Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 305

Systems Group, Department of Computer Science, ETH Zurich

Making DDNNs Friendlier In The Data Center

by
Hanjing Gao

Supervised by
Prof. Ankit Singla, Prof. Laurent Vanbever, Simon Kassing, Vojislav Dukic

March 2020–September 2020

inf | Informatik
Computer Science

Dedicated to my mom who (un)fortunately got stuck with me during the COVID-19 pandemic and voluntarily fed me every day.

Abstract

Distributed Deep Neural Network (DDNN) applications are seeing an increasing footprint in modern data center environments in recent years, made possible by evermore powerful GPUs and efficient distributed learning frameworks. This distributed training process also increases bandwidth requirements between nodes for transmitting model weights, however past works on communication scheduling of distributed deep neural networks focus on accelerating the training process in isolation, without considering the variability from a shared environment and the potential stress placed on other applications. This thesis explores the impact of running DDNN applications in a modern data center setting and presents a set of network prioritization schemes that can be applied to configurable distributed neural network applications, benefiting latency-critical applications by around 50 percent competing for the shared network resources without causing any degradation to the DDNN application itself and essentially making it a friendlier tenant in the data center.

Contents

Chapter 1

Introduction

In recent years, deep neural learning applications have tremendously advanced the state-of-the-art in the fields of image recognition, natural language processing, translation, game AI and more. The range of applications is growing as the adoption barrier of GPU training has been lowered through public clouds offering of a plethora of GPU configurations and flexible pricing. For example, AWS offers four instances of different sizes, all armed with the state-of-the-art NVIDIA GPU v100 Tensor Cores that aims to push the boundary of scalability and performance of DNN training [8].

The exponential growth rate of the training datasets and limited compute power and storage per GPU node have heightened the need to scale out beyond a single server, as can be seen in Fig 3 Training with Single vs. Multiple Nodes [9]. This gave rise to a new architecture: distributed training where a group of servers, each equipped with multiple GPUs, work in parallel to accelerate the training process. Previous studies on distributed deep neural networks (DDNNs) have mostly focused on extending existing ML frameworks with various backend primitives to enable easy-to-use cross-machine communications (tensorflow and pytorch RPC procedures), developing new parallel programming libraries specialized for DNN (cuDNN or MKL-DNN), different network technologies such as infiniband to further reduce the communication latency, and replacing TCP with RDMA to speed up individual message delivery. Other research topics on DDNN offer creative ways to reduce communication overhead directly from the application layer, ranging from reduced precision training to relaxing the synchronization restriction by introducing bounded staleness of gradient updates that can greatly benefit larger clusters and heterogeneous environments [23].

A new trend in accelerating DDNN explores communication scheduling by altering the transmission order of different DNN layers, in order to better overlap computation and network communication time [40, 26, 20]. However, previous research has failed to study the impact of this distributed DNN communication in a multi-tenant cloud or data center environment. This research makes the simplification that DNN applications are given the full network to utilize. On the other hand, research focusing on optimizing general coflow/flow scheduling in data centers [5, 48, 42, 24, 51, 3] are indiscriminate to the traffic being scheduled, missing the opportunity to take advantage of the known characteristic of certain

workload-DDNN in this case, an ever-increasing network resource consumer in data centers. This thesis fills the gap in previous research by investigating the network communication nature of DDNN models and how it can be used to influence network scheduling decisions in a dynamic multi-tenant data center environment.

Specifically the contributions of the thesis include:

- Characterization and exploration of network communications involved in DDNNs, including network slack and data dependency timing.
- Implementation of a model to express DDNN behavior from a network perspective while abstracting computation.
- Implementation of an event-driven simulator in Python to model a DDNN's training process in a distributed RingAllreduce paradigm.
- Implementation of a DDNN simulator in ns-3 for fine-grained analysis of a DDNN's impact on other data center flows with different prioritization schemes at various link utilizations.
- Identification of the network conditions (traffic mix, link utilization) under which various levels of performance gains can be observed with different DDNN compute-to-network ratios.

For network engineers, this thesis bridges the gap of ML applications and the underlying network stack by abstracting away the ML frameworks and instead only exposing the necessary model parameters that matter in designing, configuring and optimizing their networks. For ML engineers, it also enables them to explore their action and design decision in tuning the model size and communication scheme to reduce impact on the other applications sharing the same network resources. This potentially allows network and ML engineers to expand the number of machines they can use to train, or allow other non-ML critical applications to utilize unused CPU and RAM on the beefy GPU nodes. More importantly, it allows greater flexibility in adapting the scheduling algorithm to varied DDNN models in ever-changing data center network configurations. Even though the thesis's results are based on simulation runs, in adopting the practice empirically, it is expected that better network utilization can be achieved and substantial overprovisioning of network resources can be avoided, which is a prevalent problem in today's data center deployment [28].

The thesis is organized in three main sections that chronologically document the research journey. It begins with the initial investigation of profiling prevalent DNN models in popular machine learning frameworks to gather specific timing, size and layer dependency data. It then explains an event-driven simulator to characterize the network slack between the backwards gradients from the current iteration and when they are needed in the next forward pass. It concludes with the design of an event-driven simulator in the ns-3 framework that enables us to integrate DDNN-specific knowledge into optimizations at lower layers in the network stack, and the characterization of these networks on the DDNN and background flows under various conditions. Although Chapter 3 through Chapter 5 each represents different phases of the research, they are organized in

a similar fashion that start by providing brief background information followed by design and implementation details and end with analysing experimentation results.

Chapter 2

Related Work

Past researches have examined network optimization on distributed deep neural networks from various angles. Some aim to reduce the amount of data that needs to be transmitted per iteration by reducing the precision of parameters [47, 18]. Others have introduced asynchronous gradient aggregation in the hope of reducing network traffic and possible network congestion in the [23, 31]. Some offer to increase computation time [25] relative to computation time by increasing batch size. Most of this research attempts to make DNN even less network bound which can provide an orthogonal benefit to our research work, because the less network bound the DNN workload is, the more opportunity there is to yield the additional bandwidth to applications in need. The models studied in these networks are generally large, which could make them more network bound. In addition, these works are conducted a few years ago and the data center networks now have new generation of switches that are generally at least 10G at the NIC and 40G or 100G at core. Furthermore, these researches are using frameworks that rely on directed acyclic graphs that aren't inherently optimized for network transfers.

Recent works focusing on optimizing the network communications in DDNN discovered the lack of overlapping between computation and communications and have tackled it with similar approaches by reordering the network transfers based on when data is actually consumed. All of them [20, 40, 26, 21] implemented the solution with only a certain framework which doesn't work for DDNN applications running on other unsupported frameworks with the exception that ByteScheduler [40] supports multiple frameworks but not all of them. Most of the work optimizes the centralized parameter server model with the exception of ByteScheduler and Caramel [21], which support decentralized paradigm allreduce. However, none of them looked at an even more network optimized algorithm - RingAllreduce, adopted and improved further by Uber in Horovod [44]. Many of the works have also failed to remove the inter-iteration barrier whose existence would effectively reduce the network transfer budget by an amount that is equivalent to forward propagation time, which turns out to be one third of iteration time in the case of ResNet50 from our profiling results.

Another area of research optimizing for flow completion time in data center space is often in the area of transport layer design such as pfabric [7], or through

load balancing such as Hedera [3], multipath TCP [42] and DeTail [51], or via implicit rate control seen in DCTCP [5], D^2TCP [48] and HULL [6], or via explicit rate control such as D^3 [50] and PDQ [24]. All of them looked at generic flows and none of them have examined domain-specific traffic, such as DDNNs, specifically as well as its interaction with other data center applications.

Our work is inspired by all the optimizations mentioned. Specifically in the DDNN model, we implement reordering network transfers based on when data is consumed, removing the inter-iteration barrier and performing gradients aggregation via RingAllreduce with fused tensors in ns-3. All these optimizations are to make the training process less network bound so we can focus our study on the impact of running DDNNs on a shared communications network with other types of data center applications. To the best of our knowledge, this area has not been explored from either the machine learning side or flow scheduling perspective. DDNN researches focus on framework or application level optimization to reduce overall training time and have not examined the impact their performance improvements have on other applications on the same network, let alone attempting to yield traffic to be friendlier to other applications.

Our work is the first of its kind to bridge both worlds, optimizing latency-critical applications by taking advantage of the domain-specific opportunities in DDNN, an increasingly demanding tenant in today's data centers. In the future, we would like to experiment with different proposed rate control mechanisms mentioned earlier and our current work is a first exploration to show the potential with promising gains.

Chapter 3

Initial Investigation

Before implementing any simulators and optimization schemes, it is important to examine the targeted application and its variants to better scope out the project. Thus the initial investigation focuses on profiling typical DDNN models to define the network slack quantitatively.

3.1 Background

The network slack is the time interval between when the data is produced and when its consumed. There are two main steps in each iteration of a typical training process following the popular synchronous stochastic gradient decent method. The steps listed below follow a typical centralized architecture that involves one or a group of central parameter servers. section3.4 will follow up with the difference between a centralized and a decentralized setup.

1. Forward propagation: each worker processes a set of training examples (minibatch) sequentially via a set of predefined operation layers (multiply, subtract, etc) that are model-dependent. A loss value is calculated at the end of the forward propagation representing the model error to be minimized.

2. Backpropagation: each worker recursively calculates the gradients for each layer's learnable parameters and output values from the previous layer for the loss value obtained from the forward propagation. After the gradients for each parameter are calculated, they are sent synchronously or asynchronously to a parameter server. The gradients from all of the workers will be applied centrally by the parameter server to the parameter server's copy of the parameters. Each worker will then receive or actively pull the latest parameters from the parameter server. In the next iteration of forward propagation, the updated parameters will be used.

Fig 3.1 depicts a single iteration of training on a worker machine. The potential network slack is shown as the interval between when the gradients have been

transmitted to when it is consumed in the corresponding forward propagation layer in the next iteration. It is essentially the additional time budget available to transmit gradients. The purpose of the profiling is to verify and deepen our understanding of the training process by collecting corresponding timing metrics. The data collected is used to calculate the default configuration values to represent a common use case. Other configuration parameters are left flexible in the model simulation to paint a more accurate picture of the communication patterns of a wide range of models.

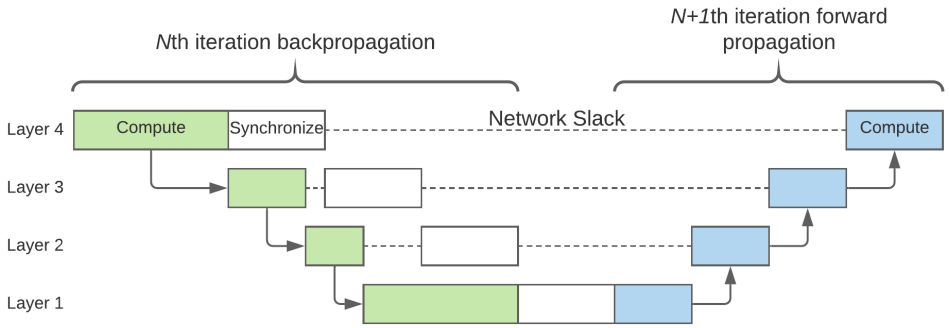

Figure 3.1: Backward and forward propagation data dependencies. Figure 3.1 shows the network slack available to synchronize the gradients from the Nth iteration's backpropagation before the updated parameters with these gradients applied are needed by the $N+1$th iteration's forward propagation. Some lower layers are on the critical path and have no or little network slack, while higher layers can have hundreds of milliseconds or more of slack to transmit their data.

3.2 Profiling Objectives

In order to model the time budget per layer, it is in our interest to gather the following metrics as a start. As we expand our simulator to support more complex architecture, for example Horovod in chapter 4 and in chapter 5, more parameters will be added. The list below represents the common set that is the core of a generic model.

- Forward propagation: computation time per layer denoted as F_i where i represents the i_{th} layer index
- Backpropagation: computation time per layer denoted as B_i where i represents the i_{th} layer index

- Size of the parameters to be transmitted per layer during backpropagation denoted as P_i
- Number of layers

In the initial investigation, we will be looking at a generic model with typical input size that runs on a commonly used GPU. The goal is to determine ballpark numbers to initialize our experiments later and also determine parameter ranges that need to be modeled in our simulator.

3.3 Framework Choice

As mentioned in the background section, past research[20, 40, 26] in accelerating distributed deep learning focuses on integrating an optimized communication scheduler that reorders network transfers of parameters and gradients updates into the state-of-the-art frameworks, such as TensorFlow[2], PyTorch[38] and MXNet[12]. All three frameworks represent the model operations using a directed acyclic graph (DAG). 3.2 shows an example of an op-level graph on TensorFlow. These operations are executed as soon as all of the input data is ready (computed or transferred) for an operation. In the case of a centralized DDNN architecture such as a parameter server setup, at the beginning of each iteration each worker will retrieve parameters from one or a set of parameter servers, however not all parameters will be used right away and instead they are consumed in the order of the underlying DAG. The delayed consumption behavior exposes opportunities for delayed or deprioritized transmission which translates to faster and/or more stable transmissions for other applications on the same shared network. The potential speedup of other tenants sharing the precious network resource in a data center setting is what inspired this thesis work. Because all the aforementioned frameworks' implementations follow such a DAG paradigm, any of them can be used for profiling purpose.

The computation time of each layer in either forward or backpropagation depends on the model size, input batch size and the underlying hardware compute units used (GPUs). The network transmission time is mostly determined by the size of the model, in other words, the size of the parameters and network bandwidth between each worker machine and a parameter server in a centralized setting or other worker machines in a decentralized setting. Thus both the computation time and the communication time are framework-agnostic which gives me of more freedom in choosing a framework that we feel most comfortably with.

In the initial investigation from someone who has little experience with any of the frameworks, we picked Pytorch for its straight-forward APIs and easy-to-understand user manuals. Next we will detail the thought process of how a particular communication architecture and an application is chosen.

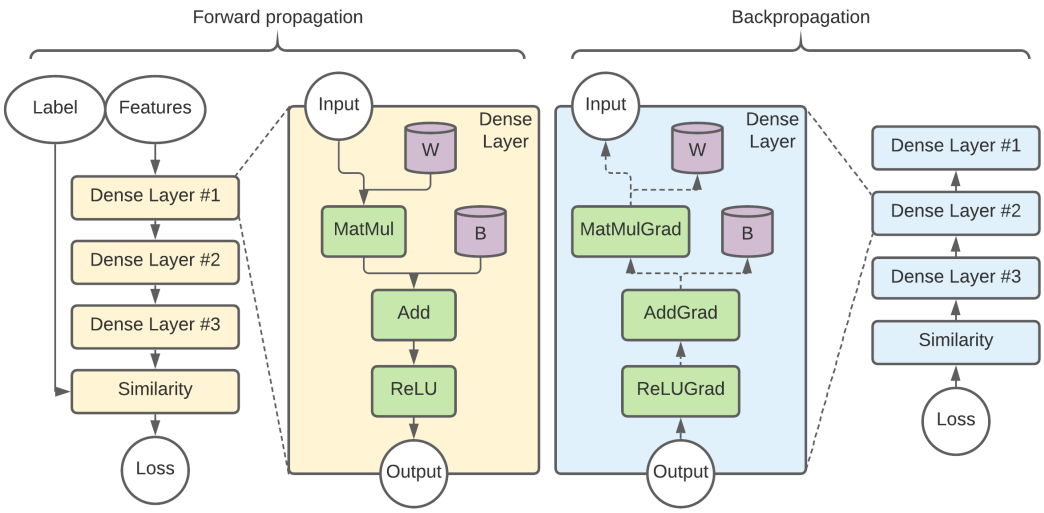

Figure 3.2: DAG graph of forward and backpropagation, showing the layer and data dependencies inside each layer.

3.4 Model Concurrency Choice

The training process of a deep neural network can be extremely time consuming due to the complex computation involved in tuning an extensive set of parameters on the order of billions, as well as processing an ever-increasing amount of training data[30]. To accelerate the process, data parallelism is often the de facto method. Training data is sharded among all the workers participating in training a DNN model.

There are many architectures when it comes to averaging and updating gradients across all workers in a distributed multi-machine deep neural network. We are most interested in two aspects that play critical parts in shaping the communication patterns - *centralization* and *model consistency*. Figure 3.3 (page 24, [9]) gives an overview of the four scenarios.

In a synchronous, centralized architecture, gradient updates from worker nodes are aggregated at a central group of parameter servers. During backpropagation, each worker computes gradients from the last layer to the first, and once the gradient is computed for a specific layer, the worker can immediately push it to the parameter server. The parameter server averages the gradients once it has received them from all the workers and applies them to

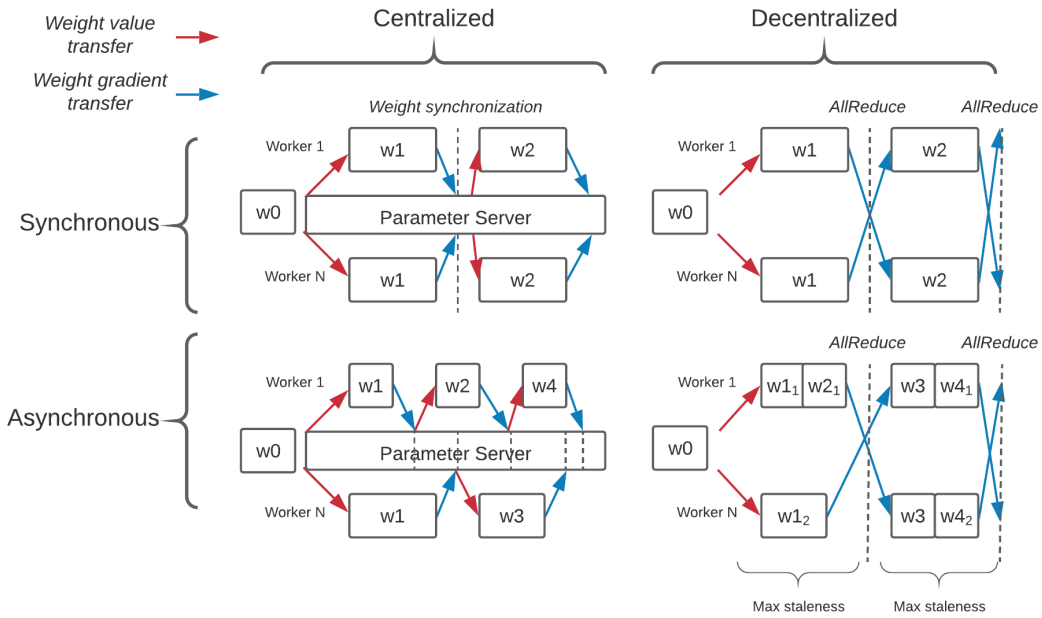

Figure 3.3: Gradient update differences between synchronous vs asynchronous and centralized vs decentralized.

the central copy of parameters. During the start of the next iteration, each agent pulls the latest layer-wise parameters to update their local weights in order to start forward propagation with the latest model. If the ratio between workers and parameter servers is not optimized to reflect the model size and network resources, incast congestion can occur at the parameter server side. Newer centralized architectures such as the sharded parameter server [14, 13] and the hierarchical parameter server [19] can help mitigate the incast problems. In addition, any worker that lags behind the computation process or experiences delays in pushing the gradients can also cause a slow stragglers problem that ends up delaying all other worker nodes.

In a synchronous, decentralized architecture the gradient updates are distributed directly to other worker nodes. Without a centralized logical point to update the gradients, each worker can send updates to one another and ring-allreduce the aggregated updates, essentially sharing the responsibility of a parameter server. Ring-allreduce, where each worker only needs to send updates to its neighbor in one direction several times to synchronize the layer-wise gradients, help reduce the amount of network transfer needed and speed up

training. This decentralized architecture is less vulnerable to network congestion as the number of workers grows, whereas the centralized architecture must keep sharding parameters across multiple servers to handle potentially hundreds of workers.

Asynchronous, centralized architecture can avoid the slow straggler problems by allowing each worker to operate using a slightly stale version of the gradients. Strong consistency has been found to not be required for distributed deep learning [9]. For example, HOGWILD [43, 31, 32], allows workers to read parameters and update gradients at any given time and is proven to converge for sparse learning networks. In decentralized model, one can also use Gossip Algorithms [11] to reduce communication where each worker communicates with a fixed number of random workers and the data will have been distributed to all the workers after a certain number of time steps. Stale-Synchronous Parallelism (SSP) [23] is often used to offer correctness guarantees in the presence of asynchrony in which a global synchronization step is enforced after a maximum of staleness is reached by an worker.

Slow straggler problems can also occur in a decentralized architecture as the computation of gradients needs to account for all workers in the network. This can be alleviated with asynchronous, decentralized models that allow each worker to perform a certain number of training steps with their own copy of the parameters before synchronizing with all workers through a ring-allreduce.

3.5 Test Setup

The goal at the initial phase is to collect the parameters listed in section 3.2 that are largely determined by input size, model size and hardware processing units, independent of the underlying synchronization and centralization. We set up a simple synchronous parameter server model for profiling and to compare our results with previous studies in this field since most of their work optimizing network performance are built upon a parameter server architecture [20, 26, 40], with ByteScheduler [40] also supporting all-reduce architecture.

3.5.1 Model Choice

The network slack is modeled in later chapters when we construct our simulator in both Python and ns3. We made a design choice to go with a decentralized architecture, specifically ring-allreduce as it is seeing a fast adoption in large data center environments [44]. The findings of this work are also transferable to centralized models as the parameter server is essentially a subset of all-reduce that reduces first then broadcasts [9]. As for various asynchronous optimizations as well as any other communication optimizations, they are orthogonal to our work and can be integrated to our simulator to further study their impact on the rest of the network.

