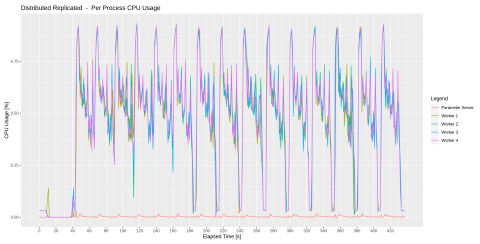
(c) Non-collocated Parameter Server



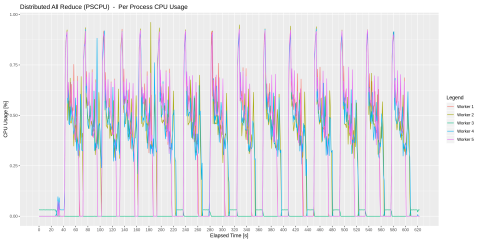
(d) Non-collocated Parameter Server with extra worker



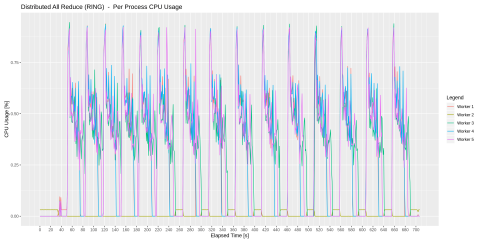
(e) Collocated Parameter Server



(f) Distributed Replicated



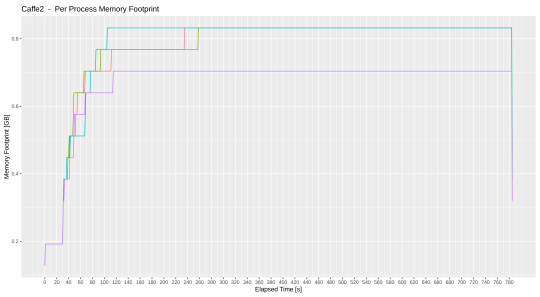
(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

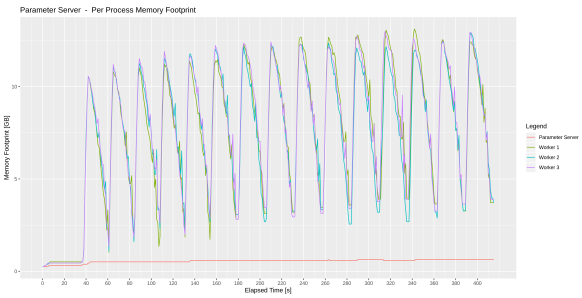
Figure C.15: Each plot presents the per process CPU consumption in an InfiniBand medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 4 node cluster.

Memory Footprint

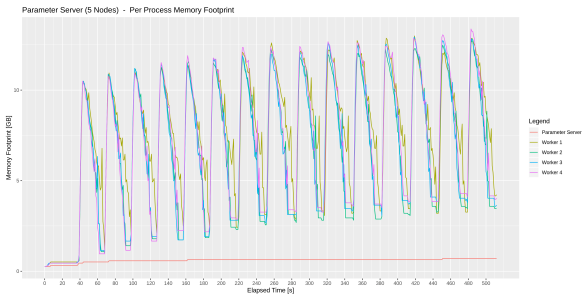


(a) Caffe2

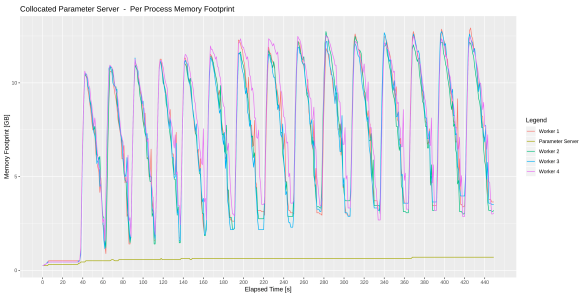
(b) Collective Allreduce



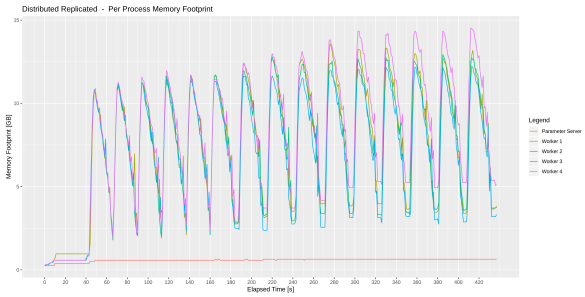
(c) Non-collocated Parameter Server



(d) Non-collocated Parameter Server with extra worker

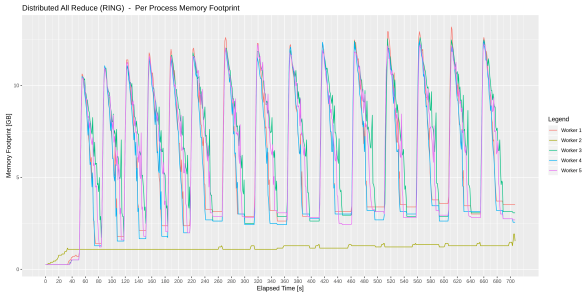


(e) Collocated Parameter Server



(f) Distributed Replicated

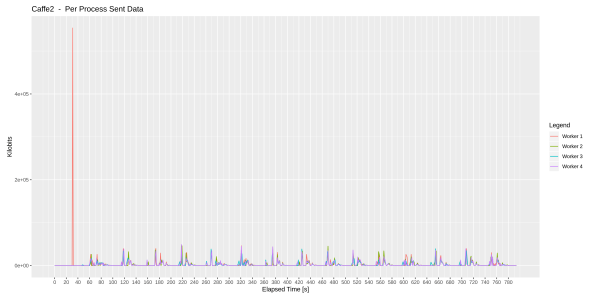
(g) Distributed Allreduce with PSCPU reduction strategy



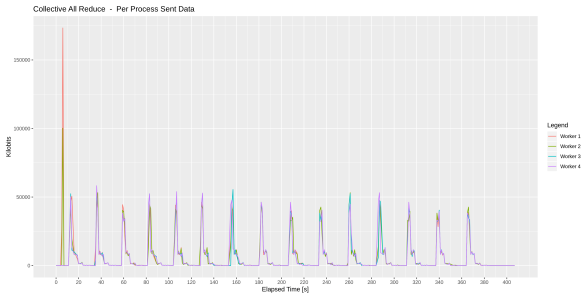
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.16: Each plot presents the per process Memory Footprint in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 4 node cluster.