For a similar reason, we use ResNet [22] to profile as it has been widely used as the standard benchmark in previous network studies [20, 26, 40] for easier verification later. In addition, ByteScheduler [40] found that that ResNet50

is more computation intensive and has fewer parameters comparing to other benchmarks (VGG16 [46], Transformers [49]). These characteristics of ResNet50 make it a perfect candidate for our study as models that are compute-bound can afford to yield some network bandwidth to other applications sharing the same network resources.

3.5.2 Instrumentation

In order to gather sizing and timing information on ResNet, a hook is registered on all modules during forward propagation to record the value of the weight attribute if it exists as some layers such as ReLU and max-pooling [45] do not have parameters.

The sample code below installs the prehook for each forward layer. The Tracker class stores timing, module name and size information as records and increments the layer index every time it is called. Later the results are dumped into a CSV file for further processing.

```
model = resnet50()
tracker = Tracker()
@torch.no_grad()
def install_time_hooks(m):
    def print_layer(m, _):
        size = 0
        if hasattr(m, 'weight'):
            size = m.weight.element_size()
            for dim in m.weight.size():
              size *= dim
        tracker.track([
            time.perf_counter_ns(),
            tracker.layer,
            m._get_name(),
            size])
    m.register_forward_pre_hook(print_layer)
model.apply(install_time_hooks)
```

Unfortunately there is no easy way to register a hook for backpropagation, As a workaround, we record how long the entire backpropagation takes and relied on the autograd profiler [37] from PyTorch to get block-wise compute time to divide up the total time to get layer-wise compute time. In general, we are looking for a trend of the compute time that can be modeled as the baseline and scaled up according to the overall model size and total compute time.

3.5.3 Hardware Setup

We take advantage of the free Google Colab platform [] that offers a free high-end GPU (NVIDIA P100) to run the ResNet50 model. This GPU model is slightly behind the state-of-the-art V100 from NVIDIA. When comparing results later, we will convert the performance numbers according to the performance difference. As several other factors like batch size are specific to each model and

affect the compute time per layer, we do not go to extensive lengths to collect performance metrics for multiple GPU models.

3.6 Profiling Results

Chart 3.4 shows a view of the profiling results by running ResNet50 on P100. The blue lines represent the computation time per layer during forward propagation and the red lines represent the size of the weight per layer. It can be clearly observed that the last 30 layers have the highest amount of parameters, followed by the 70 layers in the middle which is about one fifth the size and the first 80 layers bare very little parameters. This heavy tail distribution of the weights is promising for our optimization proposal as the most amount of the data to be transmitted is only needed at the very end of the next iteration during forward propagation which represents a large network slack potential as described at the beginning of the chapter.

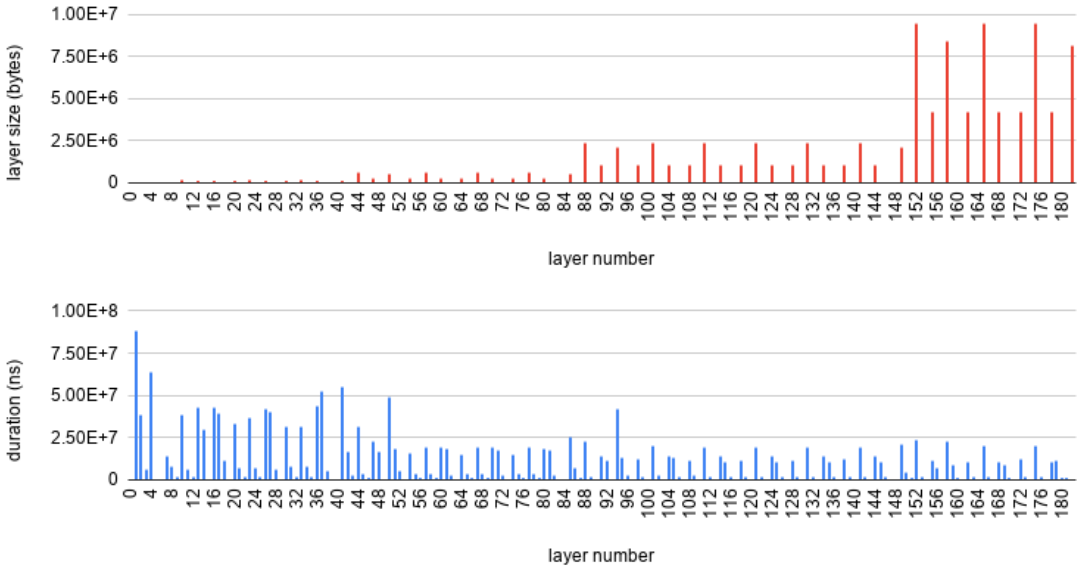

Figure 3.4: ResNet50 Forward propagation Computation Time and Layer-wise Weight Size.

Our results can also be verified with the data presented by paper [26] that displays a similar trend for per-layer parameters with larger parameters in the higher layers. As for the layer-wise computation time during forward propagation, it can be observed that the compute time decreases in training progresses into the higher layers. This trend also matches what has been found in earlier studies [29].

Table 3.1 listed the average iteration time measured during runs on P100. For comparison, we listed the benchmark results published by NVIDIA training the same model (ResNet-50 V1.5) on the same framework (PyTorch) with one Tesla T4. The iteration time for T4 is calculated using formula 3.1 as a subset of input

GPU	P100	T4
iteration_time (s)	0.85	0.90
minibatch	128	256
images/s	150	284
Max Flops (GFLOPS)	9322.46	8073.26
compute ratio	1.15	
iteration_time ratio	1.06	
test error (%)	7.9	

Table 3.1: Test results for P100 and comparison with T4

images are trained per iteration on a single node and minibatch represents the input size per node. The ratio of single-precision computation between P100 and T4 is 9322 : 8073 [] that is inversely proportional to the iteration time computed using T4 and P100. The measured iteration time is thus within 7.9% of estimation.

$$iteration\_time = \frac{minibatch}{images/s} \tag{3.1}$$

Minibatch size of 32 images / batch yields shorter iteration time and more samples per test interval vs size of 128. Figure 3.5 shows a table that contains sample outputs of the breakdown of an iteration including the time spent at each main stage (data transfer, forward propagation, backpropagation). We are most interested in the ratio between forward and backward computation time. On average, the ratio between forward propagation and backpropgation is 1:2. We are able to estimate the layer-wise computation time for backpropgation tensor operations obtained from the autograd profiler and the total computation time per backpropagation which is one third of the total iteration time.

	Event	Timestamp		Duration (s)	
2	data loading start	1586983930.52961			
3	new minibatch	0.00000			
4	data transfer start	1586983930.61763		0.08802	
5	minibatch start	1586983930.62045			
6	forwardprop done	1586983930.69011		0.06966	
7	backprop done	1586983930.82936		0.13925	
8	data loading start	1586983930.82947		0.29986	0.20891
9	new minibatch	1.00000			
10	data transfer start	1586983930.92127			
11	minibatch start	1586983930.92379		0.00252	
12	forwardprop done	1586983930.99226		0.06847	
13	backprop done	1586983931.13326		0.14100	
14	data loading start	1586983931.13336		0.21209	0.20947
15	new minibatch	2.00000			
16	data transfer start	1586983931.22646			

Figure 3.5: Breakdown of iteration time in a series of stages

3.6.1 Additional Tests

Testing on Colab has its limitations. For example, GPU offerings are random which makes it more difficult to compare between different runs. It is a great platform to gather realistic training data on single GPU node but it doesn't suit our need for testing multi-machine network conditions. To simulate a multi-machine, multi-GPU environment we use Docker containers locally on CPUs. Having worker node each reside in a docker container allows us to treat them as independent GPU server with dedicated network interface and we could use Linux TC tool to configure the bandwidth between nodes or parameter servers to simulate different network conditions. The challenge lies in mapping the the computation from GPU to CPU. We attempted to replace with key tensor operations with sleep statements in ResNet's PyTorch implementation. The sleep duration per layer is decided by the computation traces collected from GPU runs. Initially a single CPU run would take 5s locally, five times longer than what's seen on GPU. After removing computation in the convolution layers, which is supposed to be 60% of total computation time according to the profiling results in paper [16], we were able to see the iteration time go down to 3s. Figure 3.6 shows that layer-wise computation time during an iteration on CPU after the removal of Conv2D operations. The forward propagation time matches our results before where the computation time decreases towards the later stages, On the other hand, backpropagation that starts from the last layer to first spends less time on compute towards lower layers closer to the end of the iteration.

Figure 3.7 is consistent with our expectation as ratio of the total computation time of forward propagation and backpropagation is roughly 1:2. Despite the success in lowering the total iteration time down to 3s from 5s, it is still more than 3 times higher than running it on GPU. We attempted to further reduce the time by removing batch normalization operations which are supposed to account for 24% of the computation of ResNet50 trained on Titan X Pascal GPU [16]. However it only yielded negligible reduction of the iteration time. Without an accurate model of computation time per layer, it is harder to quantify the network budget per layer. Furthermore, for different models, we will need to repeat similar process which is cumbersome. These challenges with the original plan prompted us to build our own simulator that can accurately model the key stages during the training process with configurable training input and network specifications.

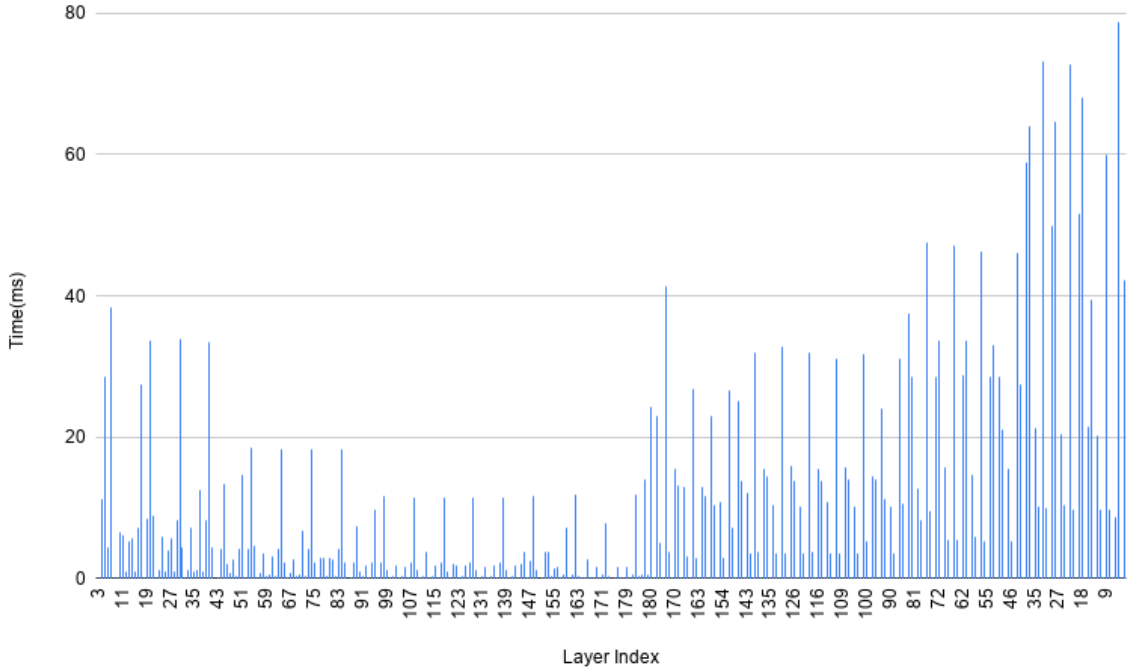

Figure 3.6: Layer-wise computation time on CPU after removing compute-heavy Conv2D operations to more mimic the timeline on GPU. The first half from layer 0 to 180 represent forward propagation and from layer 180 to 0 represent backpropagation. Overall computation time is 3 times longer than that of GPU runs and they share the same trend.

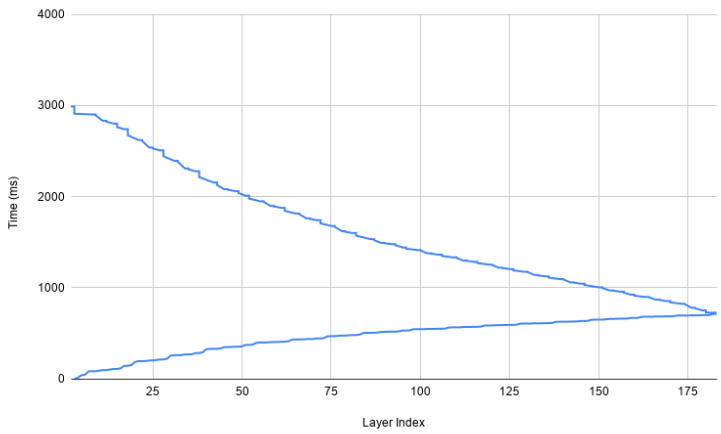

Figure 3.7: Accumulative computation time per iteration on CPU. The bottom line represents the progression of forward propagation lasted about 900 ms from the first layer to layer 180. The line corresponds to backpropagation that lasted from 900 ms to 3000ms from the last layer to the first.

Chapter 4

Event-Driven Simulator

In the previous chapter, we discussed the results of profiling ResNet-50 on a GPU to gather layer-wise statistics. We also detailed our attempt to replace compute intensive operations on CPU with timing traces collected on GPU to simulate a local GPU run with a configurable network setup using Docker containers. Because of the difficulties in hiding the computation operations embedded in PyTorch's ResNet implementation entirely, we were not able to exactly simulate a GPU run locally which can impact our goals of quantifying layer-wise network budget accurately. It also presents challenges in extending the framework to support other machine learning models or different GPU platforms as we need to repeat the profiling process each time if any of the parameters change. Because of the limitations and lack of extensibility, we decided to design and implement an event-driven simulator from scratch that models a distributed deep neural network training process with configurable network and application parameters.

4.1 Background

As discussed in Chapter 3 Section 3.4 as part of the model concurrency choices, there are two main paradigms when it comes to parameter aggregation: the centralized model commonly realized with a parameter server setup and the decentralized model where parameters are transferred and averaged collectively among all the worker machines participating in the training process. It is noted in recent studies that there is a shift from centralized parameter server method to decentralized schemes regarding parameter aggregation because of better theoretical and real-world performance [21]. Inspired by this trend, we decided to focus our effort on building a simulator that follows a decentralized paradigm which is expected to become the more dominating option of the two if the trend continues in a data center environment that is our target use case.

4.2 RingAllreduce and Tensor Fusion

The decentralized scheme relies on allreduce operations which are often implemented in Messenging Passing Interface (MPI) [1] implementations such as OpenMPI [17] or GPU-optimized communication libraries, NCCL by Nvidia[36]. These specialized libraries help set up the parallel communication infrastructure among nodes that enables collective transfers. RingAllreduce is a more network-optimal version of allreduce algorithm that was evangelized in 2017 by Baidu [15] which was based on an earlier approach by Patarasuk and Yuan in 2009 [39].

In the RingAllreduce algorithm , each worker has a different value that needs to be summed with the values of every other worker, and the combined sum needs to be transmitted to each worker. Each worker always only interacts with its two adjacent neighbors to form a ring-like communication flow. Received data from the previous worker is summed with the value in the current local buffer, and then transmitted to the next neighbor. If there are N workers in participating in the process of gradients aggregation, during the first N-1 communication steps, each worker sums the newly received data to its own data buffer and in the next N-1 steps replaces its own data with received updates. After 2 * (N-1) times, all workers will have all the same data values and updated locally.

Horovod is Uber's implementation of RingAllreduce [44]. In addition to optimizing the RingAllreduce using NCCL, communication libraries specialized for GPU-to-GPU communications, Uber engineers further optimized gradient aggregation by introducing Tensor Fusion, an algorithm to fuse tensors together before RingAllreduce is called to avoid the overhead of a large amount of tiny allreduce operations caused by small tensors.

To represent a highly network optimal DDNN training model baseline, our simulator integrates RingAllreduce and Tensor Fusion to mimic a Horovod setup.

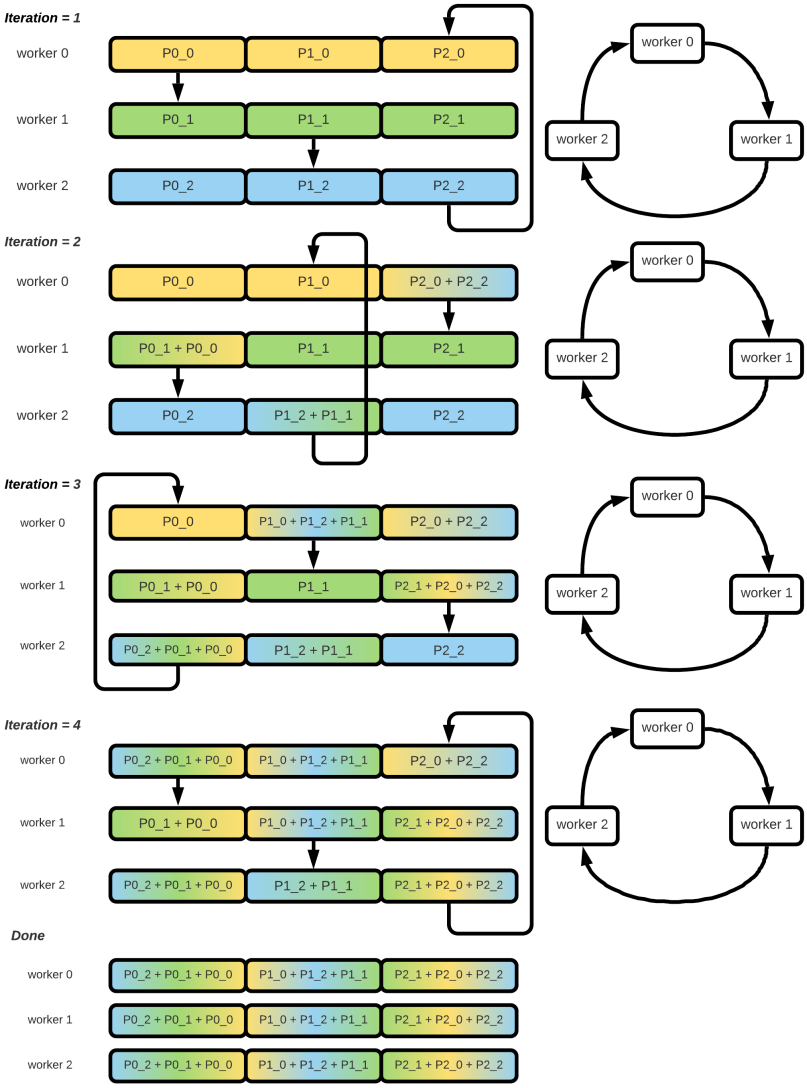

Figure 4.1: The RingAllreduce synchronization of gradients for 3 workers. 4 iterations are needed to transfer partially accumulated gradient values between the workers in a ring so all end up with the same value.

4.3 Reorder Network Transfers

Past research [40, 20, 26] has discovered the lack of overlapping between computation and communication caused by the layer-wise inter-dependencies of the underlying DAG in DDNN training. As a result, they propose to prioritise the network transfers based on when data is consumed, in this case gradients belong to earlier layers will be prioritized over those targeted for later layers.

To realize this prioritization scheme in our simulation, we assigned each fused tensor a priority that is the lowest index of all the tensors it contained. The lower the index is, the higher priority it represents, The network transmission queue is implemented using a priority queue that aims to transfer gradients targeted for the furthest layer out.

The transmission queue can also be swapped out for a FIFO queue that is to represent the non-optimized version in existing frameworks. In the results section, we will analyse both prioritization schemes and their impact on the layer-wise network budget under different network bandwidth setups.

4.3.1 Inter-Iteration Barrier

Further optimization in recent works that is crucial to expand our layer-wise network budget is to remove the inter-iteration barrier. Both TensorFlow and PyTorch enforces an iter-iteration barrier that blocks the forward propagation of next iteration from starting until all gradients have been received as seen in Figure 4.2. This essentially reduces the potential network transfer budget by an amount equivalent to the entire forward propagation computation time. By removing this barrier, we are making the training model less network-bound and increase the possibility of having extra network slack to further reduce impact on other latency-critical applications.

The simulator implemented allows users to turn on and off the barrier at initialization. We will examine in detail in Section 4.6 the impact the barrier has on the iteration time and network budget under different network conditions.

4.4 ResNet Model

We initialized our machine learning model based on the traces collected from Chapter 3 but also allow for customization of various model-specific parameters as the network optimization will be dependent on the compute-to-network ratio of each model.

Table 4.1 below shows a list of configurable parameters and their initial values. Specifically, for layer-wise computation time, we model it as an arithmetic progression for simplicity. As seen in previous profiling results, computation time decreases during later stages in forward propagation and increases in backpropagation as layer index increases. Thus the computation time per layer in forward propagation is represented using an arithmetic progression with a negative difference between successive layers and a positive delta in backpropagation. In addition, we set the ratio of total computation time between forward prop-

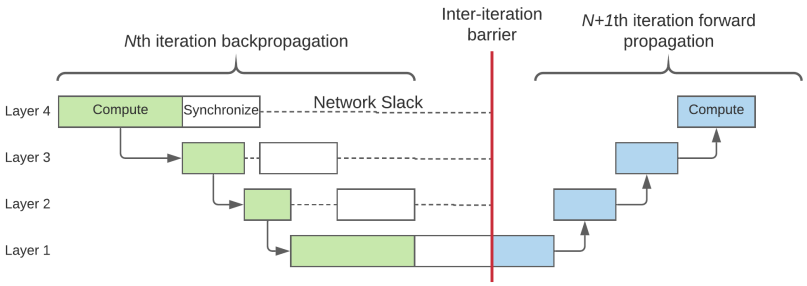

Figure 4.2: Network slack available without removing the inter-iteration barrier. There is less slack to reschedule gradient transfers because all transfers must be finished by the time layer 1 begins forward propagation.

Parameter Name	Default Value
transmission rate (Gbit/s)	10
propagation delay (ms)	0.01
number of layers	100
total iteration count	2
number of priority queues	1
qdisc	FIFO
iteration barrier	enabled
model size (MB)	100
fusion buffer (MB)	6.28
total compute time (ms)	900
number of workers	2

Table 4.1: Default values for parameters of the event-driven simulator

agation and backpropagation to 1:2 matching our previous profiling findings. Now we have everything we need to calculate layer-wise compute time in both directions. As for layer-wise parameter size, we divided up the total model size according to the parameter size distribution collected in the profiling section, shown in 3.4.

4.5 Event-Driven Simulation

The simulation is designed as a main event loop that processes events popped off from an event queue that is sorted by the timestamp of event. Each actual event in the training process is represented by two events in simulation with a start timestamp and end timestamp. The start timestamp is usually equal to the current time stamp. Recently popped events often add to the queue the

next layer's event.

At the start of the simulation, we create a series of compute events to represent forward propagation each with a timestamp equal to the sum of the current time and the computation time of that layer. We assume all parameters at the start of the simulation are updated. Thus all the events can be pushed to the event queue at the start of the first iteration. There are three main types of events that populate the event queues as shown in in Figure 4.3. F_i represents the computation of the i_{th} layer in forward propagation while B_i represents its counterpart in backpropagation. $RingAllreduce_i$ denotes the transmission of the i_{th} tensor fusion. Each tensor fusion can contain multiple tensors belong to different layers as long as the fusion is smaller than the maximum fusion size defined. Since the layer-wise tensor size is known, tensors are mapped to a RingAllreduce object at initialization. The index of RingAllreduce is determined by the lower layer index of the tensors it contains. The transmission time of a RingAllreduce is calculated as shown in equation 4.1. Each partition will be transmitted $2 * (N_{worker} - 1$ times as discussed in section 4.2. The size of the partition can be obtained by dividing the size of the RingAllreduce, denoted as S_{Ri} by the number of workers and S_{Ri} is the sum of all the tensors mapped to the i_{th} RingAllreduce. At this stage of the simulation, no network interference is introduced and we assume the simulated Horovod application is the sole tenant that can consume all the link bandwidth. Thus the transmission time is only depended on the number of workers, network bandwidth and the size of all the tensors included. It is worth noting that the transmission time is positively correlated to the number of workers.

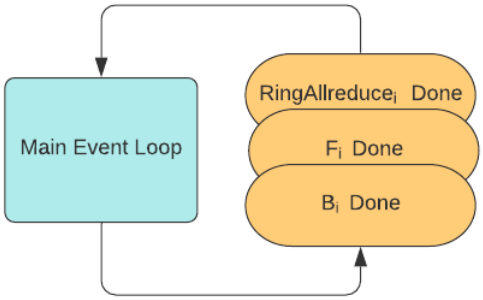

Figure 4.3: Main event loop in simulation with the three types of events

$$
\begin{aligned}
T_{R_i} &= \frac{S_{Ri}}{network\_bw} \\
&= \frac{2 * (N_{worker} - 1) * partition\_size}{network\_bw} \\
&= \frac{2 * (N_{worker} - 1) * \frac{\sum S_{ti}}{N_{worker}}}{network\_bw} \\
&= 2 * (1 - \frac{1}{N_{worker}}) * \frac{\sum S_{ti}}{network\_bw}
\end{aligned}
\tag{4.1}
$$

Figure 4.4 describes what happens when an B_i event is popped of from the queue. "B_i Done" represents computation of the i_{th} layer in backpropagation is finished. In this case, we will first check if the RingAllreduce corresponding to the i_{th} layer is ready to start if the gradients of all the tensors mapped to it are being computed. If the RingAllreduce is ready, it will be added to a transmission queue, a priority queue if it's enabled otherwise a FIFO. If nothing is being transmitted now, we will start the transmission of this RingAllreduce by adding a "RingAllreduce Done" event to the main event queue. If this is not the first layer, then the next layer of compute in backpropagation will be pushed to the event queue.

The computation in i_{th} layer in forward propagation can be triggered only if the gradients have been received for that layer and the computation in the $i_{th} - 1$ layer is finished as shown in Figure 4.5.

Once a RingAllreduce is being transmitted, we notify all layers whose gradients are contained in the RingAllreduce. If the inter-iteration barrier is enabled, we check if all the layers have received their gradient update. If they have, the iteration index will be incremented and we will push a forward propagation compute event of layer 0 to the queue. If there is no inter-iteration barrier, a computation event at layer i can be pushed to the queue as long as the previous layer $i - 1$ has finished computation as shown in Figure 4.6.

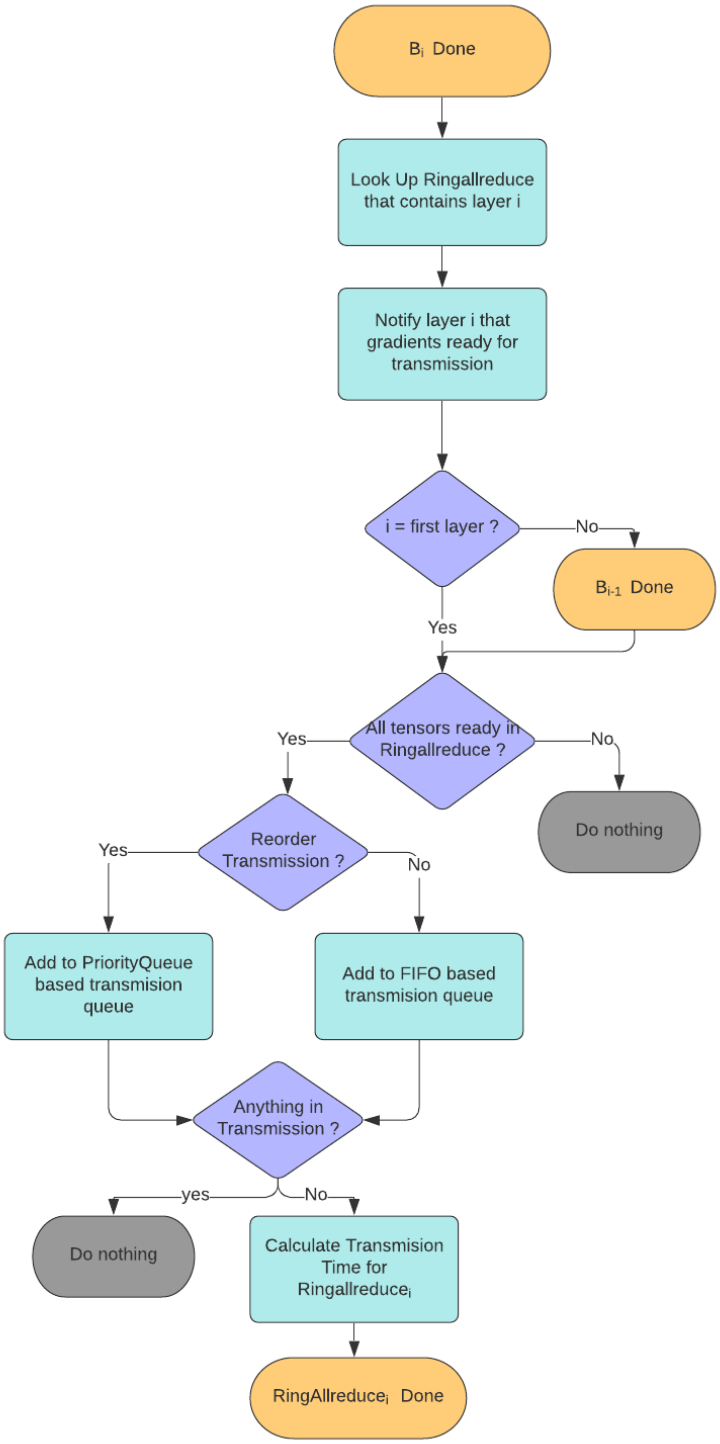

Figure 4.4: Events flow triggered by the completion of the computation of i_{th} layer in backpropagation

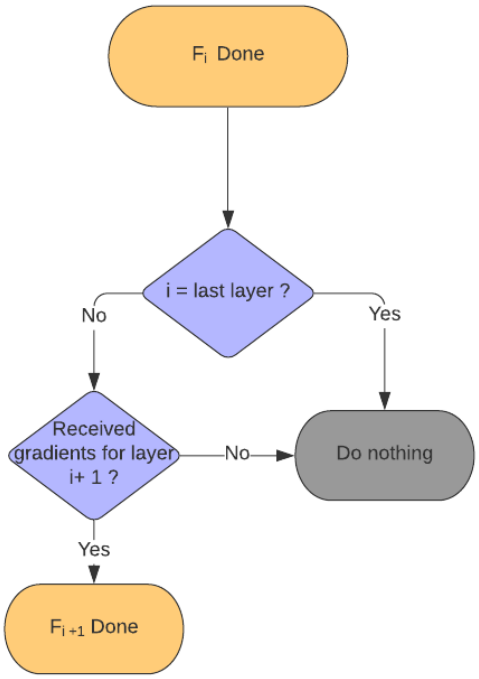

Figure 4.5: Events flow triggered by the completion of the computation of i_{th} layer in forward propagation

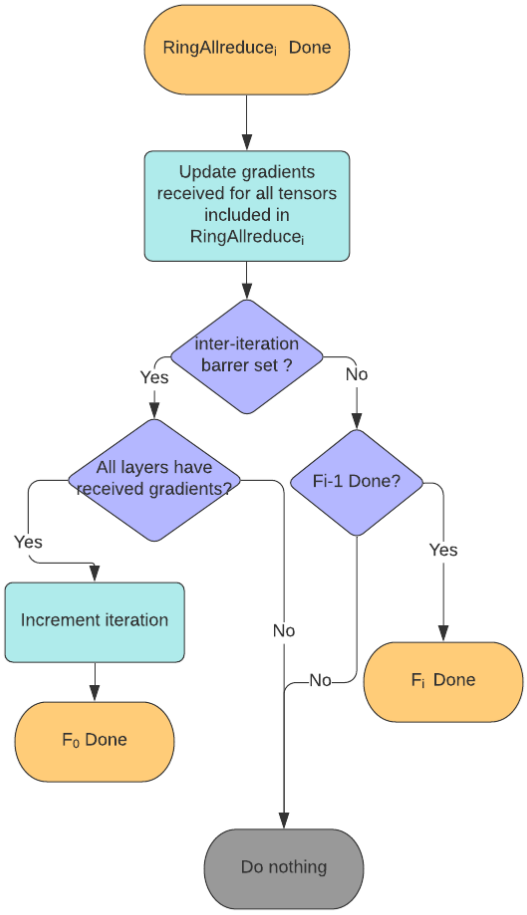

Figure 4.6: Events flow triggered by the completion of the transmission of i_{th} RingAllreduce

4.6 Results and Analysis

First we examine how iteration time changes in response to different network bandwidths with two workers and a 100MB model. This provides data on how network bound the model is, and how much slack is available at different network speeds to reorder or de-prioritize certain transmissions.

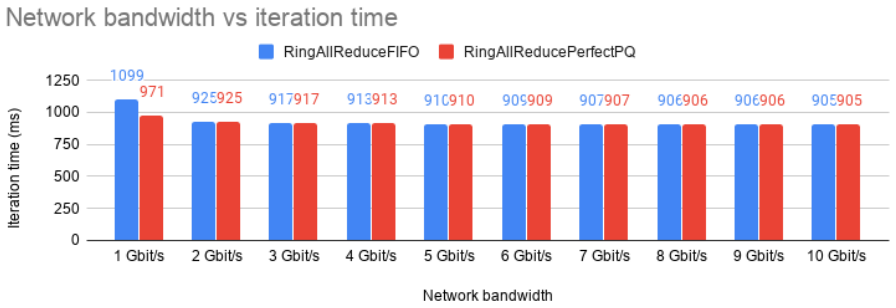

Figure 4.7: Iteration time of a FIFO based transmission vs PriorityQueue at different network bandwidth with 2 workers. At 1 Gbit/s there is a 12% reduction in iteration time when a priority queue is used to prioritize the layers whose data is needed earlier. As we increase the network bandwidth from 1 Gbit/s, there is no difference between using a FIFO queue vs priority queue. This means the model is no longer network bound if the bandwidth is equal to or higher than 2 Gbit/s.

To better visualize the gains from using a priority-based transmission queue, we plot the sequences of events during one iteration for bandwidths of 1 Gbit/s, 5 Gbit/s, 10 Gbit/s and 20 Gbit/s. As network bandwidth increases, the difference in gains between priority-based and FIFO gradient transmission queues shrinks to 0 as the model becomes compute-bound.

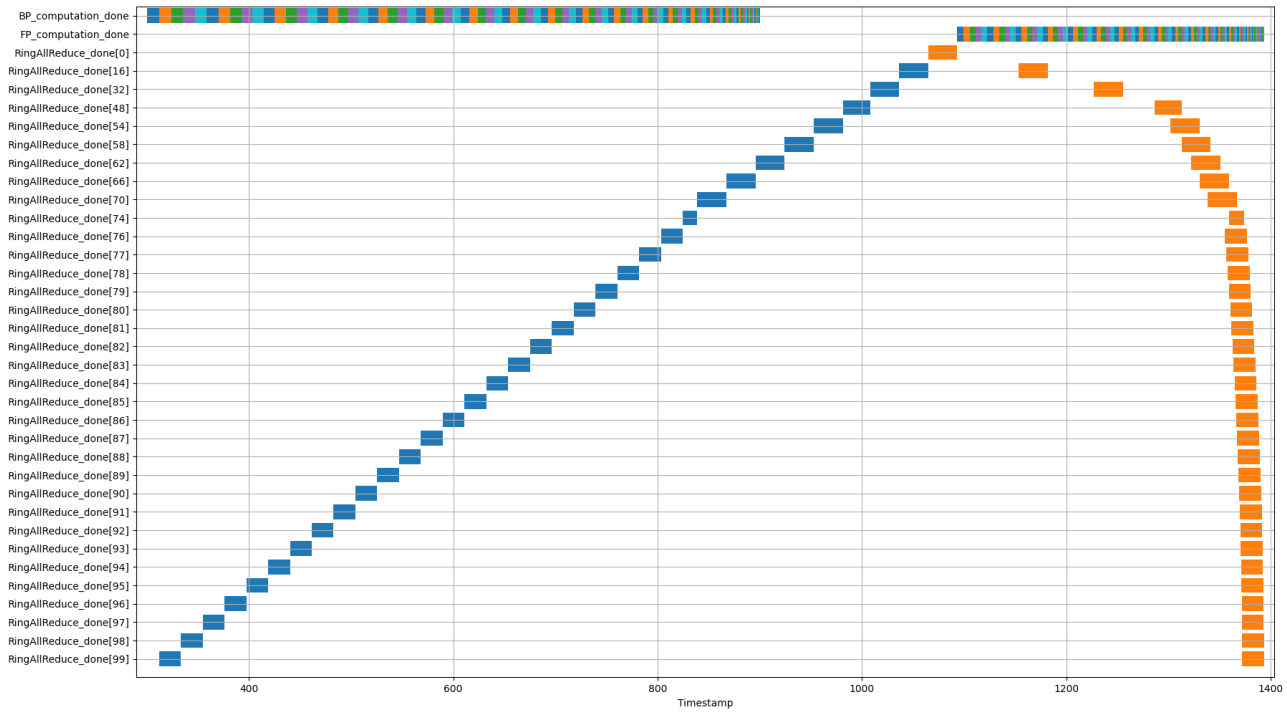

Figure 4.8: Timeline of a single iteration including computation and communication events with 1 Gbit/s network and a FIFO queue. The top two lines represent backpropagation and forward propagation respectively and the blue rectangles on the lines above denote the time period of each RingAllreduce. The transmission time can be viewed as the width of the rectangles as the x-axis is simulation timestamp in millisecond. Orange lines represent when the data that belongs to the RingAllreduce of the same priority is actually needed. The network slack time per layer is the space between the blue rectangles and orange rectangles. When the link bandwidth is at 1 Gbit/s, the model is clearly network-bound as backpropagation is completed around 900ms, but because of iteration barrier none of layers could start forward propagation until all the gradients have been transmitted and received which is not until 200 ms later at 1100 ms.

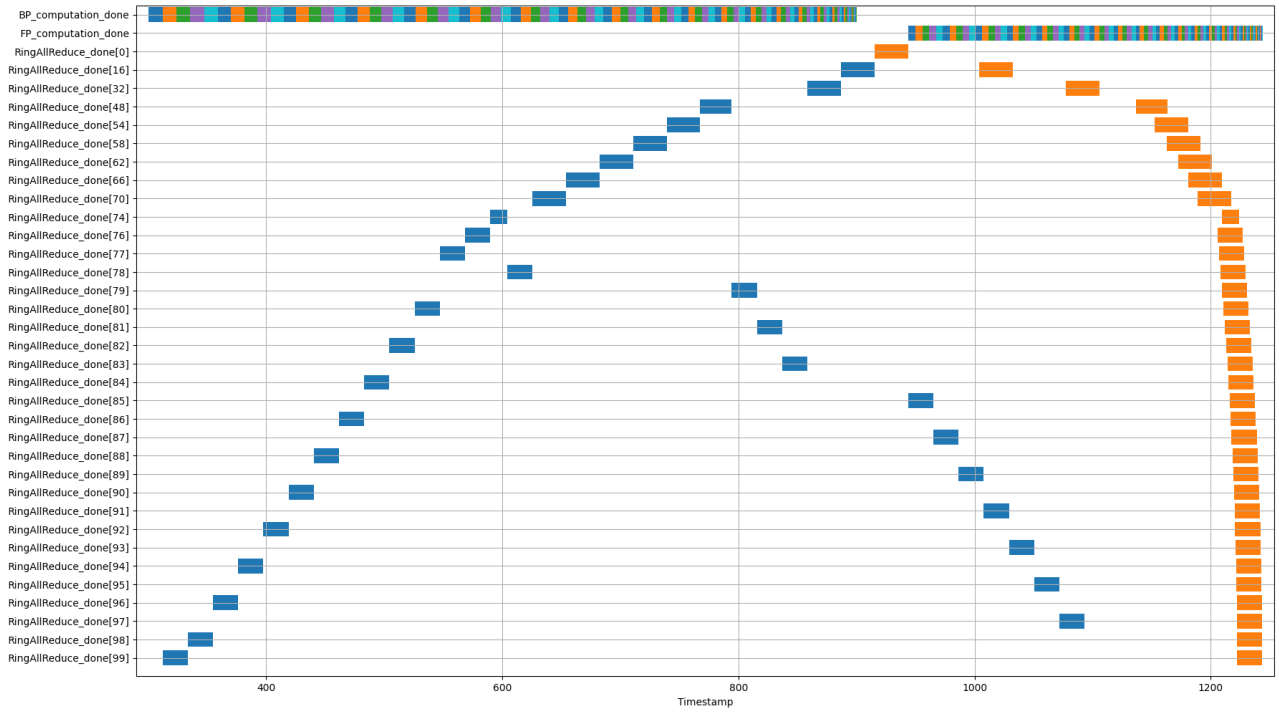

Figure 4.9: Timeline of a single iteration including computation and communication events with 1 Gbit/s network and a priority queue with removed barrier. Forward propagation is able to start as soon as the gradients belonging to layer zero have been received around 950 ms. The transmission of layer zero gradients are also prioritized to unblock the forward propagation. This is 150ms faster than the FIFO queue.

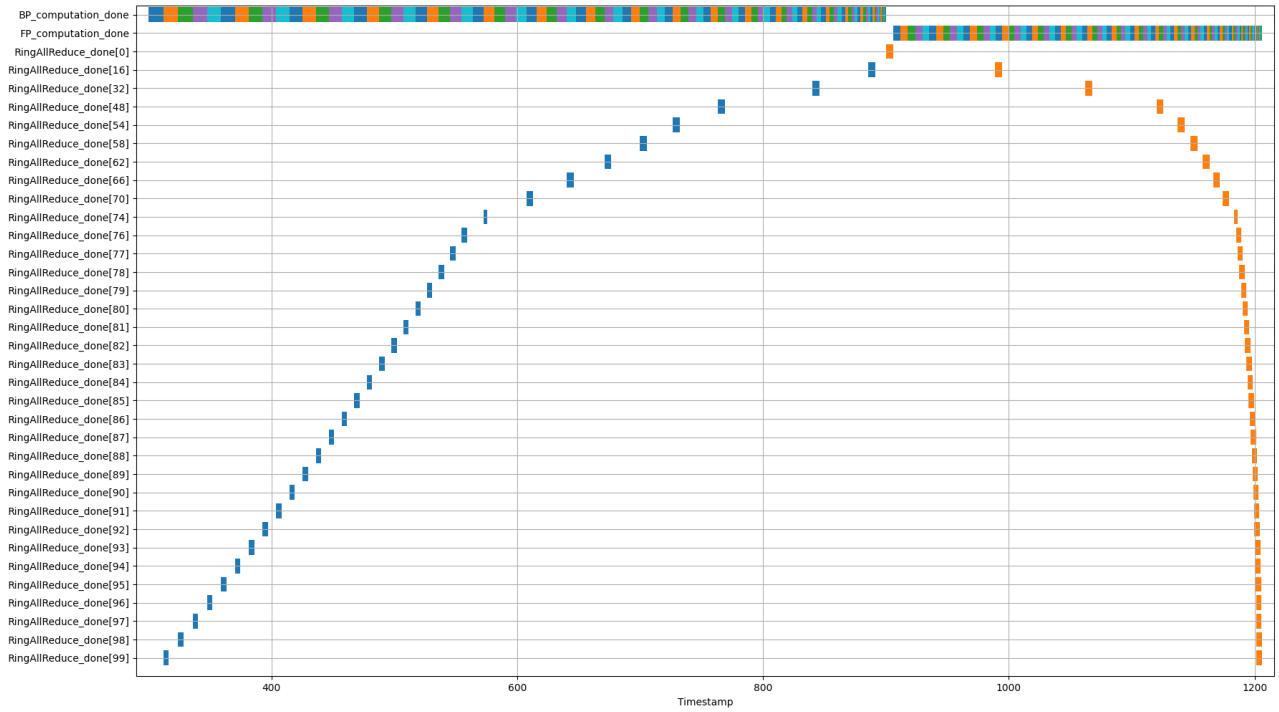

Figure 4.10: Timeline of a single iteration including computation and communication events with 5 Gbit/s and a FIFO queue. As we increase the network bandwidth to 5 Gbit/s in Figure 4.10 and Figure 4.11, each gradient transmission is completed before another one is enqueued, a priority queue in this case behaves as a FIFO queue without any reordering. The gaps between the blue rectangles and orange rectangles are enlarged which means the network slack is further increased and the model is compute-bound.