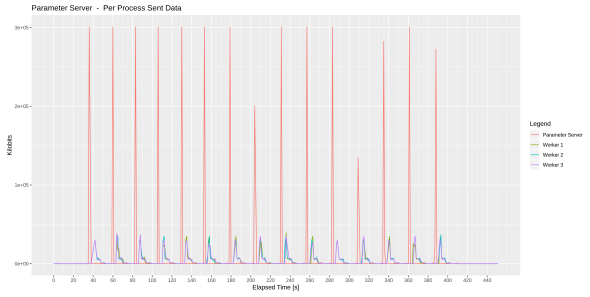
Network Traffic



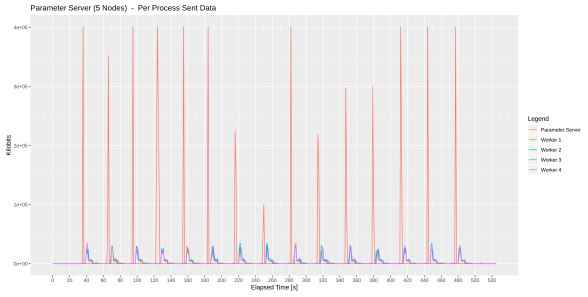
(a) Caffe2



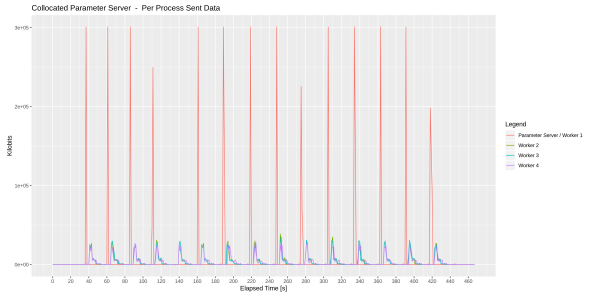
(b) Collective Allreduce



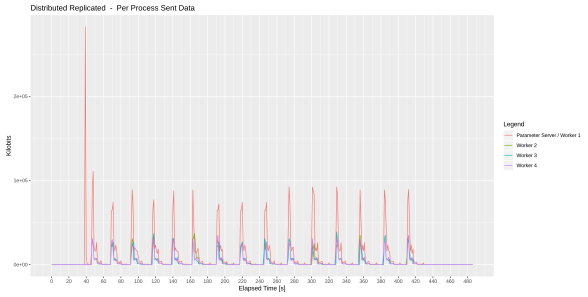
(c) Non-collocated Parameter Server



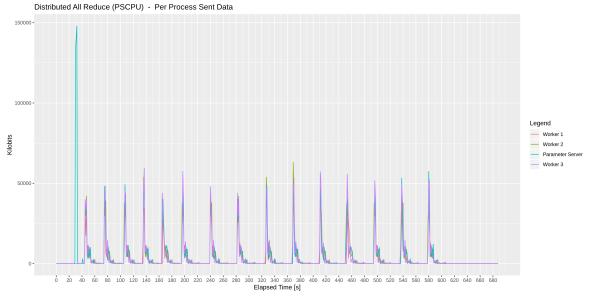
(d) Non-collocated Parameter Server with extra worker



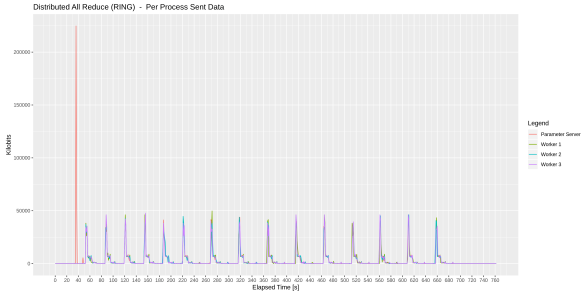
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy

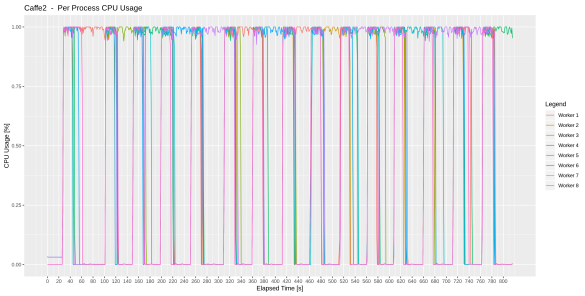


(h) Distributed Allreduce with Ringx1 reduction strategy

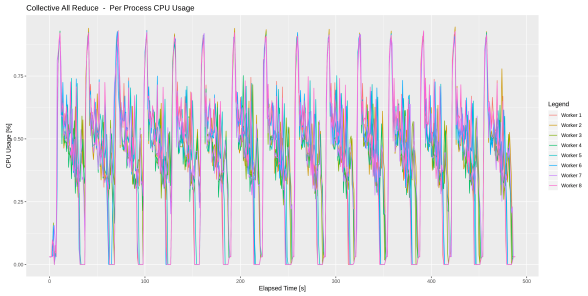
Figure C.17: Each plot presents the per process outgoing Network Traffic in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 4 node cluster.