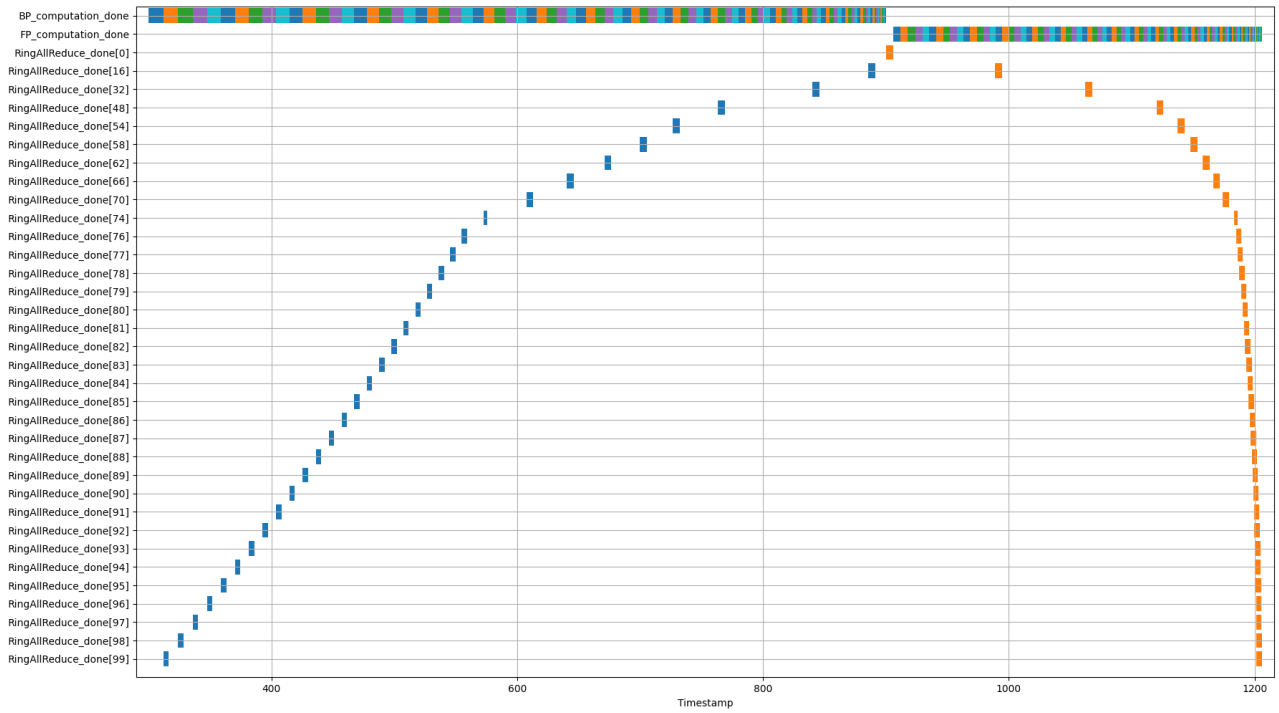

Figure 4.11: Timeline of a single iteration including computation and communication events with 5 Gbit/s network and a priority queue. No advantage between a FIFO queue and a priority queue is seen beyond 5 Gbit/s as each gradient transmission completes before the next gradient is calculated and ready to be sent.

Chapter 5

Ns-3 Model

This chapter details the design and implementation of a Horovod simulator using the ns-3 framework [33]. It is a step forward from the Python simulator described in Chapter 4 that more accurately models the RingAllreduce operation among workers. In addition, it examines how Horovod traffic with different priorities impact other flows sharing the same network.

5.1 DNN Model

The parameters of the deep neural network are initialized to simulate the popular image classification model ResNet-50. The initialization values align with our profiling findings from Chapter 3. For example, the model size is defined by the total number of parameters which are summed across all layers. Parameter-size-per-layer is defined in a user-provided text file that is read by the model later during initialization. In all our tests, the layer size distribution follows the same setup described in chapter 4 and contain the compute time per-layer for both forward propagation and backpropagation.

Table 5.1 lists the default values of the parameters used. The designed ns-3 simulator offers users the flexibility to model any of the key parameters from previous sections as well as inject and measure realistic background flows. The main program sets up the model with parameters whose values are read from configuration files that can be generated using any higher-level scripting programming language of user's choice. The abstraction of the configuration interface aims to help bridge the gap between network traffic engineers who are often more comfortable in tuning the low-level network stack parameters and designing the scheduling algorithms in C++ and machine learning data scientists who are proficient with building training models with high-level programming languages such as Python.

Note that fusion_buffer_size is initialized to be slightly more than the largest layer to make sure that all layers are encapsulated in a fusion, as the implementation does not support splitting a single layer across multiple fusion buffers.

Parameter Name	Value
layer_size_file	layer_weight_model_size_100.0MB.csv
num_workers	8
num_layers	50
fusion_buffer_size	5333329 bytes
fp_compute_time_file	fp_compute_iter_time_900.0_ms.csv
bp_compute_time_file	bp_compute_iter_time_900.0_ms.csv
horovod_initial_priority	0x10

Table 5.1: Initial Horovod Configuration

5.2 RingAllreduce Design and Implementation

Unlike the event-driven simulator implemented earlier, with ns-3 framework we can model each worker participating in RingAllreduce with a dedicated network device and a full network stack to actually send and receive bytes, more accurately mimicking network transfer on hardware devices. We can now also breakdown each RingAllreduce process to synchronous transfers of partitions in discrete iterations. This is not only a more accurate model of the actual RingAllreduce algorithm but it also allows us to examine how one delayed transfer of any partition around the ring caused by slow stragglers or background flows on the network can lead to delay for whole RingAllreduce.

5.2.1 Synchronous Transmission

In RingAllreduce, every worker aggregates parameter gradients into a fusion buffer. When a fusion buffer is ready and no other fusion buffer is being sent, the worker splits the fusion buffer into $1/num\_workers$ partitions and synchronously sends a partition to its logical right-hand neighbor in one interval. In the next interval, the workers receive a different partition sent from its left-hand neighbor and sums it with the local buffer, then sends the updated partition to it right-hand neighbor. As described in Chapter 4 Section **??**, after the first $N-1$ intervals, each worker will have one partition that is fully summed and it will take another $N-1$ intervals to pass these fully accumulated partitions in a ring so all partitions in all workers contain the same value.

Since the simulation is focusing on the network aspect of the data transfers, we can abstract away the contents of data being transferred. Thus instead of checking the actual contents to verify the completion of the data transfer at the receiver side, each worker keeps a vector of sent bytes and it's right-hand neighbor can peak at the vector when checking if they have received enough bytes that correspond to the latest partition sent. In reality, the AllReduce frameworks (MPI, NCCL, etc) notify the worker when all of a partition has been fully received.

Figure 5.1 shows how the send and receive buffers are populated in neighboring workers during discrete iterations. Worker 0 is logically at the left-hand of worker 1 and worker 1 is to the left of worker 2 (not pictured). During an RingAllreduce, worker 0 starts to send partition of size 5000 bytes (for exam-

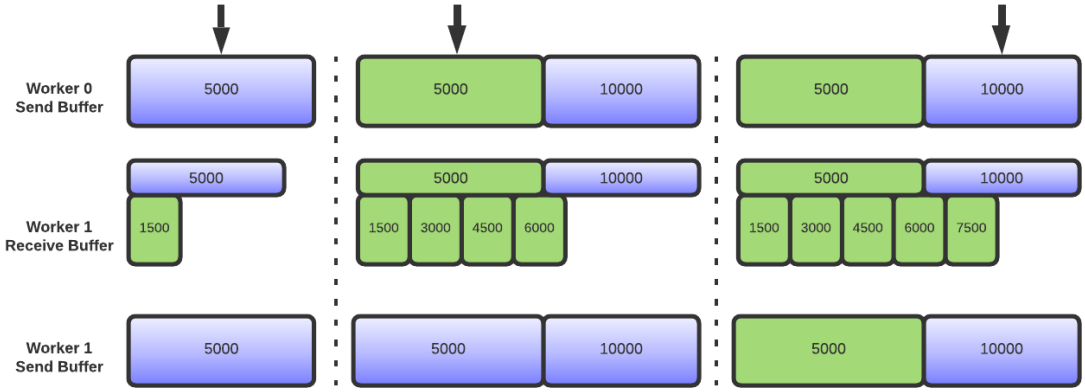

Figure 5.1: A closer look into the send and receive buffers in neighboring worker nodes in consecutive iterations. Purple blocks are in the process of sending and receiving, green blocks have been fully sent or received.

ple) to worker 1 and at the same time worker 1 sends a partition of 5000 bytes to worker 2. The interactions between worker 1 and worker 2 are similar to worker 0 and worker 1 so it is omitted from the diagram. The sending of the first partition during each RingAllreduce is triggered automatically once the RingAllreduce starts, but all subsequent transmissions are scheduled only once the sender has received a partition from its neighbor to the left. In the example, worker 1 receives data in 1500 byte chunks and has a pointer to the last partition that hasn't been received in worker 1's send buffer, denoted as an arrow in the graph. Every time worker 1 receives a data packet, it compares the accumulated received bytes with the pointer to the upstream sender's send buffer. If the accumulated bytes are greater than what the pointer is pointing to, it means that worker has received the partition and will now add the partition to its local send buffer to be transmitted to the downstream node in the ring - worker 2. The time to sum the existing and new partition is negligible and not simulated. At the same time, the pointer will be incremented to point to the next chunk in the buffer - 10000 bytes, which is the total amount of data worker 0 has sent so far. Because the bytes from different partitions aren't tagged differently, they can be grouped together in the downstream network stack and sent as one packet if the buffer is always full. This is why we check for if the total received bytes are equal to or larger than the partition being sent. In the example, the last 1500 bytes received in the 6000 chunk contains bytes from both the first partition and the second.

5.2.2 Partition Progress Status

Without inspecting the contents of received packets, we designed a mechanism to verify whether each partition in the local buffer has accounted for the values of all the worker and we want to stop sending as soon as all partitions have reached there. This is done by introducing a progress attribute to each partition

to record the number of times it has been transmitted. All partitions at the start of a RingAllreduce have progress of zero. Upon receiving a partition, a worker will look up progress of the sent partition from the sender and increment it by one locally, then send it to the next worker in the ring. After $2 * (N - 1)$ times, where N is the number of workers, all partitions will be synced. However, we are going to introduce various amounts of competing traffic to different workers, it is possible that updates to some partitions in the buffer are being delayed while the current partition is fully synced. Checking only the progress of the last received partition does not guarantee that all partitions are ready, so to avoid prematurely ending the RingAllreduce we verify the progress of all the partitions locally.

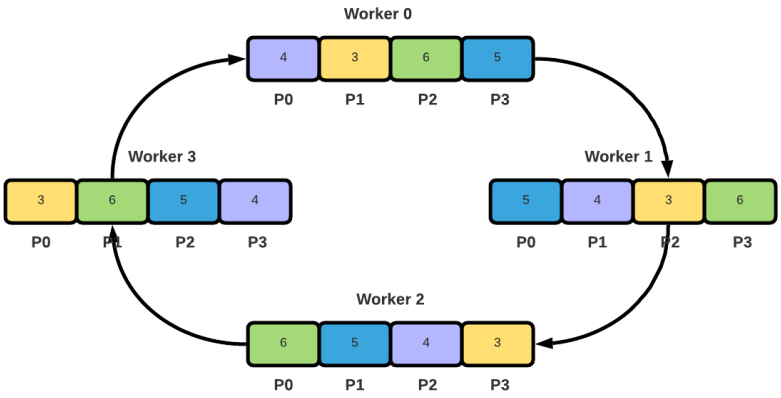

Figure 5.2: Progress value of each partition at each worker when RingAllreduce is finished

We observe that at the finish state, one partition in each worker's local buffer will have exactly a progress value of $2 * (N - 1)$ and the rest of partition's progress values will depend on their relative positions to that partition. To be exact, the progress value is decremented by one for every step we move away from the partition of the highest progress and as we reach the end of the buffer, the pattern continues as we wrap around and come to the first partition in the buffer until we have accounted for all the partitions. Figure 5.2 shows the end state with the progress value of each partition in a ring of four workers. Because there are four workers, the maximum progress a partition can reach is $2(4 - 1)$ which is 6 and each worker has exactly one partition with that progress. For worker 0, partition 2 has progress of 6 and applying the observation described earlier, we can deduce that the next partition one step from partition 2 will have a progress of $6 - 1 = 5$. As we continue and wrap around the buffer, partition 0 of worker 0 will have a progress of $5 - 1 = 4$ and partition 1 will have progress of $4 - 1 = 3$. The same applies for the rest of the workers.

In simulation, once a worker verifies that all partitions are done being accumulated it will notify a global syncer. The global syncer will then check if all workers are finished and if they are all done, it will notify them to start on the next RingAllreduce. This is done instantly in our simulation as each worker has

a pointer to the global syncer and the clean up of completed RingAllreduce is done through callbacks to each individual worker. In reality, one worker is selected to take on the responsibility of a global syncer which would incur a small amount of network traffic among the workers to exchange statuses. This is not currently modeled in our simulator as we believe the extra network traffic is negligible comparing to the gradients being sent and packets containing status information should always be prioritized.

5.3 Traffic Control

The traffic control layer in ns-3 aims to provide an equivalent infrastructure as its counterpart in Linux, located between the internet layer (IP, ICMP, etc) and the network device layer (L2) [35]. It intercepts outgoing packets coming down from the internet layer to the network device. Various queuing disciplines can be applied to the outgoing packets to control priority, capacity and dropping packets.

To examine the impacts of de-prioritized machine learning traffic in a shared network environment, we choose to use PfifoFastQueueDisc which is ns-3 version of pfifo_fast, the default queuing discipline enabled on Linux [34]. PfifoFastQueueDisc is composed of three droptail queues where packets from a higher priority band are always dequeued before packets from a lower priority band.

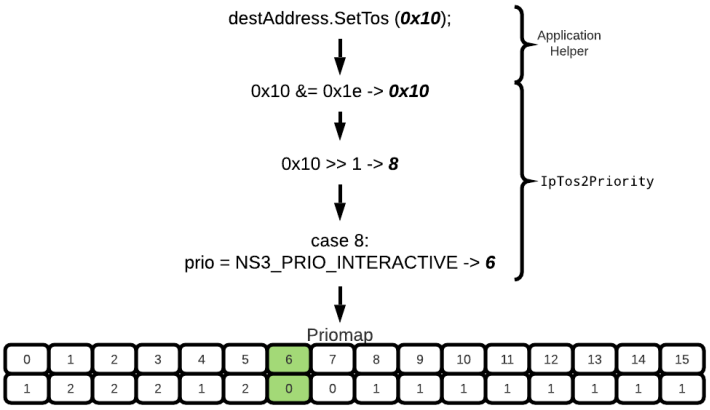

Figure 5.3: Setting send socket the highest priority band 0 from setting ToS field of the destination address to 0x10

There are three priority bands 0, 1, and 2, each corresponding to one droptail queue. A droptail queue is a FIFO queue that drops tail-end packets on overflow and the maximum number of packets each queue can hold is set to 1000 packets as default.

In ns-3, the priorities of the packets are set based on the priority of the socket that the outgoing packets are sent from. In ns-3, the priority of the socket can be set through the destination address which is usually part of an application

helper. In the application, once Socket::Connect() is called, the Type-of-Service (ToS) field will be set equal to the ToS value of the destination address. At the same time, the priority of the socket will also be calculated based on the ToS value. Last but not least, the priority value of the socket will be used as an index into the priomap that maps to a priority band that decides which priority queue the packet from this socket will be enqueued to.

In our case, we use two of the three priority bands as we assign all background flows one priority and Horovod another. Figure 5.3 shows how a high priority is set to a ToS value of 0x10 as an example.

5.4 Experiments

We would like to study the impact on the background traffic by running Horovod at different priorities. All background traffic is set at the highest priority throughout all the tests and Horovod is either set at the same priority as the background traffic or at a lower priority. The expectation is that running Horovod at lower priority will benefit the background flows in terms of reducing the flow completion time of the background traffic in certain conditions. The goal is to find out what these conditions are, to what extent the gain can be obtained, any additional optimization opportunity to create these conditions, and the impact on the training speed of Horovod.

5.4.1 Topology

Although the logical topology is ring-shaped, the physical topology is a single ToR architecture as shown in Figure 5.4, defined in the topology file used for setting up ns-3 network devices, links, utilization tracking and the rest of the network stack. Any two workers are two hops apart and the network link delay between any worker is configurable and is initialized to 10 us. It is worth pointing out that the link delay reflects part of the propagation delay and is used together with number of hops, link data rate and link max queue size to estimate the worst case RTT. Various TCP timeouts are then set according to the estimated worst case RTT. The current version of the simulator does not implement reconnection for TCP timeouts which cause the socket to close, so this parameter is increased to avoid TCP timeouts.

Although it is a simple topology, it is representative of today's data center traffic patterns as paper [10] concluded that as much as 80% of data center traffic originated from the servers stays within the rack. This traffic pattern may evolve in the future driven by advances in hardware technology as well as load-balancing and fault-tolerance strategies. The simulator provides the flexibility to construct customized and complex network topology in future testing.

5.4.2 Background Flows

The background flows are meant to mimic the traffic in a typical data center, specifically a web search application as these are highly latency-sensitive and

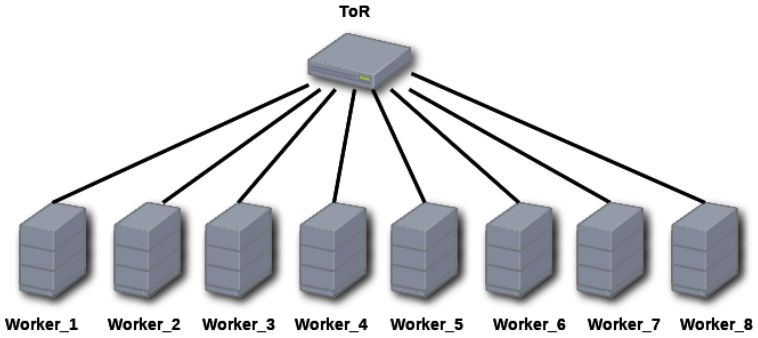

Figure 5.4: Single ToR topology with 8 workers

Parameter Name	Default Value
simulation time (s)	12
number of priority queues	1
horovod priority	low
background flows priority	high
link bandwidth (Gbit/s)	10
link delay (ns)	10000
link max queue size (packet)	1

Table 5.2: Default values for network-related parameters

could benefit from additional network bandwidth if there is competition for resources. We used Simon's implementation of the web search application [27]. The start times of background flows follow a Poisson inter-arrival distributions and the flow size is drawn from an interpretation of a CDF of web search traffic observed in paper DCTCP [5] and pFabric [7].

5.4.3 Test Parameters

Table 5.2 lists the default value of network-related parameters used.

5.4.4 Test Configurations

The following tests are designed to inspect how the tenants sharing the same network interact with each other under different network conditions. Each configuration is run three times with different seeds for background flow generation and the results are averaged among all three runs.

1. Varying the ratio between Horovod's compute time and network transmission time of a single iteration while maintaining a constant ratio of background and Horovod traffic at different link utilizations.
2. Varying the ratio between background flow traffic and Horovod traffic while maintaining a constant ratio of Horovod compute and network at different link utilizations.

The ratio between Horovod compute time and network transmission time can be represented using the formula below:

$$\begin{aligned} R_{cp\_to\_nw} &= \frac{compute\_time}{network\_transmission\_time} \\ &= \frac{iter\_time * link\_bw\_Mbits * 10^3}{bytes\_per\_iteration * 8} \\ &= \frac{iter\_time * link\_bw\_Mbits * 10^3}{model\_size\_in\_bytes * 2(1 - 1/N_{worker}) * 8} \end{aligned} \tag{5.1}$$

By decomposing and examining the formula above, we can make an interesting observation that is the ratio of compute and network for Horovod is actually the reciprocal of the link utilization by Horovod for a given network bandwidth as shown in Formula 5.2. The assumption is that Horovod is compute bound and the iteration_time is defined using the compute time. It is currently not our focus to examine a DDNN model that is bottlenecked at network as there are fewer opportunities to de-prioritize traffic without harming performance. However it is still an opportunity to consider the trade-off of improving latency-critical applications at the cost of a controlled degradation of Horovod's performance in a resource-constrained data center setting. Our simulation platform allows users to experiment and measure the performance degradation and improvement that would best suit their needs.

$$\begin{aligned} U_{Horovod} &= \frac{avg\_bw\_consumed\_per\_iteration}{link\_bw\_Mbits} \\ &= \frac{\frac{bytes\_per\_iteration*8}{iter\_time}}{link\_bw\_Mbits} \\ &= \frac{model\_size\_in\_bytes * 2 * (1 - \frac{1}{N_{worker}}) * 8}{iter\_time * link\_bw\_Mbits * 10^3} \end{aligned} \tag{5.2}$$

$$\begin{aligned} U_{pfabric} &= \frac{flow\_traffic\_per\_sec}{link\_bw\_Mbits} \\ &= \frac{avg\_flow\_rates\_per\_sec * avg\_flow\_size\_MB * byte\_to\_bit}{link\_bw\_Mbits} \\ &= \frac{avg\_flow\_rates\_per\_sec * 1.7 * 8}{link\_bw\_Mbits} \end{aligned} \tag{5.3}$$

$$U_{total} = U_{Horovod} + U_{pfabric} \tag{5.4}$$

As for the ratio between background flow and Horovod, we consider the total amount of bytes transmitted per second by each application as depicted

$R_{cp\_to\_nw}$	iteration_time_ms	$U_{Horovod}$
16	2240	6.25%
8	1120	12.5%
4	560	25%
2	280	50%

Table 5.3: Test configuration: Horovod Iteration Time and Horovod Link utilization at different $R_{cp\_to\_nw}$

quantitatively by Formula 5.5

$$\begin{aligned} R_{p\_to\_hrvd} &= \frac{flow\_traffic\_per\_sec}{horovod\_traffic\_per\_sec} \\ &= \frac{avg\_flow\_rates\_per\_sec * avg\_flow\_size\_in\_MB}{horovod\_traffic\_per\_sec} \\ &= \frac{avg\_flow\_rates\_per\_sec * 1.7 * 8}{\frac{model\_size\_in\_bytes * 2(1-\frac{1}{N_{worker}}) * 8}{iter\_time}} \\ &= \frac{avg\_flow\_rates\_per\_sec * 1.7 * iter\_time}{model\_size\_in\_bytes * 2(1-\frac{1}{N_{worker}})} \end{aligned} \tag{5.5}$$

To obtain different compute and network ratios for Horovod, we can vary the iteration time while keeping the model size, number of workers and link bandwidth constant. In the real world, this would be caused by changes in the minibatch size or GPU hardware. However, as mentioned before, the link utilization of Horovod is proportional to the ratio between network and compute time of Horovod. As a result, Horovod link utilization is implicitly altered and not possible to be kept constant. In fact, by decreasing the iteration time, we are essentially increasing the network utilization of Horovod as the amount of data to be transferred per iteration stays the same but the time to transmit it is shortened, assuming Horovod is still compute-bound and all gradients will be synchronized before the end of each iteration.

Based off the underlying dependencies between the key metrics, we will not be able to vary the link utilization while keeping the ratio of network bandwidth consumption between the two applications (background flows and Horovod) constant for a given Horovod compute-to-network ratio. But it is still valuable to examine how a fixed Horovod link utilization with different network prioritization would interact with an increasing number of background flows.

The variable in the tests here is the ratio between Horovod's compute and network time per iteration. Because we are currently only interested in the compute-bound case that would potentially leave us room for network de-prioritization of Horovod traffic, the ratios to be tested are all above 1. Halving the ratio at each test makes Horovod less compute bottlenecked and more vulnerable to network congestion. Table 5.3 shows that ratio and corresponding Horovod iteration time in milliseconds used in the tests, starting from the most compute-bound configuration of 16 and halving the ratio at each subsequent test. For each $R_{cp\_to\_nw}$ value, we vary the ratio between Horovod traffic and background flows by doubling the background flow arrival rates four times, ranging from as

$R_{cp\_to\_nw}$	16			
Flow_rate	23	46	92	184
$U_{Horovod}(\%)$	6.25	6.25	6.25	6.25
$U_{pfabric}(\%)$	3.125	6.25	12.5	25
$U_{total}(\%)$	9.375	12.5	18.75	31.25
$R_{p\_to\_hrvd}$	0.5	1	2	4
$R_{cp\_to\_nw}$	8			
Flow_rate	23	46	92	184
$U_{Horovod}(\%)$	12.5	12.5	12.5	12.5
$U_{pfabric}(\%)$	3.125	6.25	12.5	25
$U_{total}(\%)$	15.63	18.76	25.01	37.5
$R_{p\_to\_hrvd}$	0.25	0.5	1	2
$R_{cp\_to\_nw}$	4			
Flow_rate	23	46	92	184
$U_{Horovod}(\%)$	25	25	25	25
$U_{pfabric}(\%)$	3.125	6.25	12.5	25
$U_{total}(\%)$	28.13	31.25	37.51	50.02
$R_{p\_to\_hrvd}$	0.125	0.25	0.5	1
$R_{cp\_to\_nw}$	2			
Flow_rate	23	46	92	184
$U_{Horovod}(\%)$	50	50	50	50
$U_{pfabric}(\%)$	3.125	6.25	12.5	25
$U_{total}(\%)$	53.13	56.25	72.5	75
$R_{p\_to\_hrvd}$	0.0625	0.125	0.25	0.5

Table 5.4: Test configuration: Link Utilization at different flow rates and $R_{cp\_to\_nw}$

small as 6.25% of Horovod traffic to at most four times larger than Horovod. As a result, the expected total link utilization also increases as the background flow grows from 9.375% to 75%. The tests are designed to run in the range of 10% to 40% of full link capacity to reflect that network utilization are rather low in modern data centers in almost all layers but the core as past research [10] has reported.

5.4.5 Results and Analysis

We first examine the performance of both Horovod and background flows as we increase the volume of background flows while keeping Horovod traffic constant for a given Horovod compute-to-network ratio.

Effects of different flow rates at given compute-to-network ratio

Compute-to-network = 16

Figure 5.5 shows a set of performance metrics for background flows and Horovod as well as link utilization of each application when the compute-to-

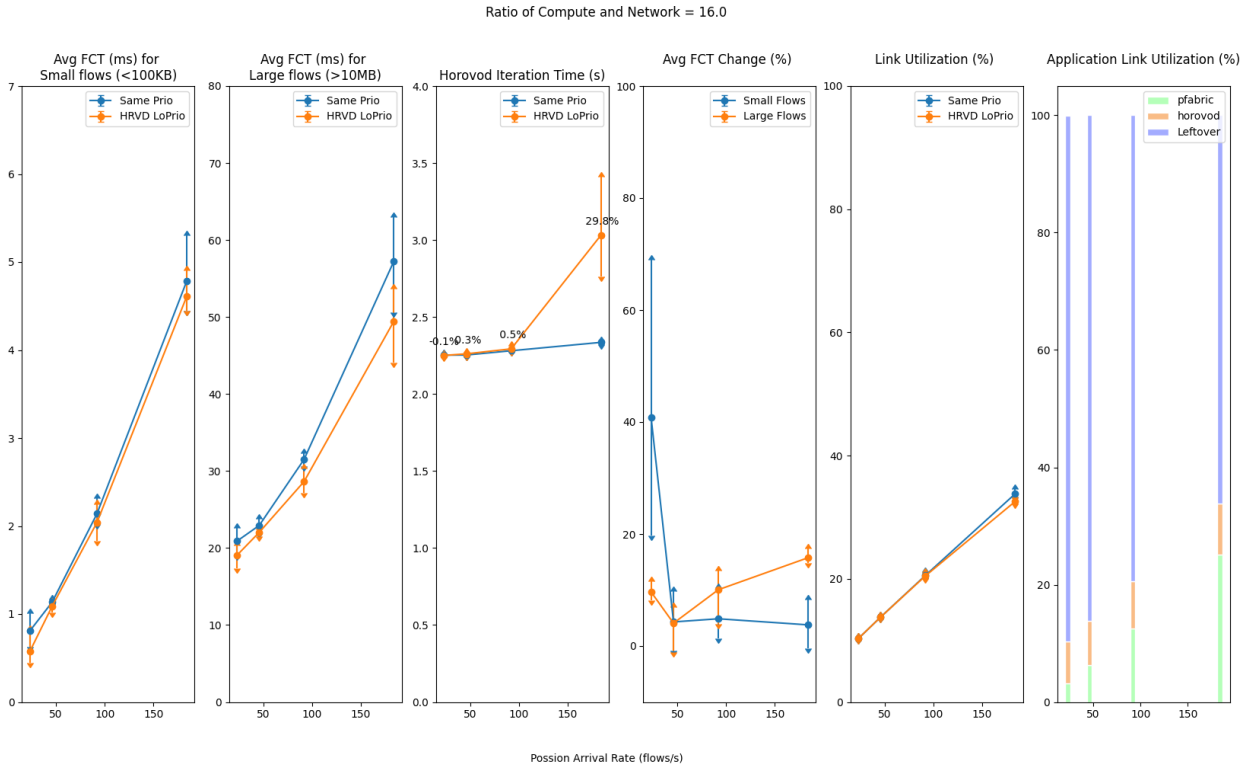

Figure 5.5: Performance Results for $R_{cp\_to\_nw} = 16$, heavily compute-bound.

network ratio of Horovod is 16. It can be observed that regardless of the priorities, as link utilization increases, average flow completion time increases for both small and large flows. The link utilization for Horovod is kept at 6.25% for all runs regardless of the background traffic. By running Horovod at the same priority as the background flows, the iteration time suffers almost no degradation as the background traffic grows from half of Horovod's traffic to four times more taking up as much as 25% of total link utilization.

On the other hand, by operating Horovod at a lower priority than the background flows without other optimization, we can expect a 40% reduction of average-flow-completion-time for small flows and 10% for large flows when the size of network traffic of Horovod is twice the amount of the background flows. However, as we increase the background traffic gradually, the performance gain for small flows reduces drastically and levels out below 10% while large flows continue to see an increased reduction in average flow-completion-time close to 15%. However, without additional optimization, it is not recommended deprioritize all of Horovod traffic if the background flows are over twice as much as that of Horovod. Otherwise, it is safe to run all Horovod traffic at a lower priority and expect no performance degradation while benefiting both the small and large background flows. It is especially beneficial to data centers where a small amount latency-critical flows taking up less than 5% of the link utilization to be set at a higher priority than Horovod as we can expect on average of

40% and sometimes up to 70% FCT reduction without compromising Horovod's performance. There is opportunity here to maintain the performance gain for large flows at high link utilization while dynamically de-prioritizing some layers of Horovod's traffic. We would like to address this in future work.

Although the small flows enjoy a significant performance gain at low utilization as high as 70%, the variance of the gain is not negligible among three runs. To investigate the high variance for this configuration (corresponding to the leftmost points of each graph in Figure 5.5), we first look at the number of completed flows among flows which are shown in Table 5.5. Each configuration is run three times with different seeds which means same run with different configurations are fed with the same seed. Regardless of Horovod's priority settings, we should expect to see the same number of background flows with same flow size distribution during the test interval, which has a two-second warm-up and a two-second cool-down time to make sure that all flows to be examined are completed to avoid side effects from unfinished flows. As the last column shows, the completion ratio of all flows are 100% which meets our requirement. In addition, the number of completed small flows across different runs are at most 3% apart. The difference is within the normal testing randomness which can't explain the high variance we saw earlier. Next we will examine the distribution of the duration of all the completed small flows across different runs.

All three runs have very similar cumulative distributions for small flows as seen in Figure 5.6. They all show that by running Horovod at the lower priority than that of the background traffic, we can help reduce the completion time of about 15% of all small flows. In the middle plot which is during the second run, there is significant reduction of the long tail which means some of the 15% flows are the longest running small flows. The total reduction of average flow completion time is about 72.8%, 3 times higher than the performance gain measured in the third run where most of flows benefited have shorter duration to start with.

It is also worth taking a look where these flows that enjoyed a performance boost sit on the simulation timeline in Figure 5.7. The gains of small flows represented in vertical lines in the second run (leftmost plot on second row in the figure) are clearly longer than those in the other two runs (top row and bottom row). The start time of these flows also align with when RingAllreduces start in Horovod. The high variance between runs can be attributed to a higher percentage of long-tail small flows in the second run that start around the same time when RingAllreduce starts in Horovod.

run index	all		
flow size	completed	collected	completion ratio (%)
all flows	4341	4341	100
small flows	2345	2345	100
medium flows	1867	1867	100
large flows	129	129	100
run index	0		
flow size	completed	collected	completion ratio (%)
all flows	1467	1467	100
small flows	820	820	100
medium flows	609	609	100
large flows	38	38	100
run index	1		
flow size	completed	collected	completion ratio (%)
all flows	1458	1458	100
small flows	792	792	100
medium flows	623	623	100
large flows	43	43	100
run index	2		
flow size	completed	collected	completion ratio (%)
all flows	1416	1416	100
small flows	733	733	100
medium flows	635	635	100
large flows	48	48	100

Table 5.5: Flow Counts for $R_{cp\_to\_nw} = 16$

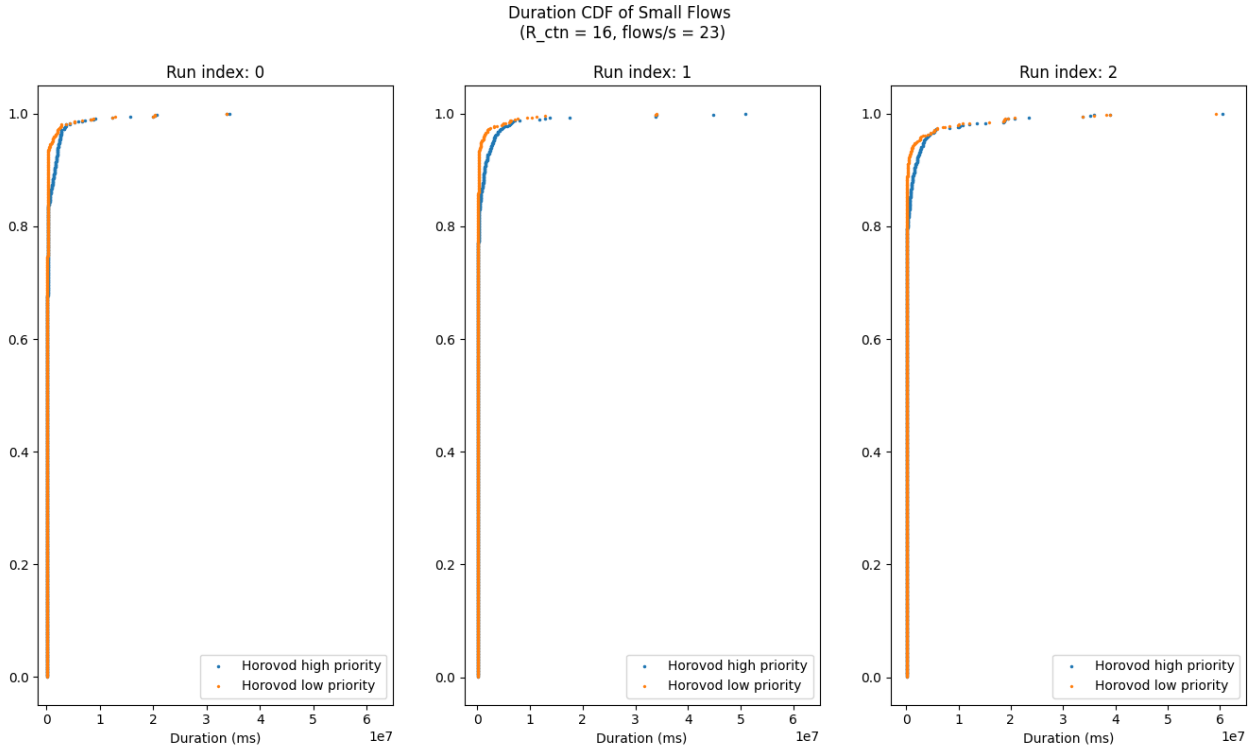

Figure 5.6: Duration CDF for small flows across three runs with $R_{cp\_to\_nw} = 16$ and $flows/sec = 23$

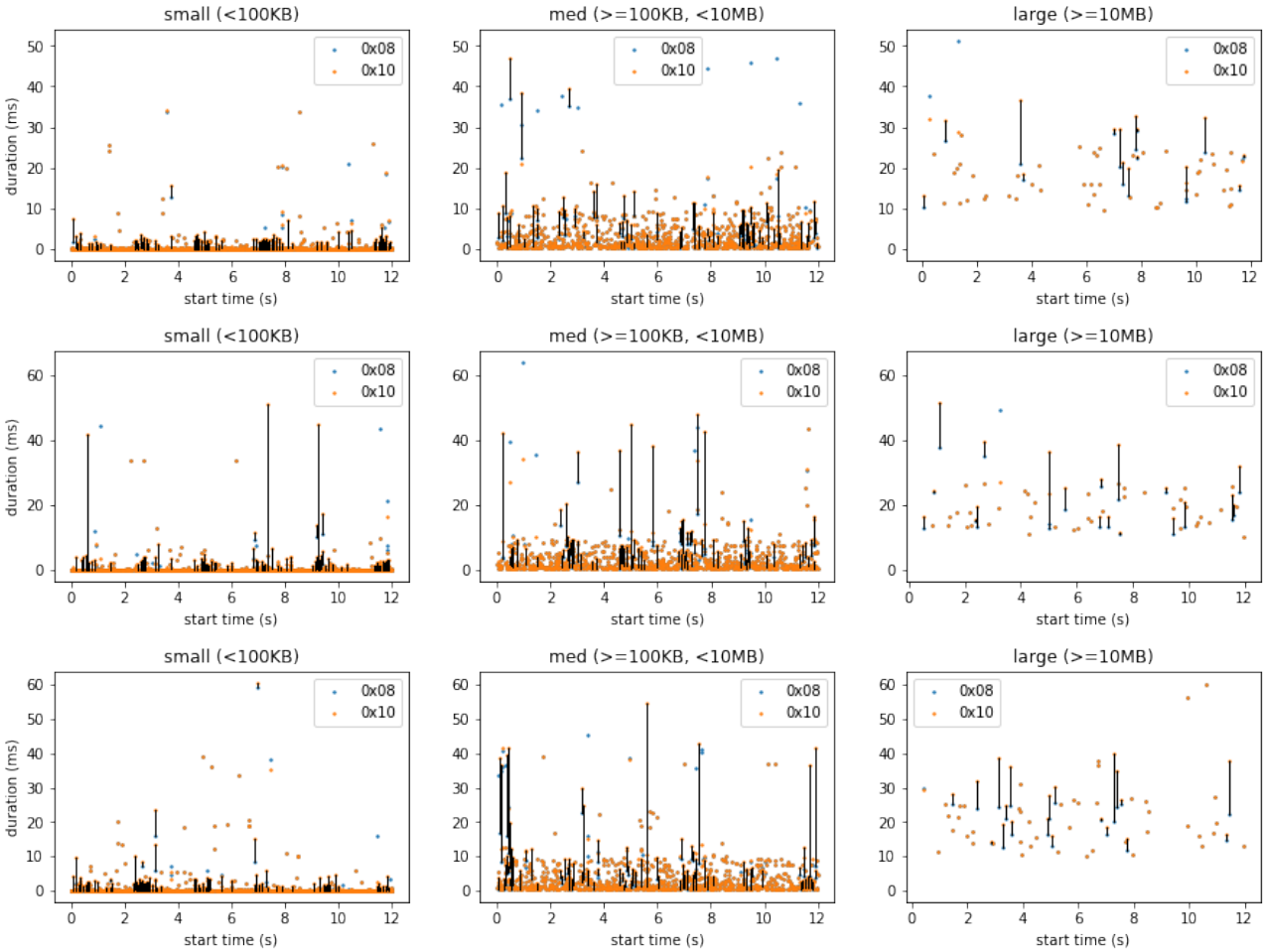

Figure 5.7: Timeline of execution $R_{cp\_to\_nw} = 16$ and $flows/sec = 23$ for 3 different runs with Horovod at high priority (0x08) vs low priority (0x10). Each high and low priority run use the same random schedule for sending background traffic, so the duration of each background flow is directly comparable to the other priority run. Vertical black lines mark the parallel flows that had the largest completion time difference.

Compute-to-network = 8

With the ratio of compute to network halving to 8 from 16, Horovod now is more network demanding and requires a higher network bandwidth to stay compute-bound. The last graph in Figure 5.8 shows that Horovod occupies about 12.5% of the total link capacity for the first three configured runs and around 7% on the last run. The drop in link utilization can be explained by a jump in Horovod iteration time from 1.4s to 3.06s, a 118.5 % difference comparing to the previous run with 92 flows/s and 117.2% longer than running at a higher priority. However, as with last experiment when $R_{cp\_to\_nw}$ is set to 16 and with the highest flow rate of 184, there is an opportunity in optimizing Horovod's traffic with a more selective prioritization schema to reduce the iteration time. We will address that in chapter 6 with a few proposals.