8 Nodes

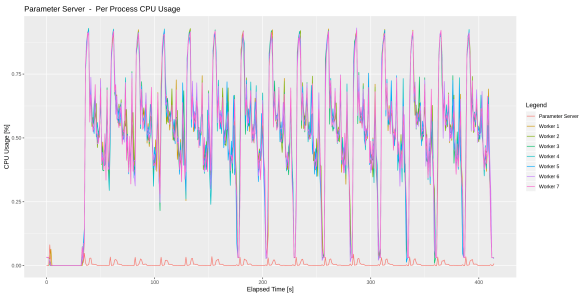
CPU Utilization



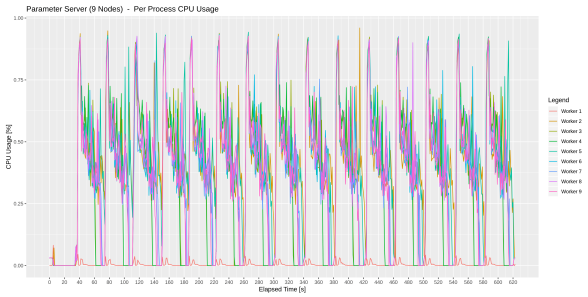
(a) Caffe2



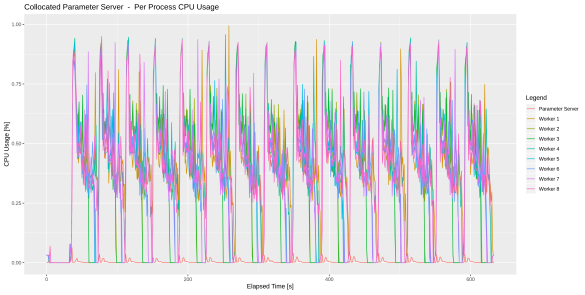
(b) Collective Allreduce



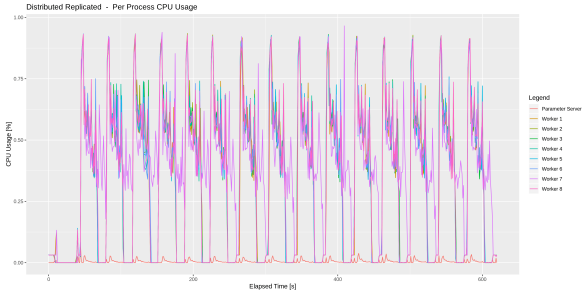
(c) Non-collocated Parameter Server



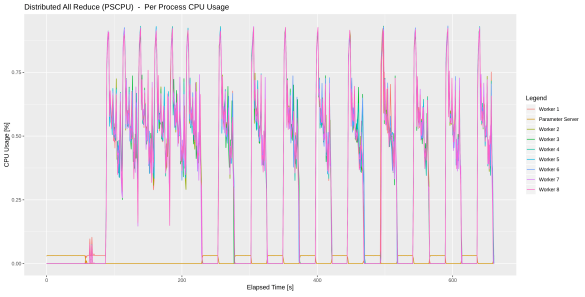
(d) Non-collocated Parameter Server with extra worker



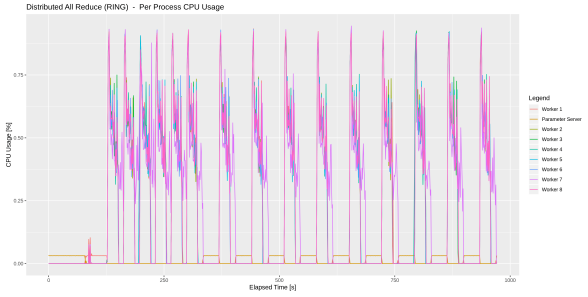
(e) Collocated Parameter Server



(f) Distributed Replicated



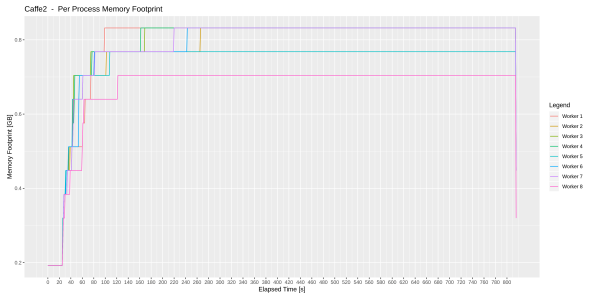
(g) Distributed Allreduce with PSCPU reduction strategy



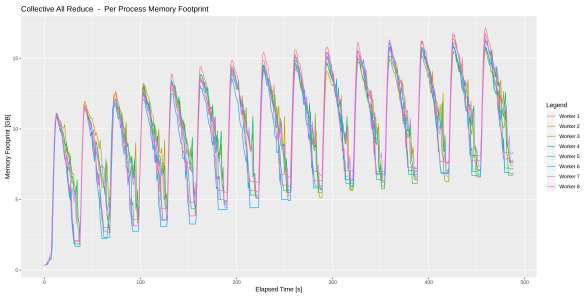
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.18: Each plot presents the per process CPU consumption in an InfiniBand medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 8 node cluster.

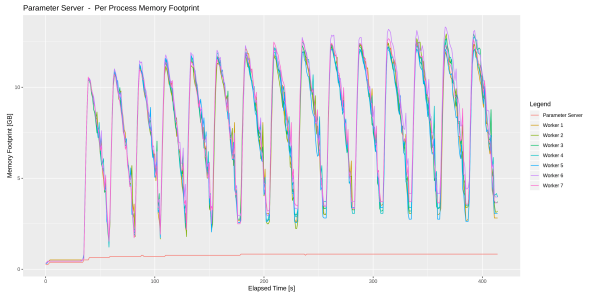
Memory Footprint



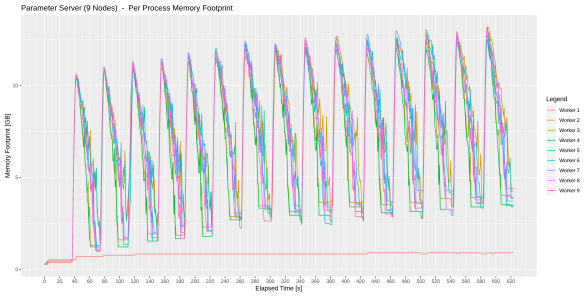
(a) Caffe2



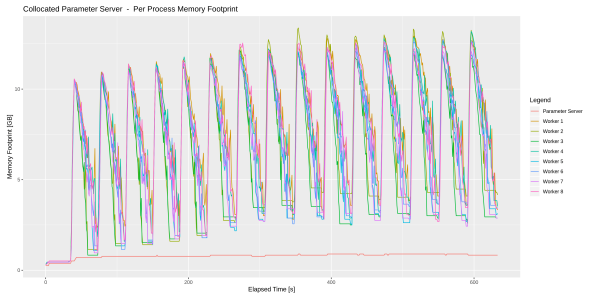
(b) Collective Allreduce



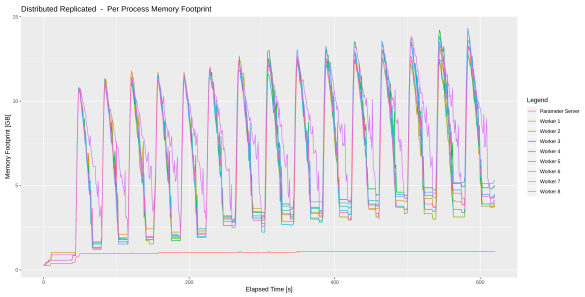
(c) Non-collocated Parameter Server



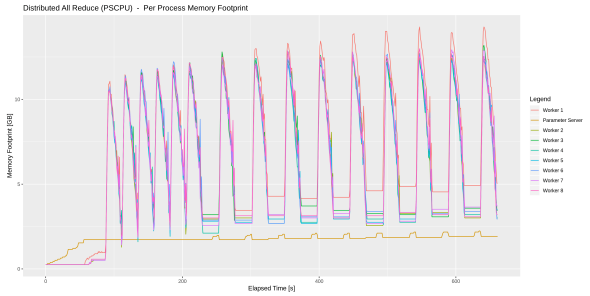
(d) Non-collocated Parameter Server with extra worker