If the ratio between Horovod traffic and background flows is kept under one, as the first three points of each graph show, we can safely de-prioritize all of Horovod's packets and expect a close-to 20% performance gain for both small and large flows and as high as 70% when Horovod transmits four times more traffic than the background flows. Similar to previous experiment, the high variance in average flow-completion-time change at the low application utilization can be contributed to the heavy tail of cumulative duration distribution of small flows. If the prioritized packets belong to the rare but high-latency flows, it will reduce the average flow completion time significantly.

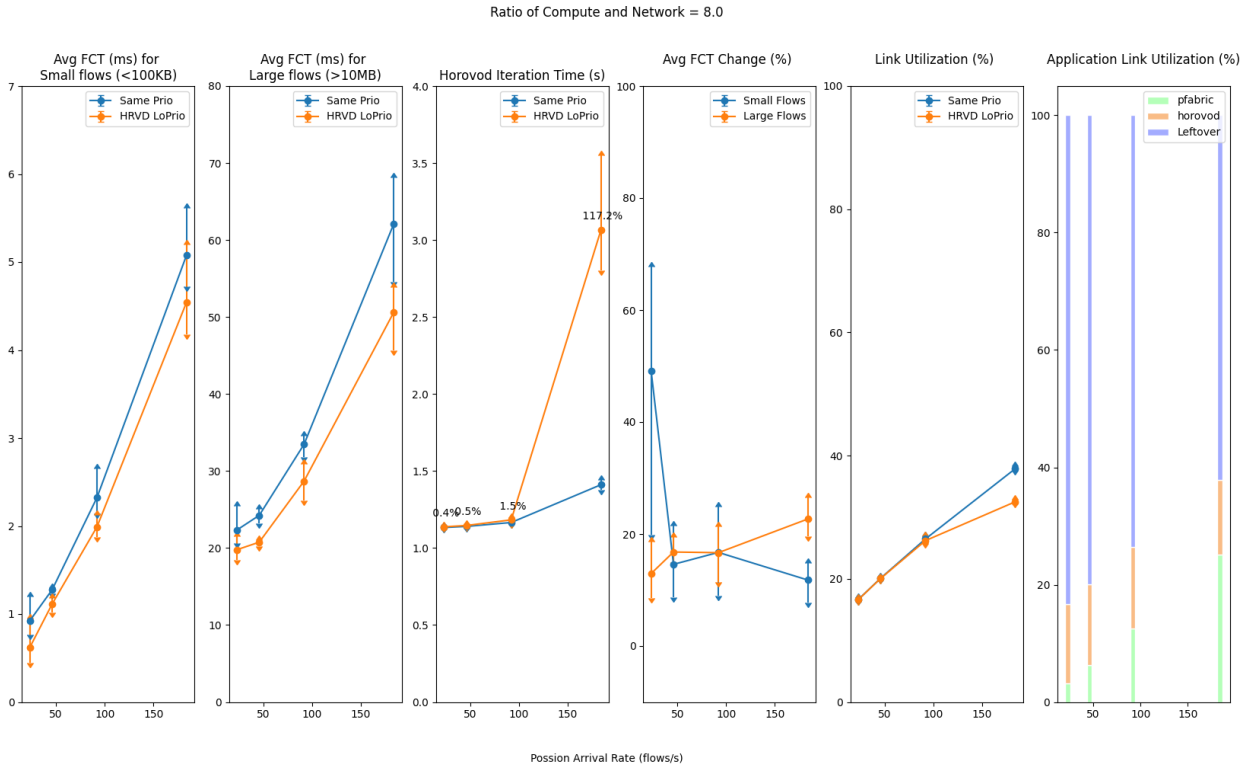

Figure 5.8: Performance Results for $R_{cp\_to\_nw} = 8$

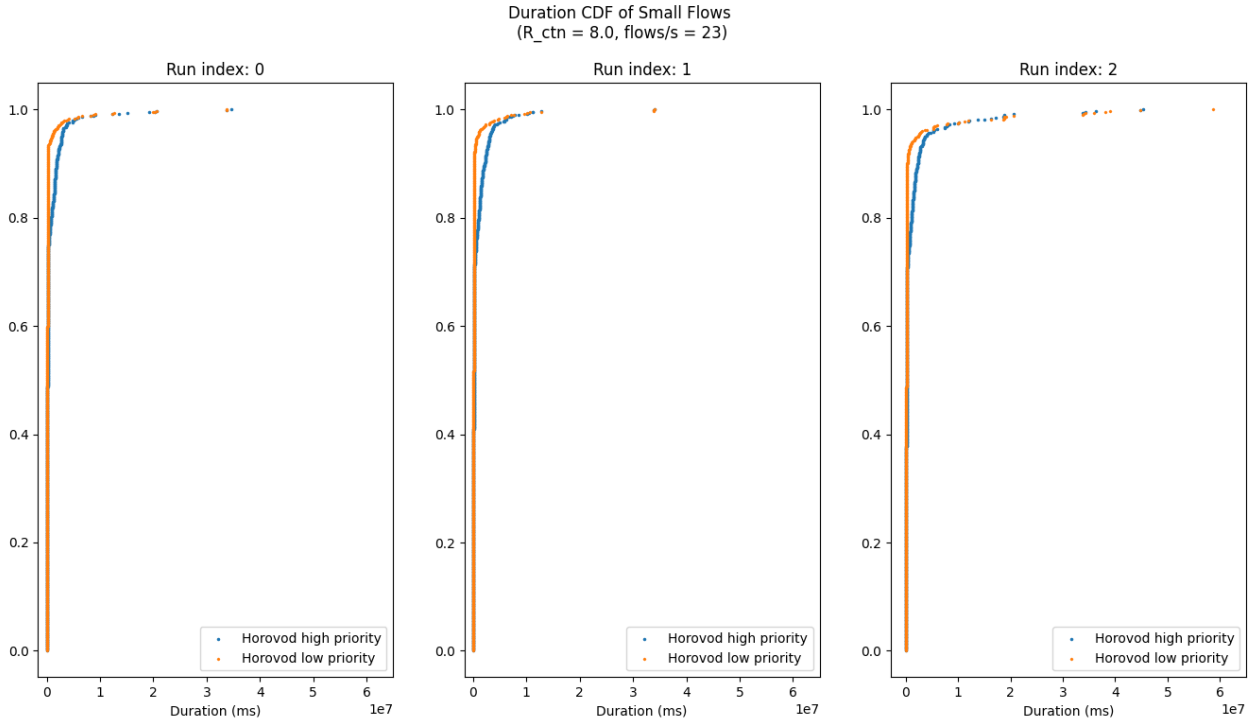

Figure 5.9: Duration CDF for small flows across three runs with $R_{cp\_to\_nw} = 8$ and $flows/sec = 23$. Figure 5.9 shows the duration CDF of the small flows and the plot from left to right represents a performance gain of 66%, 74% and 26% respectively. The third run with a different seed generated flows with a much longer tail and the long-latency flows mostly fit in between Horovod's traffic gaps and thus are unable to take advantage of Horovod's de-prioritization.

When Horovod uses an even larger 25% link bandwidth, at low flow rates, we can expect to see an even higher latency reduction for background flows as there is a higher chance that the two application will need to compete for network resources and background flows would have been starved longer than before as Horovod traffic is now 8 times more. By backing off Horovod's traffic, we can greatly improve the performance of small flows by at least 50% and large flows by at least 25% without compromising the performance of Horovod as long as Horovod's traffic is at least four times larger than the background traffic. As the background traffic keeps doubling, it will slow down Horovod traffic which is tagged at a lower priority. This is similar to previous tests that expose an opportunity for a more sophisticated prioritization algorithm, but the degradation to Horovod's performance happens at an earlier stage.

Compute-to-network = 4

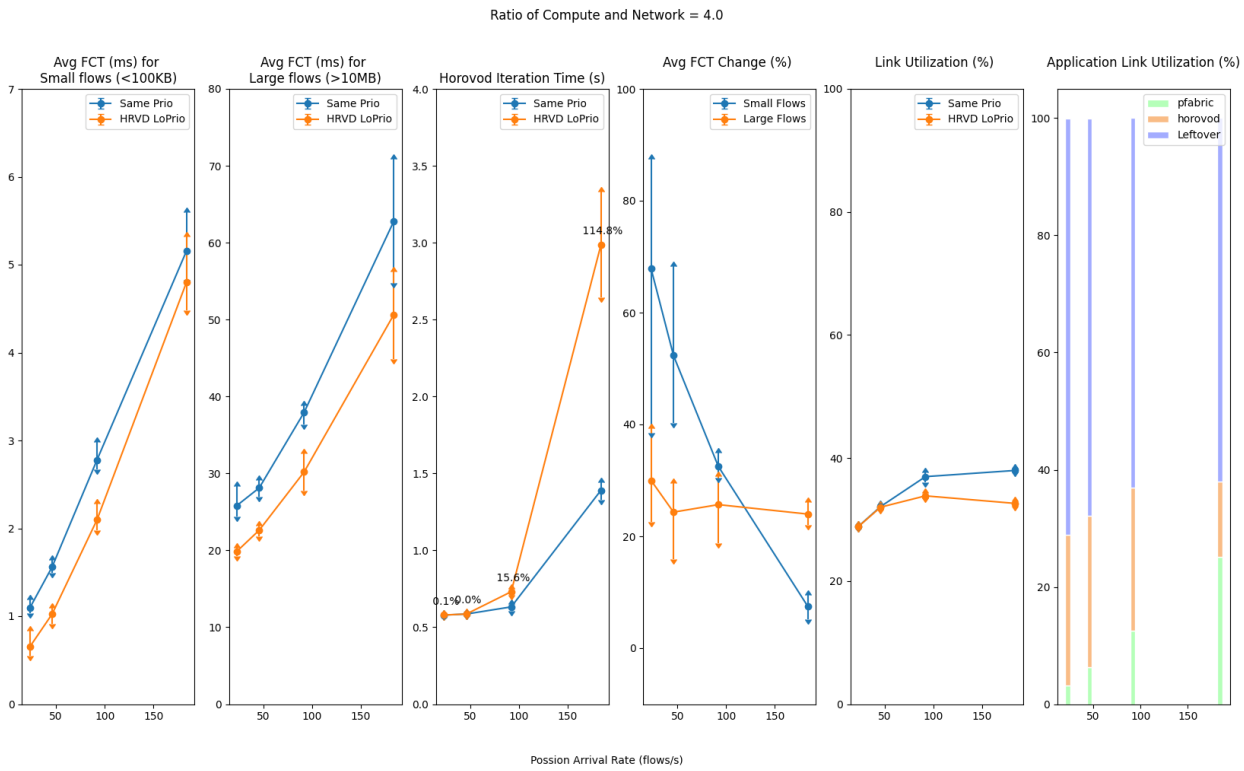

Figure 5.10: Performance Results for $R_{cp\_to\_nw} = 4$

Consistent with previous observations, the high variance in the results of the first experiment with a flow rate of 23 flows per second is contributed to the fact that the baseline is comprised of a higher percentage of long latency small flows than the other two runs as shown in Figure 5.11. Because all of the third run of each configuration is seeded with the same random number which gave rise to a long-tail baseline, it is expected to see that all third runs have shown the lowest performance gains across all three runs.

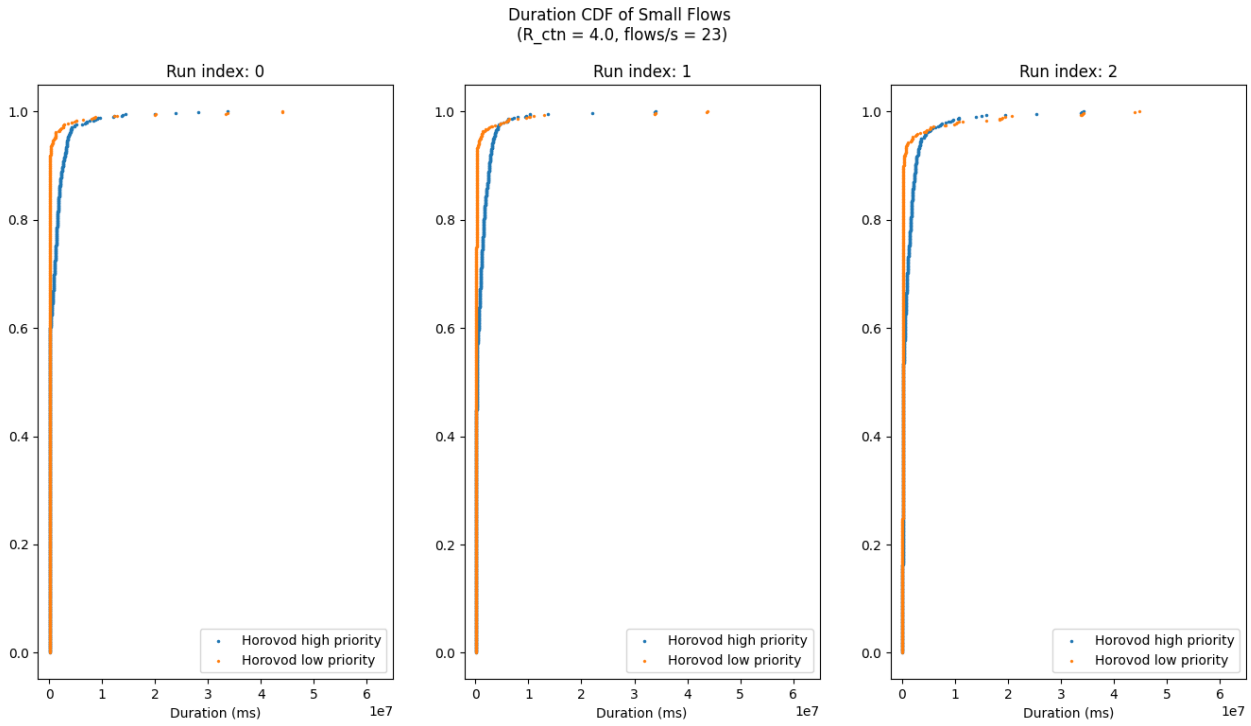

Figure 5.11: Duration CDF for Small flows across three runs with $R_{cp\_to\_nw} = 4$ and $flows/sec = 23$

Compute-to-network = 2

As Horovod's compute-to-network ratio reaches 2, which gives the lowest time slack among all configurations so far, by running Horovod at a lower priority would have a negative effect on its performance even when the background traffic is kept minimal at 6.25% of the total link usage. The only time where we can afford to de-prioritize all Horovod traffic is when the ratio between Horovod and background flows are high as 16 to 1 or when the background flows are only occupying about 3% of the total link bandwidth.

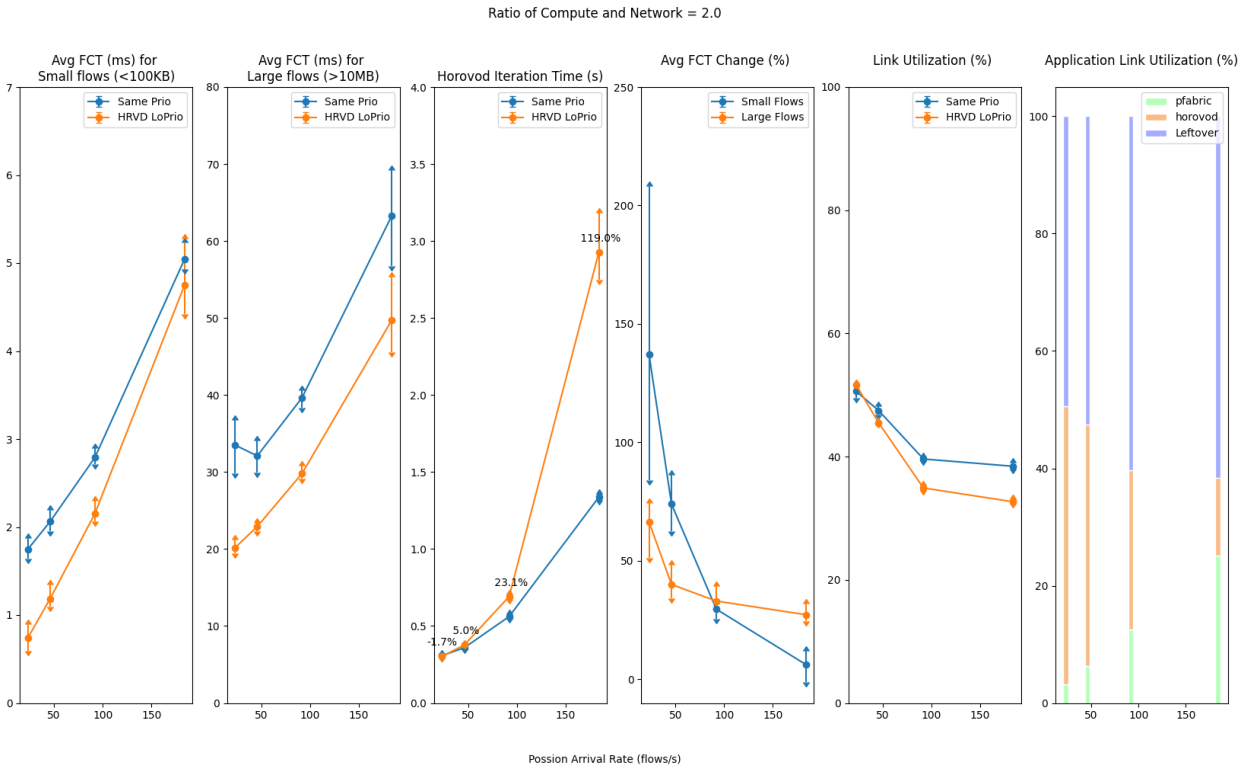

Figure 5.12: Performance Results for $R_{cp\_to\_nw} = 2$

Effect of different compute-to-nework ratios at given flow rate

To deepen our understanding of the gains and implications of the all-or-nothing prioritization policies, we will look at the results from different angles by changing the control variables. Previously, the results are grouped together based on a fixed compute-to-network ratio of Horovod. We observed some trends across different compute-to-network ratios which inspired us to look at results from the point of a fixed flow rate with varying compute-to-network ratios.

Flow arrival rate = 23

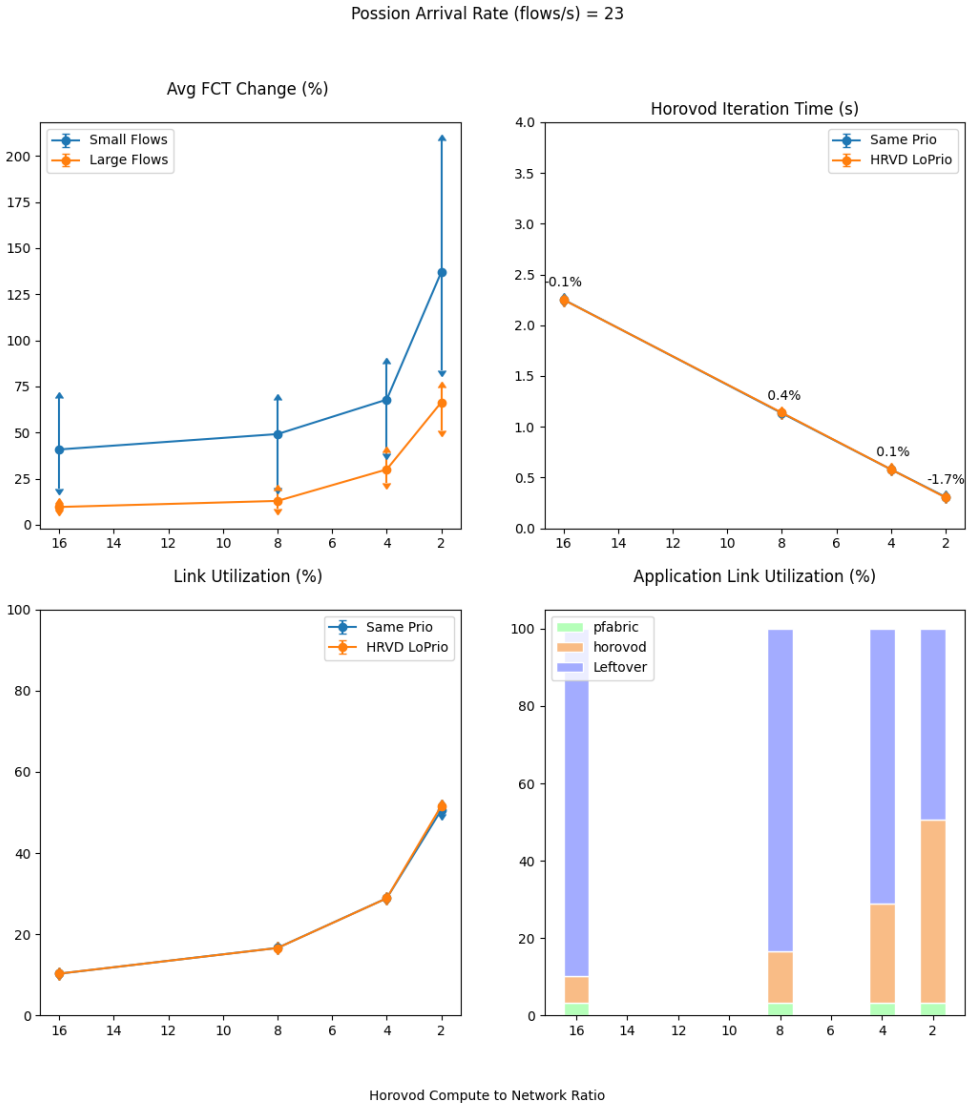

Figure 5.13: Performance Results for Flow Arrival Rate = 23

The top-left plot in Figure 5.13 shows the average FCT percentage differences small and large background flows when running Horovod at the same priority as them versus running Horovod traffic at a lower priority.