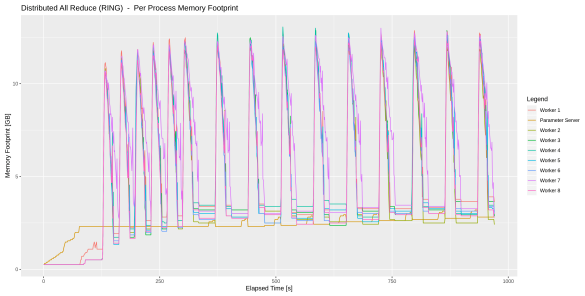
(e) Collocated Parameter Server



(f) Distributed Replicated



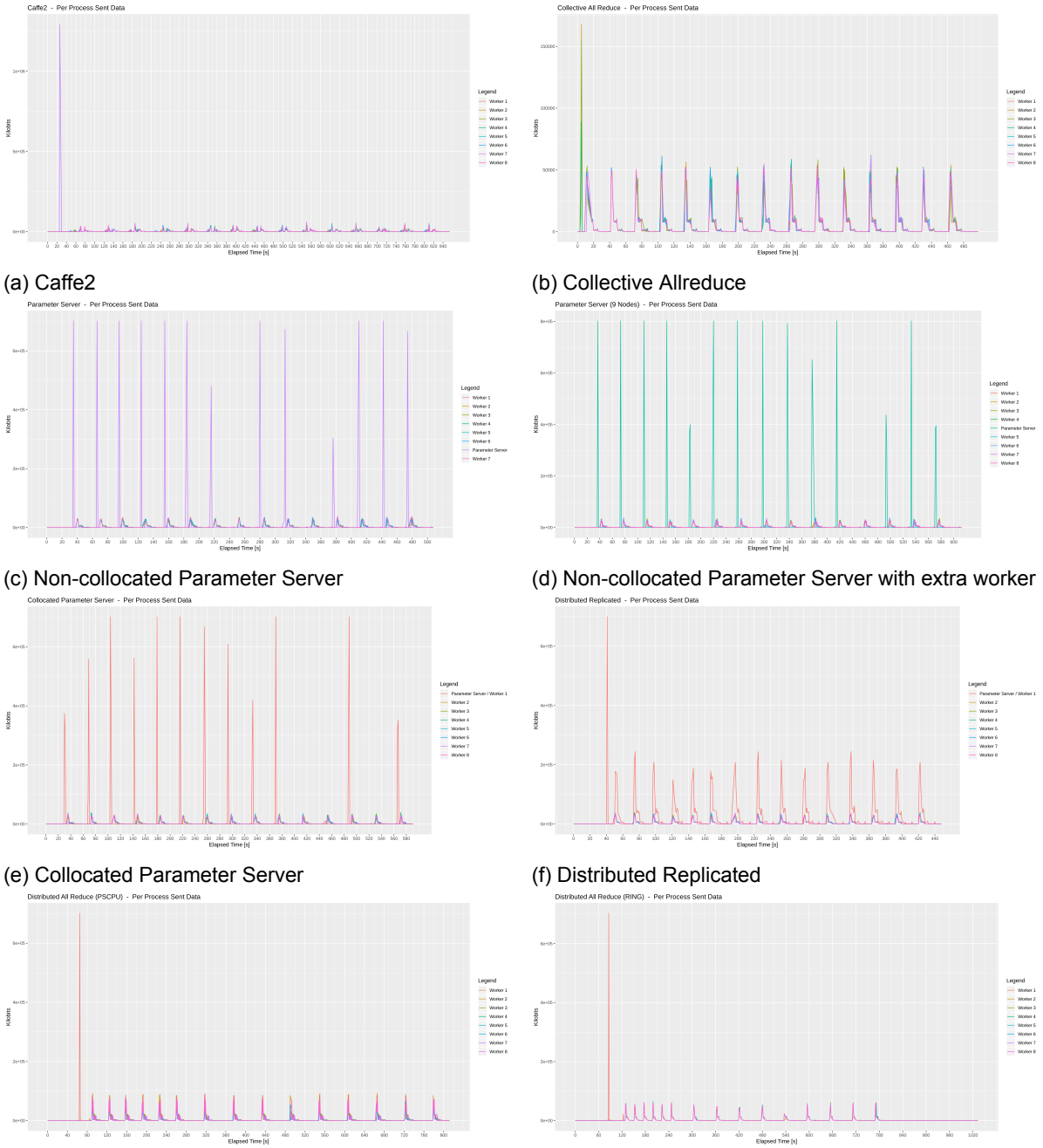
(g) Distributed Allreduce with PSCPU reduction strategy



(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.19: Each plot presents the per process Memory Footprint in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 8 node cluster.

Network Traffic



(a) Caffe2

(b) Collective Allreduce

(c) Non-collocated Parameter Server

(d) Non-collocated Parameter Server with extra worker

(e) Collocated Parameter Server

(f) Distributed Replicated

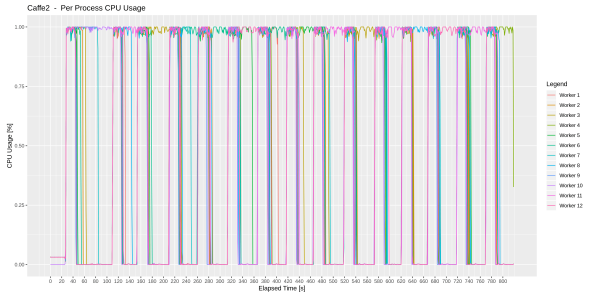
(g) Distributed Allreduce with PSCPU reduction strategy

(h) Distributed Allreduce with Ringx1 reduction strategy

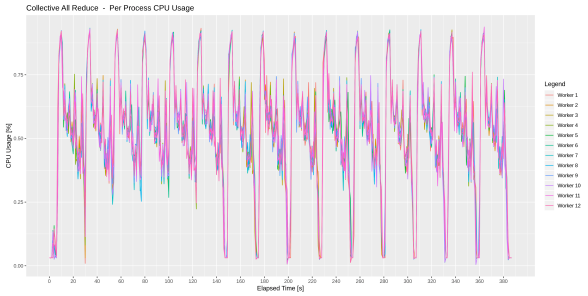
Figure C.20: Each plot presents the per process outgoing Network Traffic in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 8 node cluster.