At 23 flows/s, which corresponds to less than 4% of the link capacity, we can safely ramp up Horovod traffic at a lower priority as long as the ratio between the two application is above 2. The performance gain increases as the gap between Horovod and background flows grows larger with Horovod becoming more dominating. It is even more beneficial to reducing latency of small flows than that of large flows. One can expect a gain of 40% at least for small flows and 10% for large flows.

Flow arrival rate = 46

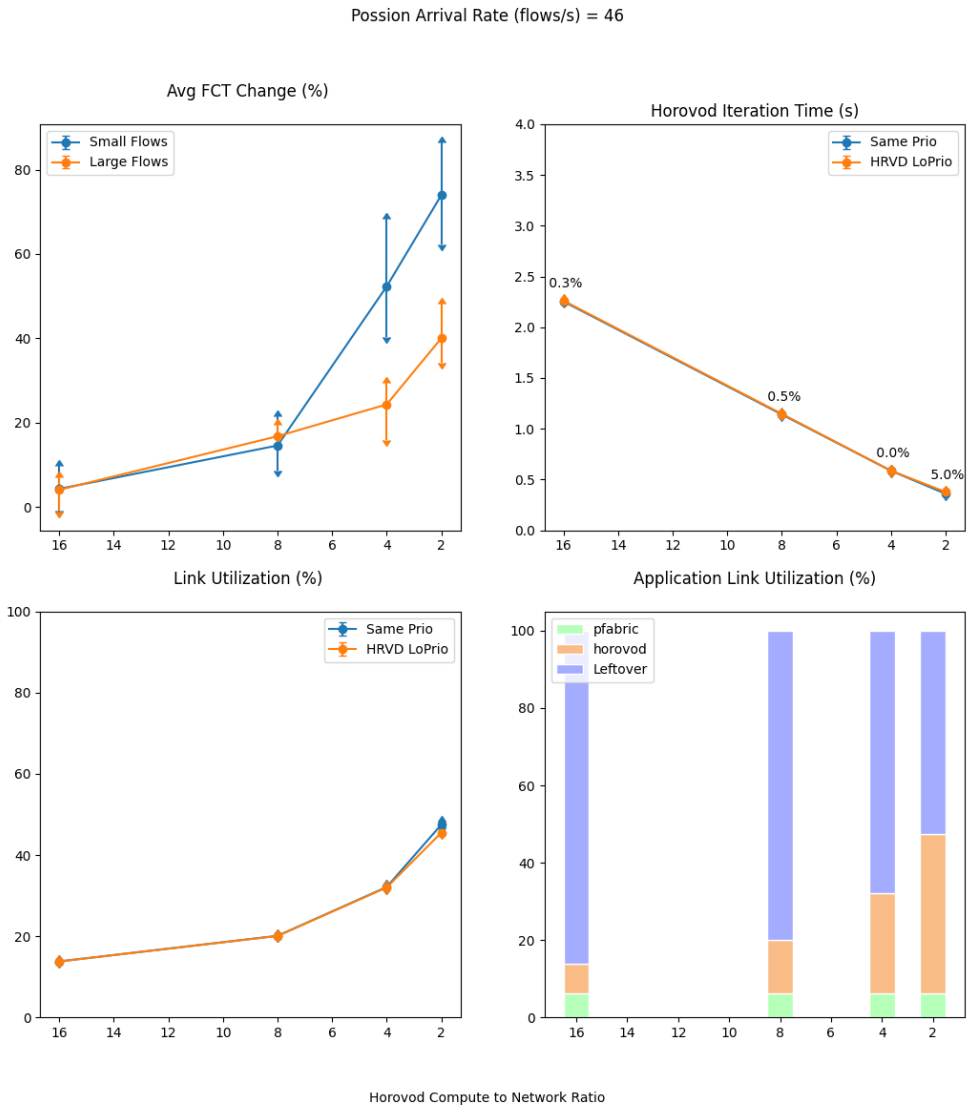

Figure 5.14: Performance Results for Flow Arrival Rate = 46

When we increase the background traffic from 3% to 6% as shown in Figure 5.14, the ratio between background flows and Horovod traffic now starts at 1 instead of 0.5 and gradually gets halved to 0.125 with Horovod taking up more bandwidth at each step. The performance gains are again, as expected, increasing as the gap between background flows and Horovod traffic increases. In addition Horovod's de-prioritization has a more balanced effect on both small and large flows at low utilization but favors small flows more as we ramp up the traffic. However, as we ramp up Horovod to 50% of the link bandwidth at a low utilization, there is a 5% performance degradation to Horovod traffic. With-

out further optimization, data center operators take other factors into account and make a performance trade-off accordingly. This data can provide network operators the tools to run experiments with configurable parameters that are representative of their use cases to eventually enable them to make data-driven decisions.

Flow arrival rate = 92

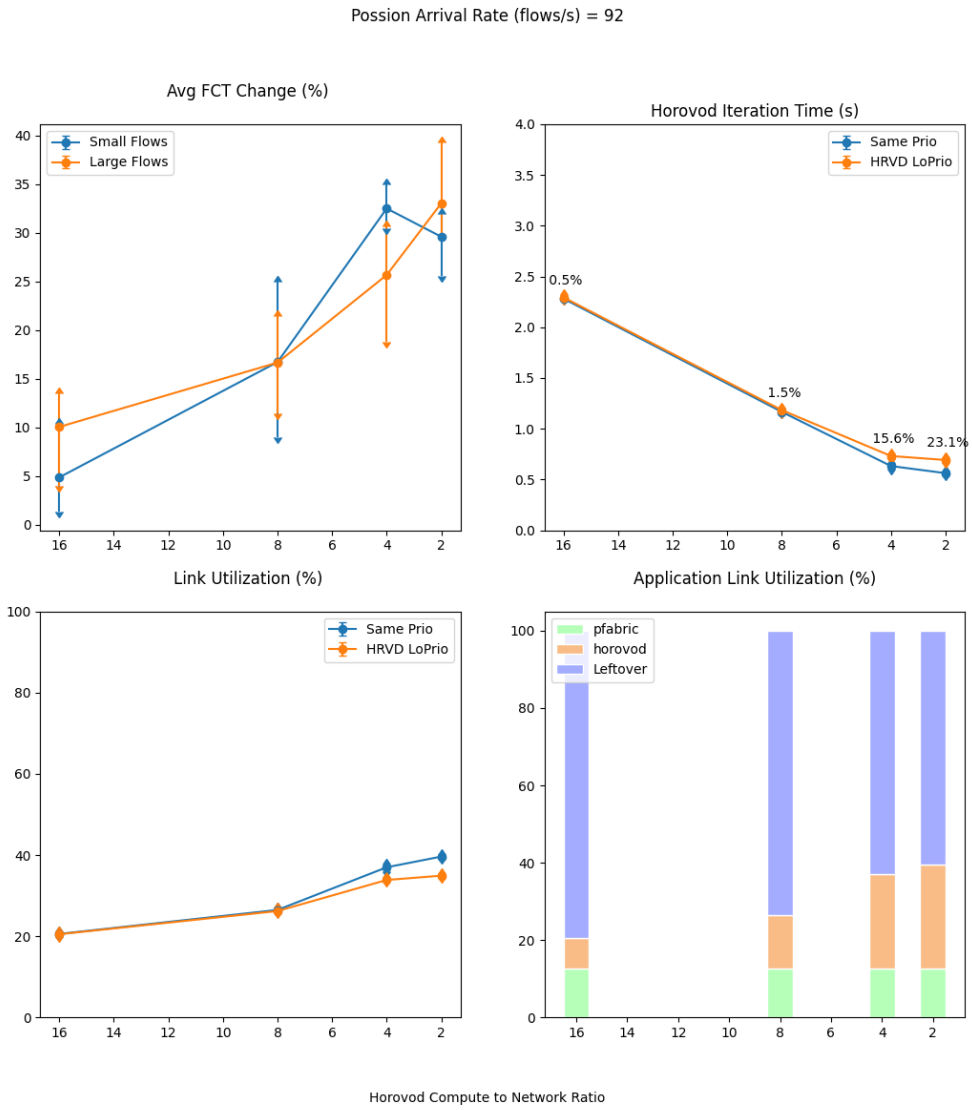

Figure 5.15: Performance Results for Flow Arrival Rate = 92

We continue to see the performance gain climbing up from 5% and 10% to 35% for large and small flows respectively shown in 5.15, but the gain comes at the cost of Horovod's performance as Horovod becomes more less compute bound. Specifically, when Horovod's compute-to-network ratio drops to 4, running background flows at higher priority will cause a 12.5% performance degradation to Horovod despite background traffic being only half of what Horovod generates. The degradation can go up to 23% when Horovod's compute-to-network ratio drops to 2. The actual link utilization reaches a plateau of 40% as Horovod has been significantly slowed down and there are few iterations collected during the

testing interval.

Flow arrival rate = 184

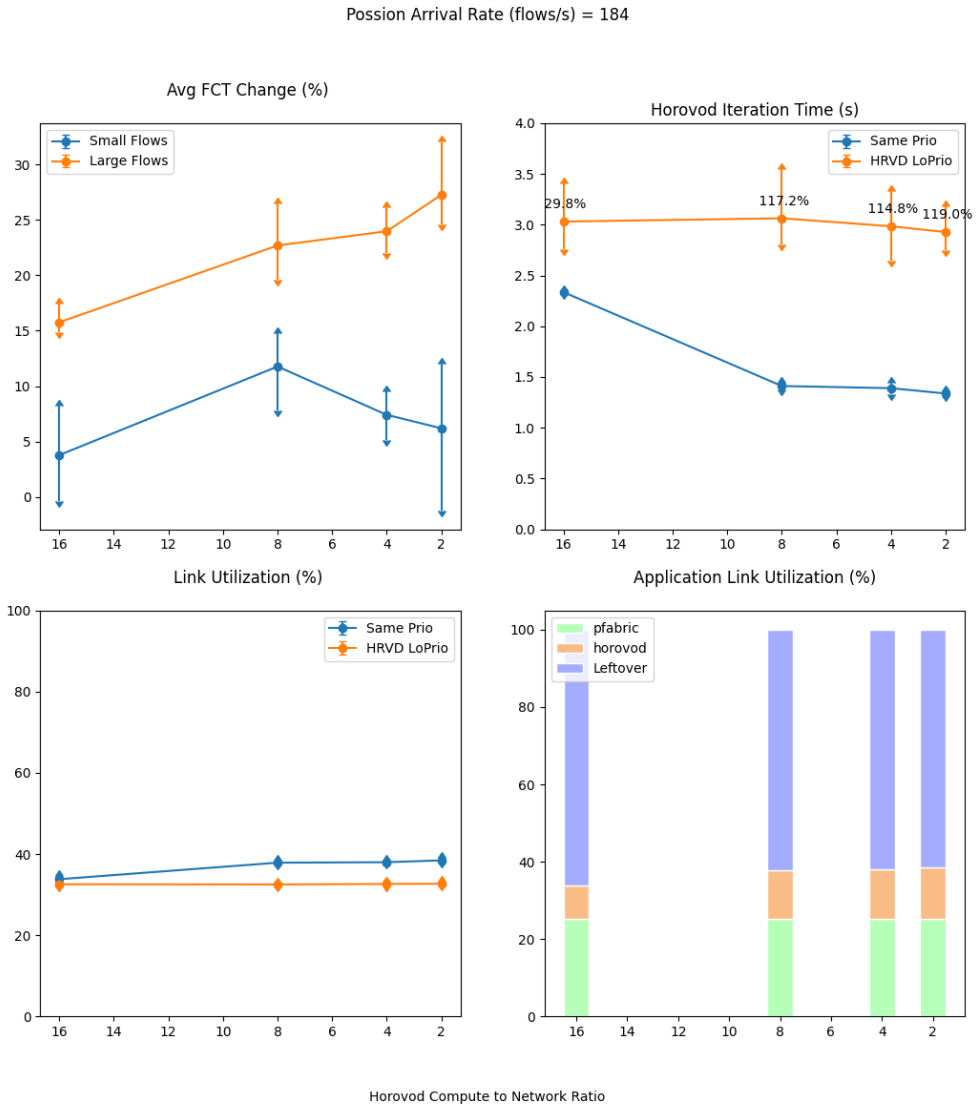

Figure 5.16: Performance Results for Flow Arrival Rate = 184

At an arrival rate of 184 flows/s the background traffic takes up 25% of the total link capacity, which is four times more than Horovod traffic on average. Even if Horovod is heavily compute-bound with a compute-to-network ratio of 16 (leftmost data points in each graph in Figure 5.16), it will suffer at least a 29% performance degradation when at a lower priority. The degradation worsens as Horovod becomes more network-demanding as we decrease the ratio of compute-to-network. It also leads to a diminishing return for the background

flows leveling out at 25% for large flows and 10% at small flows. An interesting result in this configuration is that the usually favored small flows enjoy a much lower gain comparing to large flows during high link utilization. This also presents an insightful data point to data center operators to consider. Depending on the co-flow composition of their business critical applications, there is decision to be made as to weather it is worth applying further traffic optimizations to Horovod if their applications are mostly comprised of small flows that would at most be boosted by 10% comparing to leaving all traffic running at the same priority. However the decision may be different if the applications have mostly latency-sensitive large flows.

Performance of background flows at different link utilization with constant traffic ratio to Horovod

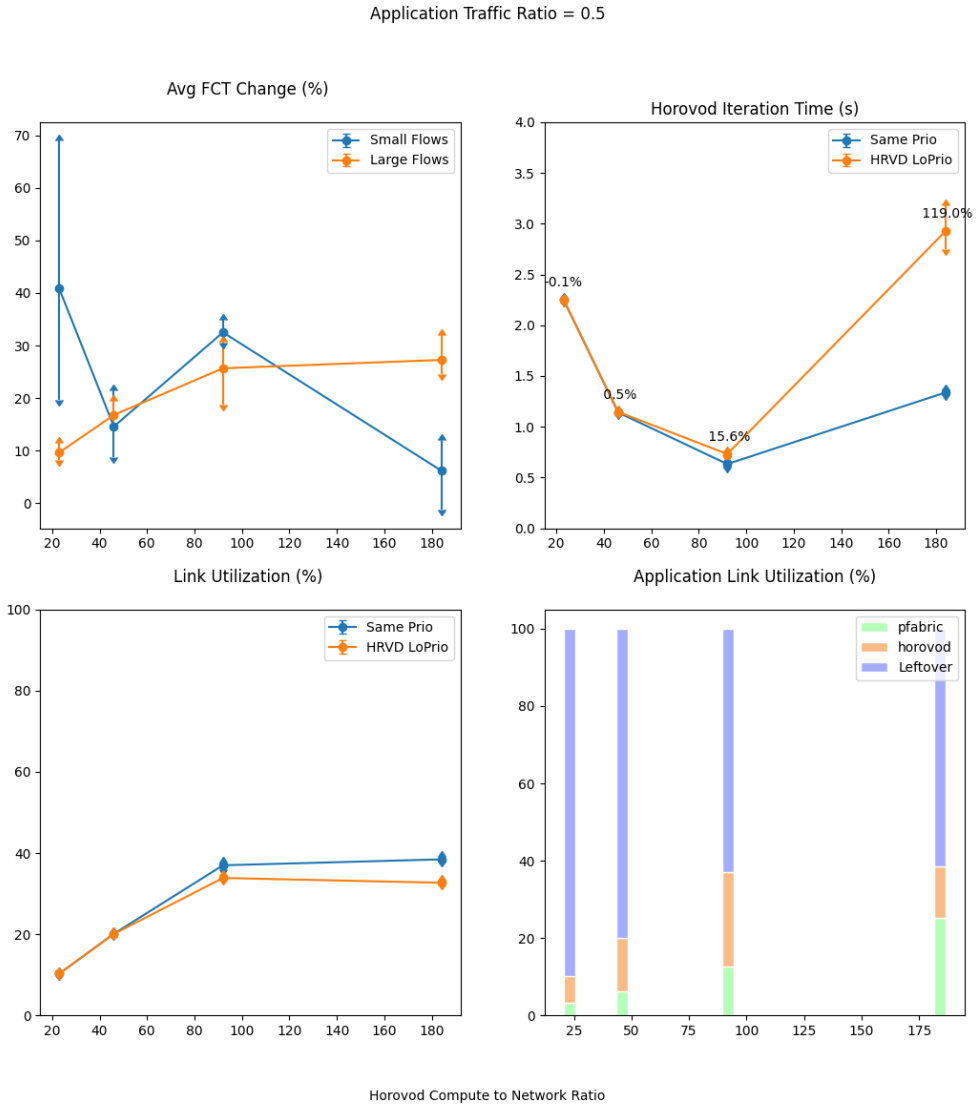

Figure 5.17: Performance Results for Application Ratio = 0.5

For data center environments where the application traffic is generally stable, it is useful to analyse the performance from the point of view of a fixed application ratio. For example, if we keep the ratio between Horovod and background flows to 0.5 with Horovod being twice as large shown in Figure 5.17, we can expect to see a 10-15% flow completion time reduction for small flows and a 15% to 40% reduction for large flows when the link utilization is below 20%. The gains go up to 25% for small flows and 35% for large flows if we keep increasing

the link utilization close to 40%, however Horovod will see a 15% increase in iteration time. As background flows reach up to 30% of link utilization, Horovod becomes network bound and failed to transmit all gradients within the desired iteration time.

Horovod-background ratio = 0.25

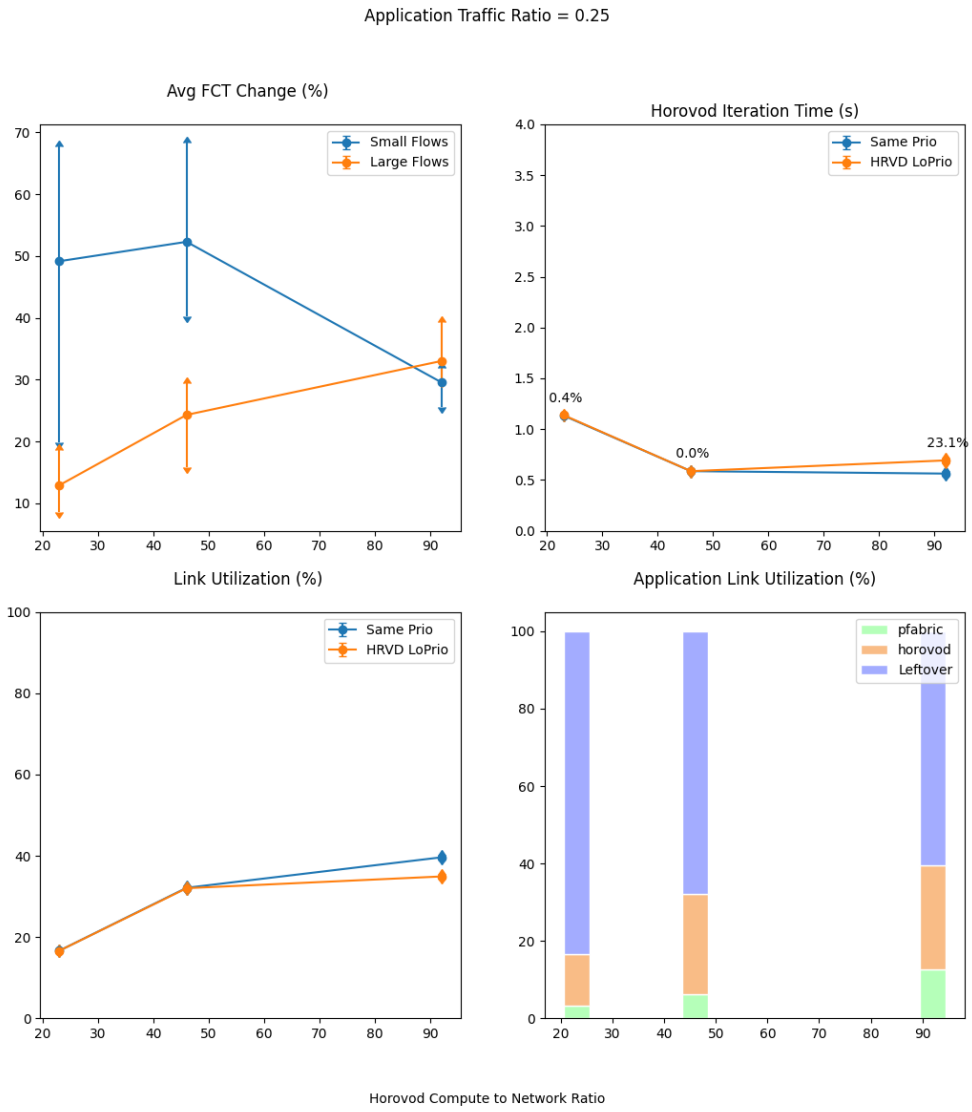

Figure 5.18: Performance Results for Application Ratio = 0.25

If Horovod generates four times more traffic than background flows, we can expect a 10% to 20% gain for large flows and 30% to 50% for small flows as seen in 5.18. Again, we are unable to push Horovod's traffic to above 40% as packets are getting heavily delayed from de-prioritization.