12 Nodes

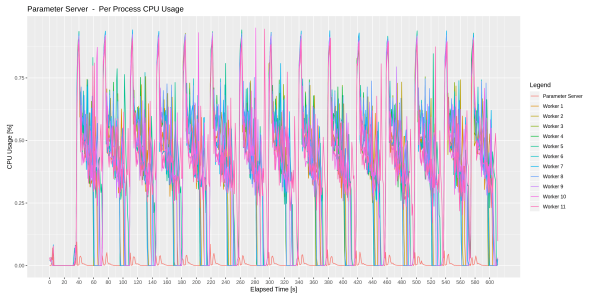
CPU Utilization



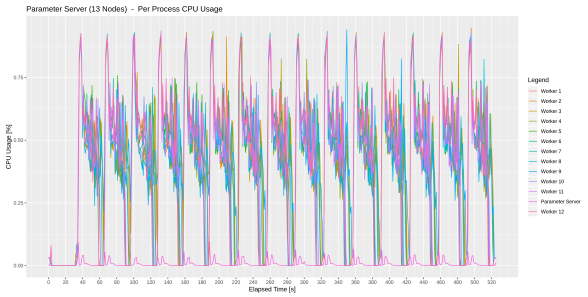
(a) Caffe2



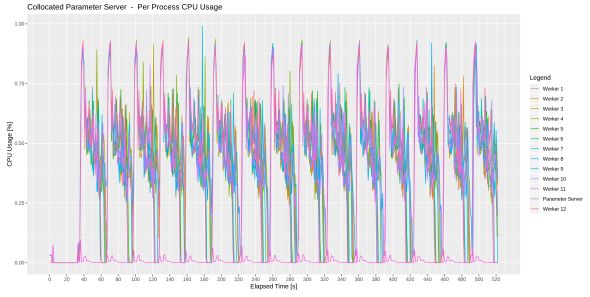
(b) Collective Allreduce



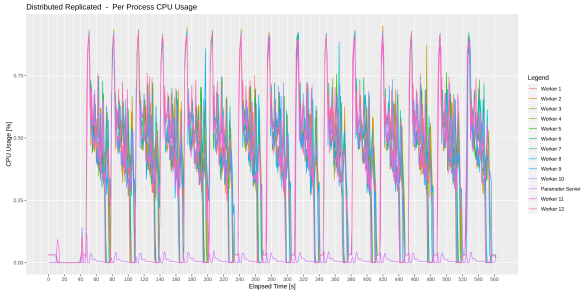
(c) Non-collocated Parameter Server



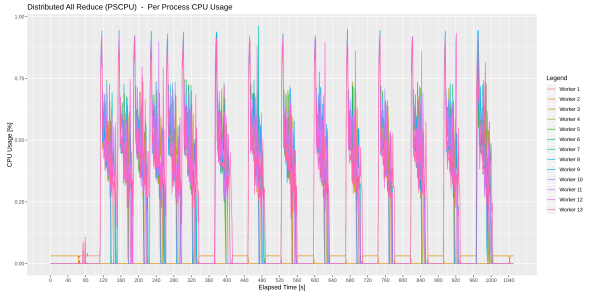
(d) Non-collocated Parameter Server with extra worker



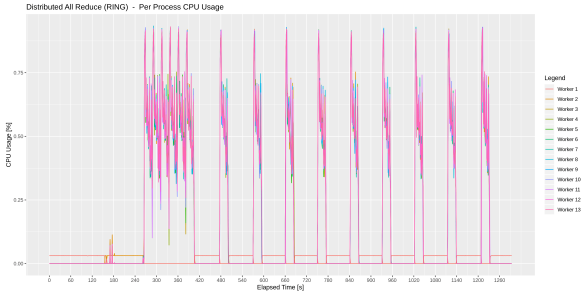
(e) Collocated Parameter Server



(f) Distributed Replicated



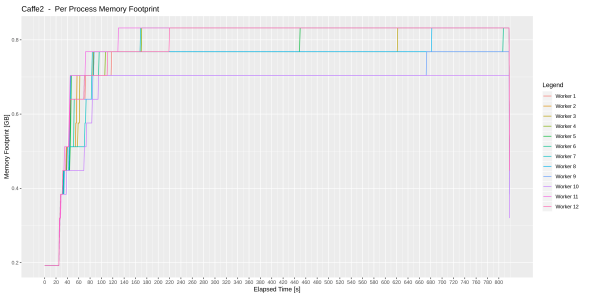
(g) Distributed Allreduce with PSCPU reduction strategy



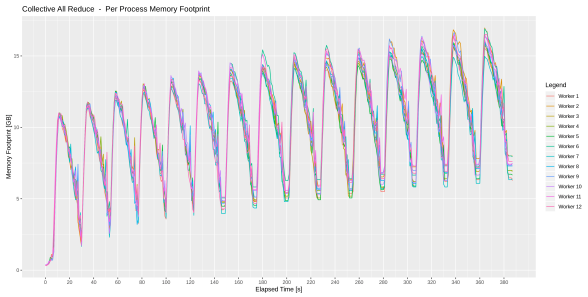
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.21: Each plot presents the per process CPU consumption in an InfiniBand medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 12 node cluster.

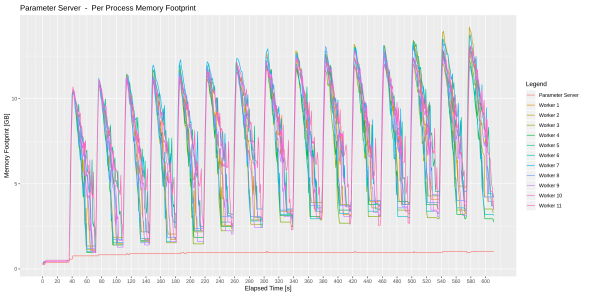
Memory Footprint



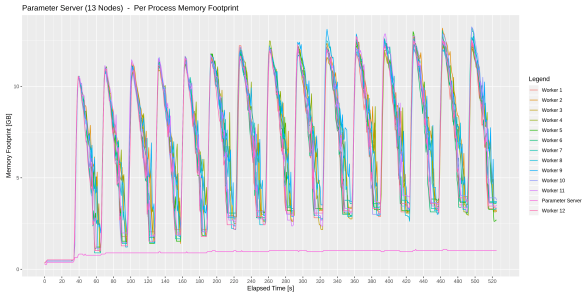
(a) Caffe2



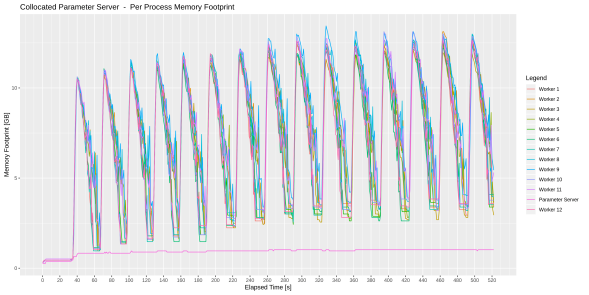
(b) Collective Allreduce



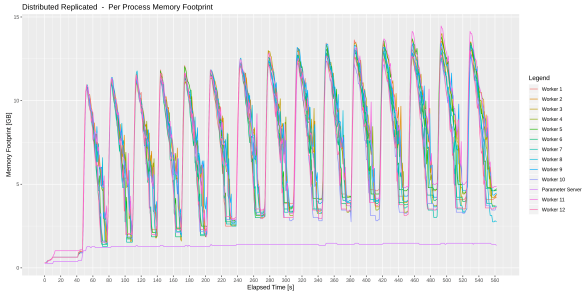
(c) Non-collocated Parameter Server



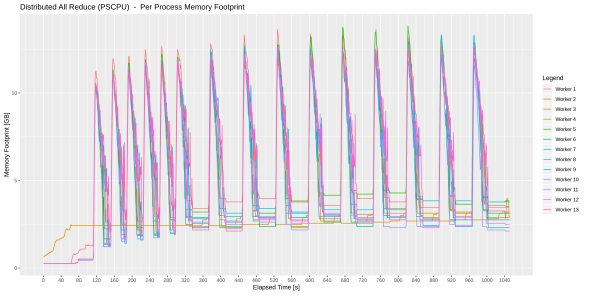
(d) Non-collocated Parameter Server with extra worker



(e) Collocated Parameter Server



(f) Distributed Replicated

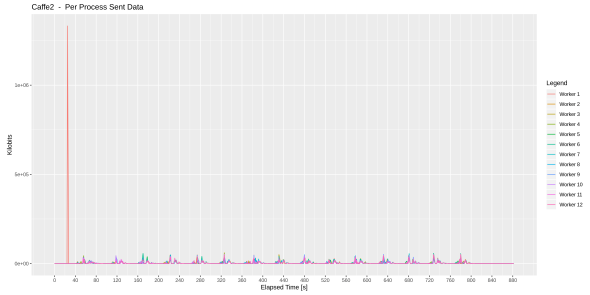


(g) Distributed Allreduce with PSCPU reduction strategy

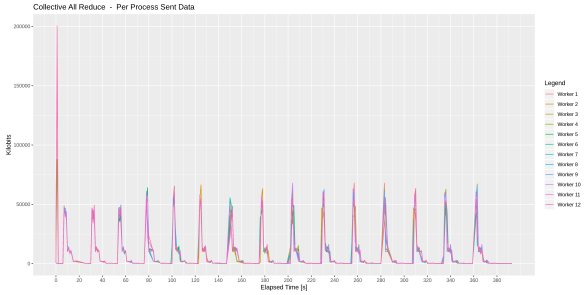
(h) Distributed Allreduce with Ringx1 reduction strategy

Figure C.22: Each plot presents the per process Memory Footprint in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 12 node cluster.

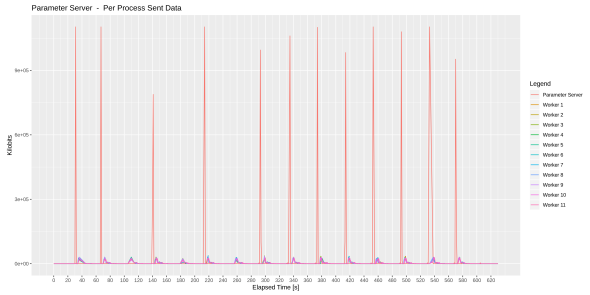
Network Traffic



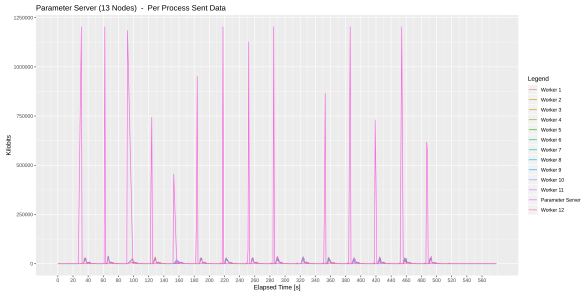
(a) Caffe2



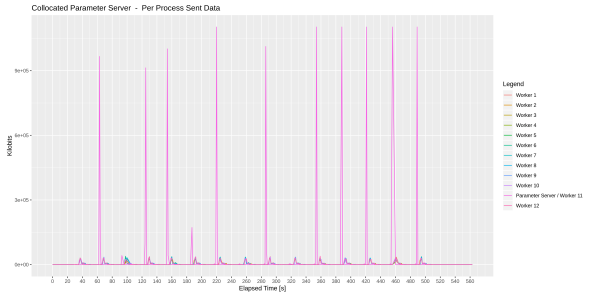
(b) Collective Allreduce



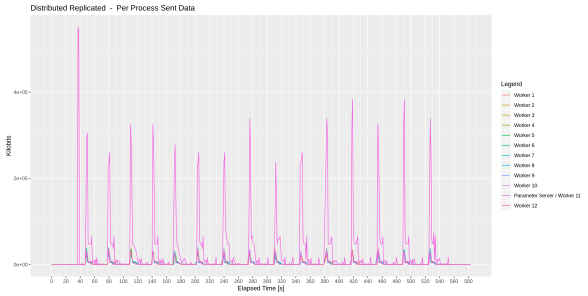
(c) Non-collocated Parameter Server



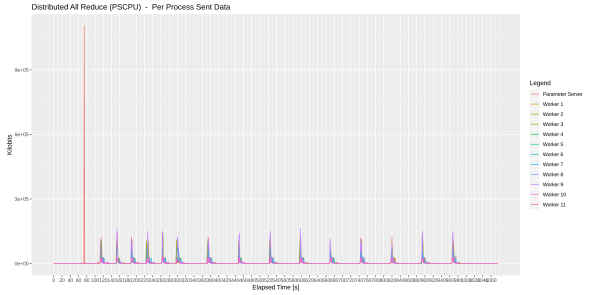
(d) Non-collocated Parameter Server with extra worker



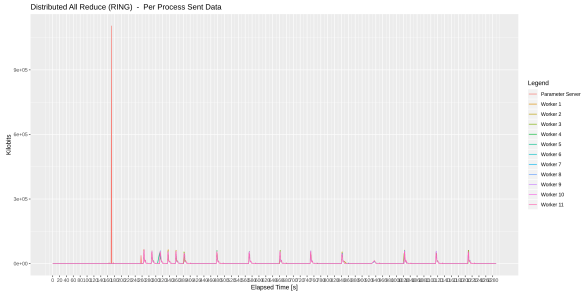
(e) Collocated Parameter Server



(f) Distributed Replicated



(g) Distributed Allreduce with PSCPU reduction strategy

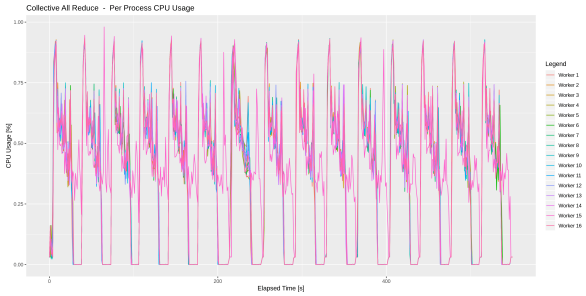


(h) Distributed Allreduce with Ringx1 reduction strategy

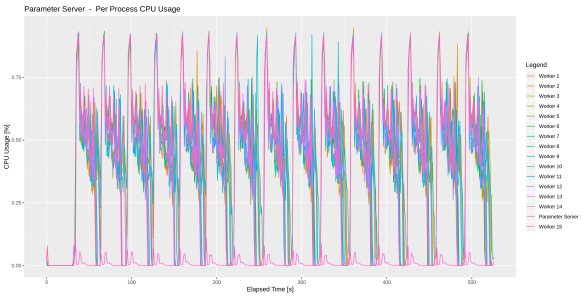
Figure C.23: Each plot presents the per process outgoing Network Traffic in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 12 node cluster.

16 Nodes

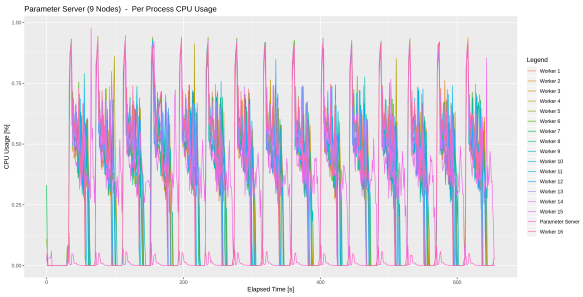
CPU Utilization



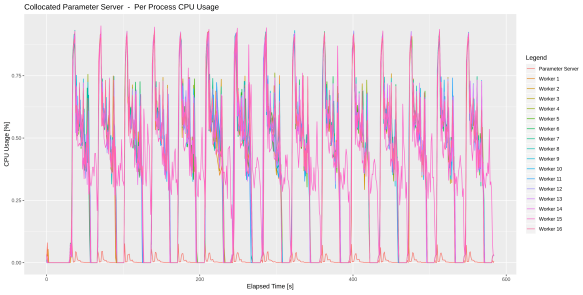
(a) Collective Allreduce



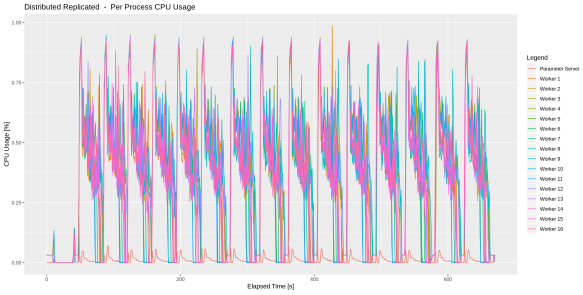
(b) Non-collocated Parameter Server



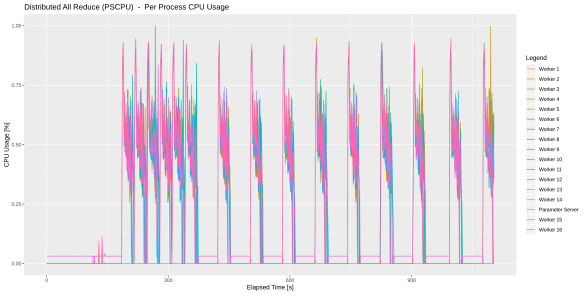
(c) Non-collocated Parameter Server with extra worker



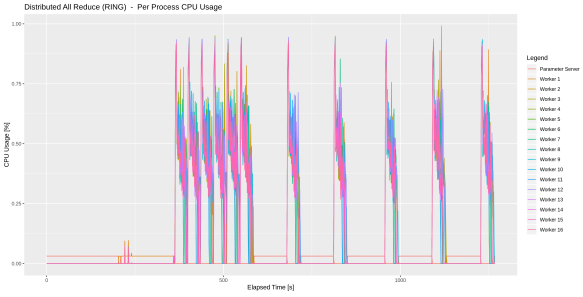
(d) Collocated Parameter Server



(e) Distributed Replicated



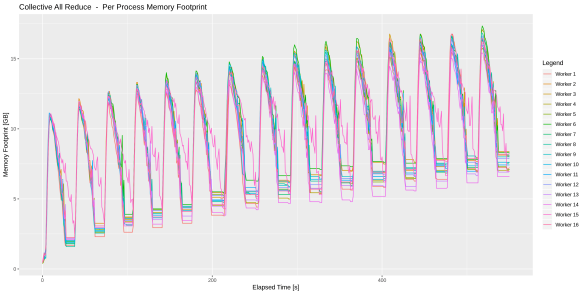
(f) Distributed Allreduce with PSCPU reduction strategy



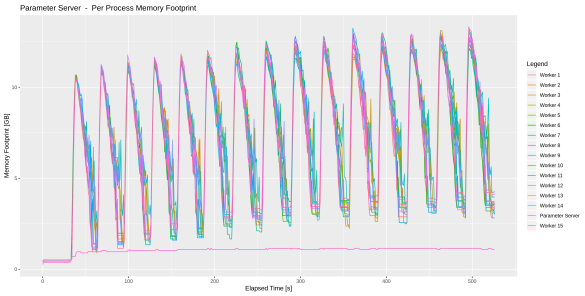
(g) Distributed Allreduce with Ringx1 reduction strategy

Figure C.24: Each plot presents the per process CPU consumption in an InfiniBand medium, as a set of time series, one for each process. The x axis is unnormalized across plots. 16 node cluster.

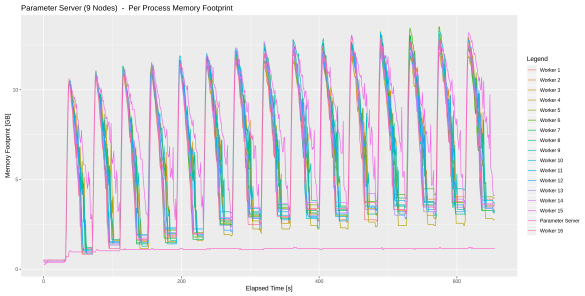
Memory Footprint



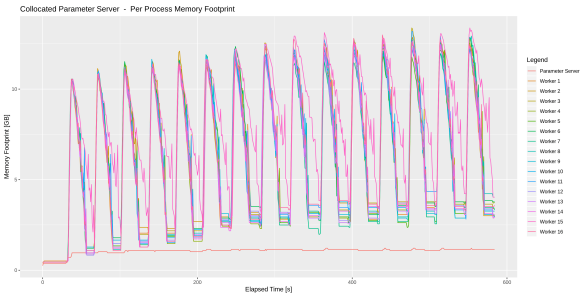
(a) Collective Allreduce



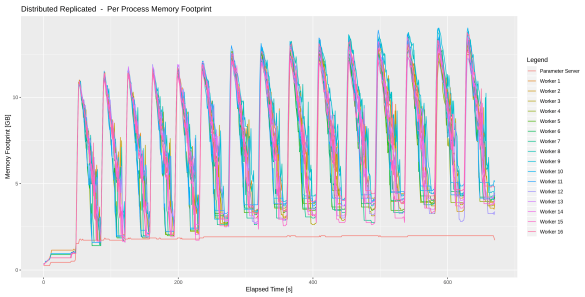
(b) Non-collocated Parameter Server



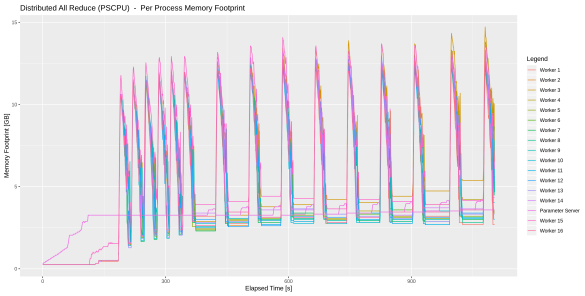
(c) Non-collocated Parameter Server with extra worker



(d) Collocated Parameter Server



(e) Distributed Replicated

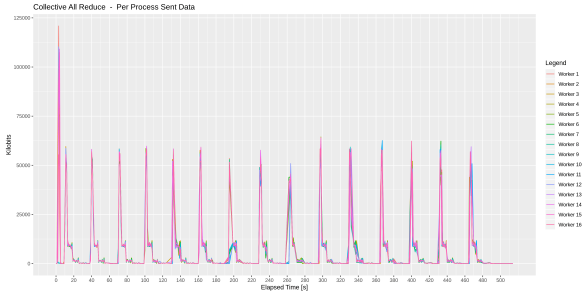


(f) Distributed Allreduce with PSCPU reduction strategy

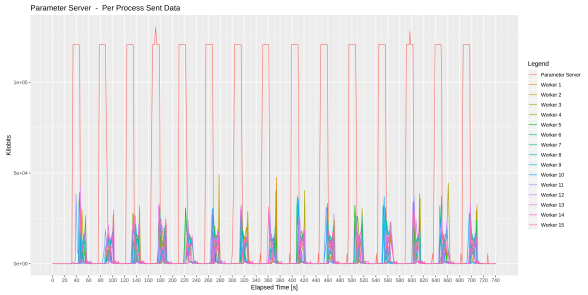
(g) Distributed Allreduce with Ringx1 reduction strategy

Figure C.25: Each plot presents the per process Memory Footprint in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 16 node cluster.

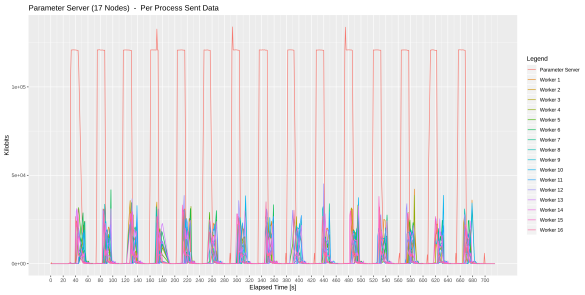
Network Traffic



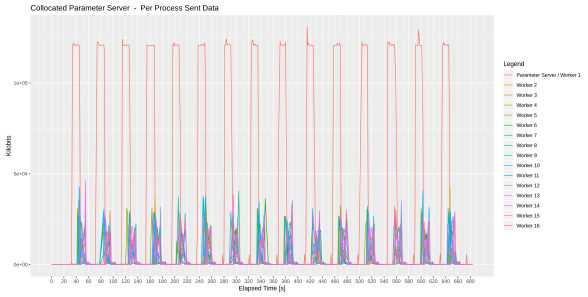
(a) Collective Allreduce



(b) Non-collocated Parameter Server

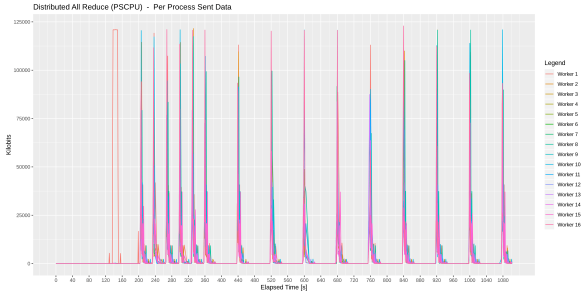


(c) Non-collocated Parameter Server with extra worker

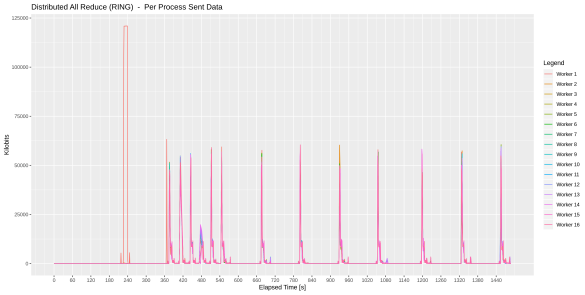


(d) Collocated Parameter Server

(e) Distributed Replicated



(f) Distributed Allreduce with PSCPU reduction strategy



(g) Distributed Allreduce with Ringx1 reduction strategy

Figure C.26: Each plot presents the per process outgoing Network Traffic in an InfiniBand medium, as a set of time series, one for each process. Both the x and y axes are unnormalized across plots. 16 node cluster.