5.4.6 Operating Boundary

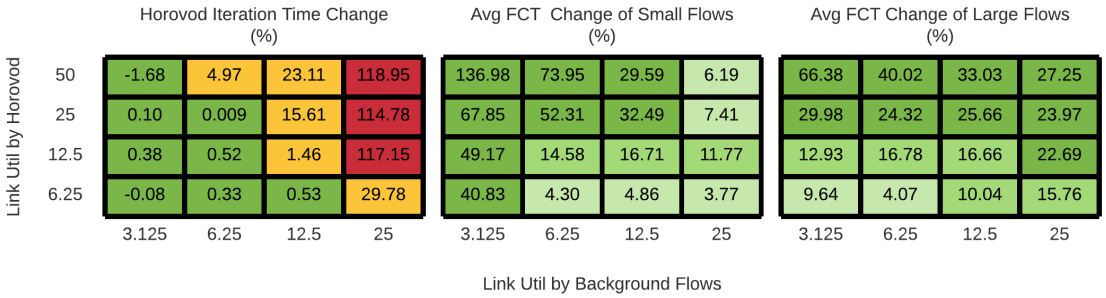

Figure 5.19: Summary of all previous tests sorted by the expected link utilization % of Horovod (y-axis) and of background flows (x-axis). From left to right, it shows the iteration time percent change of Horovod when run at the same vs lower priority (leftmost plot), the average flow completion time percent difference of small flows (middle plot), and the average flow completion time percent difference of large flows (right plot). The differences are attributed to de-priotization Horovod traffic as opposed to running Horovod at the same priority as the background flows.

We analyzed the runs and mapped out the operating regions where we can safely de-prioritize Horovod without any performance impact (green boxes, shown in the leftmost plot of Figure 5.19). In the yellow regions Horovod suffers between between 1% to 30% performance degradation when running at a lower priority. The red region represents the most serious degradation for Horovod when the background flows are taking 25% of the link capacity. In the middle and rightmost plots, the darker green represents larger time improvements for small and large flows, respectively. Some of the lightest green blocks that represent the smallest gains (under 10%) overlap with the red region of Horovod, specifically the top right two blocks on the last column in the second plot with gains of 6.19% and 7.41% respectively. It is up to the data center operators and application owners to decide if it is worth the effort to further optimize the de-prioritization schema for Horovod to reduce the performance degradation as at most the gains are under 10% for small flows. If their most latency critical conditions are in majority comprised of large flows, it may be worth it to devise more optimization to reduce Horovod's iteration time in exchange for about 25% gains for large flows. On the other hand, a more promising region to work on are the cells colored yellow as they represent a moderate amount of degradation but on average higher potential gains for both small and large flows. For that we will take a closer look at how Horovod progresses and see where we can bump the priority level to reduce the transfer time ideally. First we will examine the case when Horovod is at 25% of link utilization and background flows at 12.5%, in which case Horovod experiences a 15.61% performance degradation due to de-prioritization.

Figure 5.20 shows the progression of Horovod when running at the same

priority as the background flows. It shows the computation time of each layer as well as the duration of each partition transfer. There are 18 iterations completed during the 12s simulation run. Even though the computation-to-network ratio is 4, which makes Horovod compute bound, it can be seen that some iterations are struggling at sending partitions in time. For example, in the second iteration counting from the left, the transmission of the tensor fusion buffer containing partitions that belong to layers from 17 to 25 experienced significant delay comparing to the transmissions of other partitions which made this iteration more network bound despite being theoretically compute-bound.

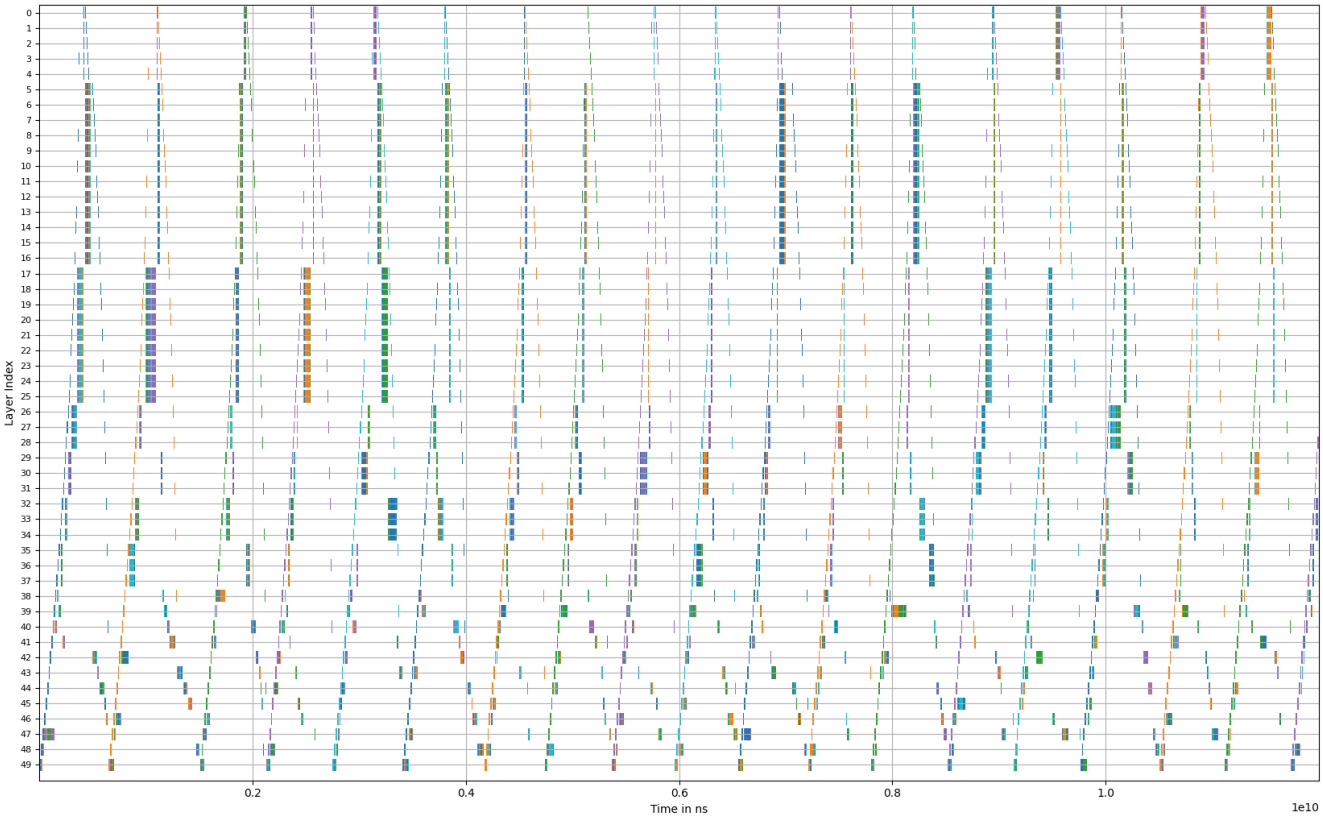

Figure 5.20: Horovod progression with Horovod set at the same priority as the background traffic with a compute-to-network ratio of 4 to 1 when the background traffic takes up about 12.5% of link bandwidth.

Comparing to Figure 5.20, Figure 5.21 shows a higher percentage of partitions that experienced longer transmission time, some of which belong to lower layers. One optimization that can be applied is to always send partitions of the lower

layers at high priority to minimize of the probability of getting delayed. However this simple optimization may not always work well as will be seen later.

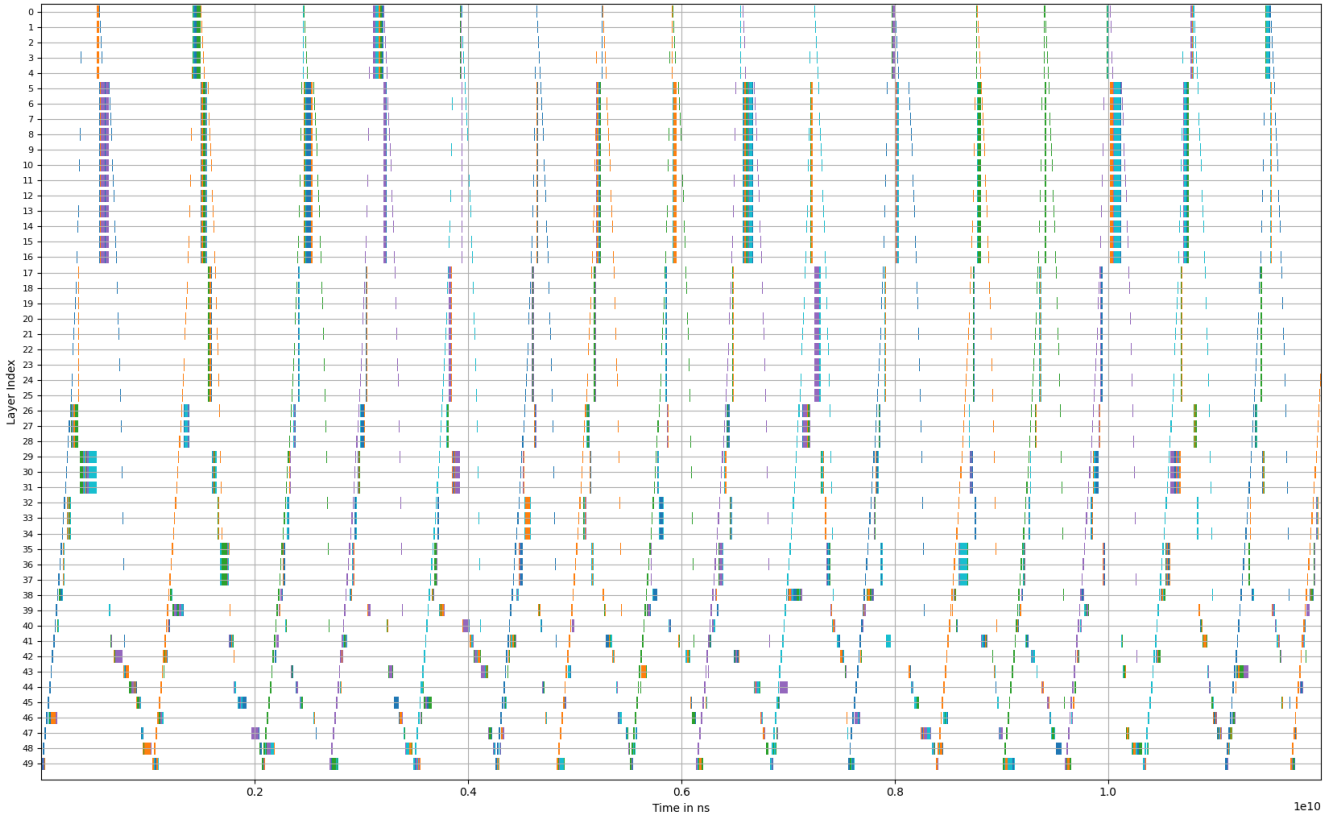

Figure 5.21: Horovod progression when Horovod is set at a lower priority as the background traffic with a compute-to-network ratio of 4 to 1 when the background traffic takes up about 12.5% of link bandwidth.

Another promising optimization opportunity is the yellow cell with a Horovod slowdown of 23.11% when the expected Horovod link utilization is at 50% (corresponding to a compute-to-network ratio of 2:1) and the link utilization by background flows is at 12.5%. Figure 5.22 and Figure 5.23 shows the progression when Horovod is set at the same priority and a lower priority as the background traffic, respectively. Unlike the delays observed earlier in the last two graphs, some partitions belonging to later layers also experience significant delays which increases the overall iteration time. Because of the unpredictability of which layers will struggle more with network bandwidth, simply setting lower layers at a higher priority will not always help. In the future we will look

into applying adaptive dynamic prioritization schemes to further optimize this particular case.

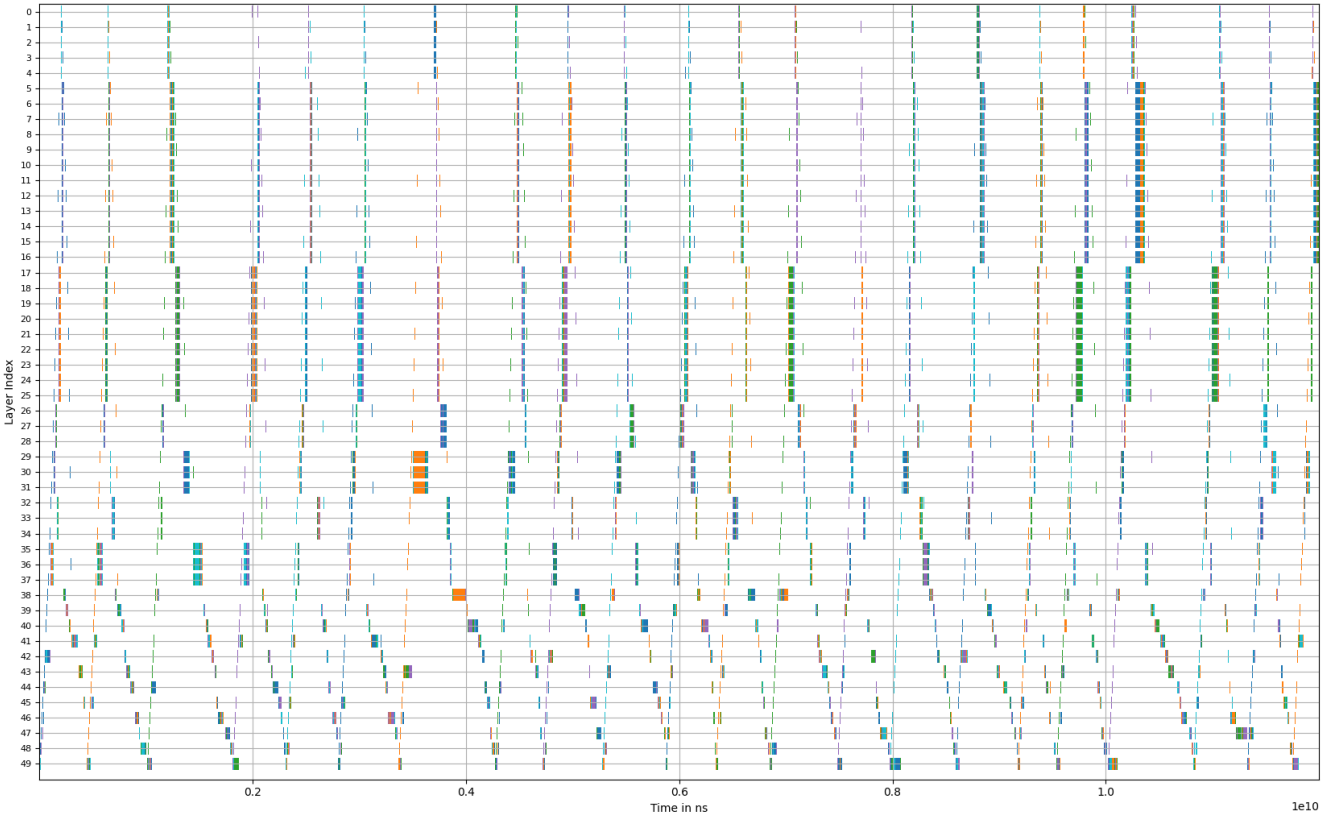

Figure 5.22: Horovod progression when Horovod is set at the same priority as the background traffic with a compute-to-network ratio of 2 to 1 when the background traffic takes up about 12.5% of link bandwidth.

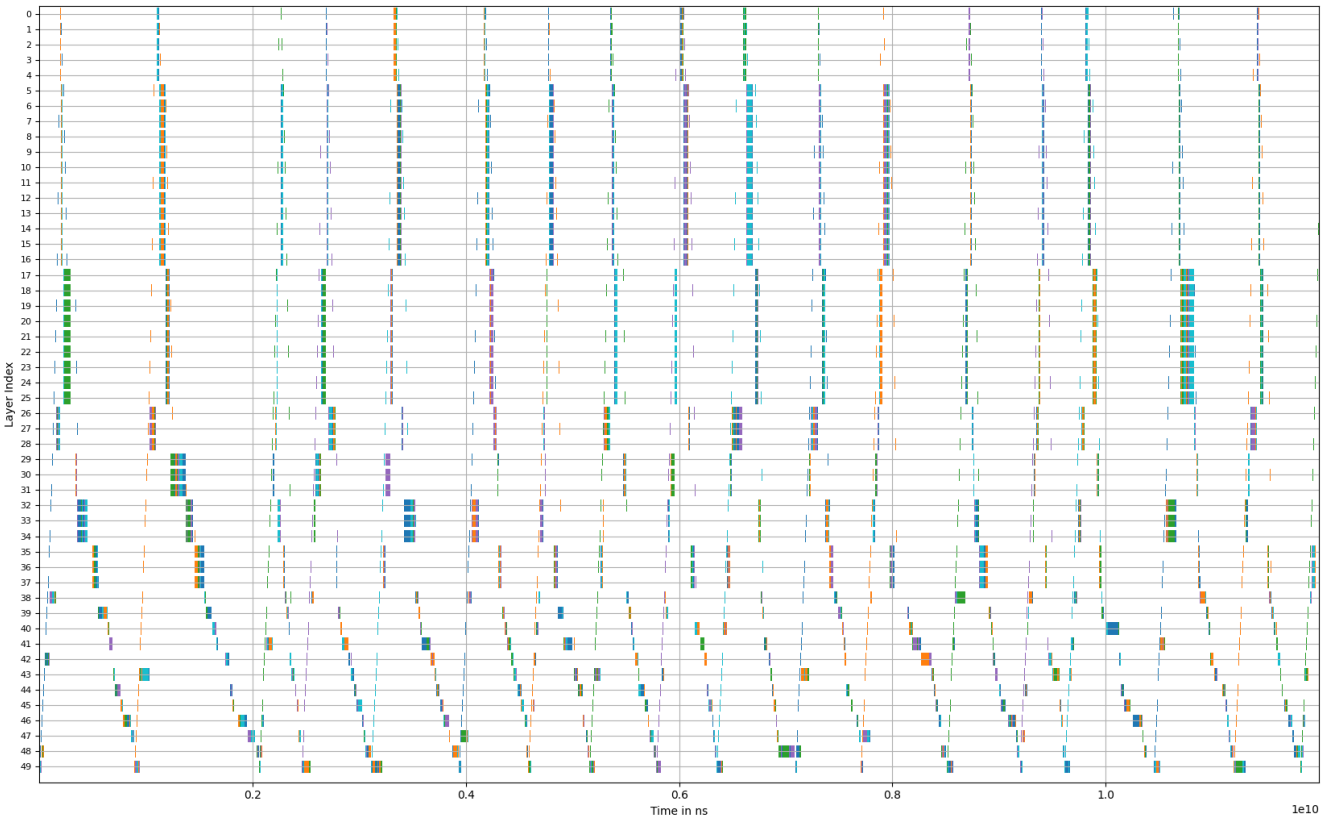

Figure 5.23: Horovod progression when Horovod is set at a lower priority than the background traffic with a compute-to-network ratio of 2 to 1 when the background traffic takes up about 12.5% of link bandwidth.

Chapter 6

Future Work

We have identified the following areas for future work.

- **Prioritization Schemes**: We began exploring linear programming to schedule the start time of each tensor fusion and the priority of each transfer given there is a high-confidence mapping between priority to network transfer time. We experimented with a small 5 layer DNN and 3 priority levels with pre-defined mapping to network transfer time and it led to promising results. However in later ns-3 testing with background flows and two priority bands, the duration samples of the same priority exhibited high variance which is challenging to be fit over a known distribution. Another promising scheme is applying adaptive prioritization on the fly based on the leftover network budget and how close the current stage is to the expected iteration deadline.
- **Queuing Disciplines**: Challenges described in the last item inspire us to look into other queuing disciplines or prioritization schemes that can offer more predictable network transfer time instead of relying on strict priority queues.
- **Transport Layer and Rate Control**: In our experiments we relied on using TCP sockets. It would be interesting to experiment with more specialized transport protocols such as DCTCP, D2TCP and QUIC [41]. It's also worth experimenting with various implicit and explicit rate control schemes such as HULL [6], D^3[50], and PDQ [24].
- **Slow Stragglers Effect**: Slow stragglers problems sometimes are inevitable due to various resource constraints (shared CPU, GPU or RAM). It is worth looking into how other workers in the ring can take advantage of knowledge of slow stragglers in other workers to adjust their own priority.
- **Asynchronous Gradient Aggregation**: On a broader note, we can examine the impact of various asynchronous gradient aggregation schemes on a shared network.

- **Impacts on 99th Percentile Flows**: We have focused on optimizing for the average flow completion time for small and large flows in this work. There are opportunities in optimizing the long tail (90th or 99th percentile) flows. Along the same direction, besides having web search as the main network resource competitor, we can explore other potentially latency-sensitive applications such as video/audio streaming where consistent latency is valued and measure the impact on domain-specific performance (jitter, QPS, etc).

- **Tensor Fusion and Partition**: Examine the impact of tensor fusion and tensor partitioning as well as varying the number of workers as they determine the smallest transfer size or possibly the smallest unit for preemption. For example, examine the impact of tagging tensors from the same layer with different priority that can introduce preemption inside the boundary of a RingAllreduce.

- **Network Topology**: In the future, we would like to experiment with more complex data center topology, for example the prevalent leaf-spine topology [4].

- **Other DDNN Architectures and Models**: Experiment with other DDNN architectures that are potentially more network constrained or in a centralized architecture. Experiment with off-the-shelf TensorFlow/PyTorch/MXNet configurations (no Horovod, keeping the inter-iteration barrier). These setups provide even less network slack to utilize and deprioritize.

- **Generic Data Flow Network Slack Problems**: Expand to more generic data flow network slack problems. For example, can we de-prioritize one or more non-competitive paths when handling a web request with multiple parallel background requests?

Chapter 7

Conclusion

Our work has uncovered a promising area that has not been examined before. Previous work focusing on optimizing distributed deep neural networks has not crossed the boundary to look at its impact on a multi-tenant environment, and thus missed the opportunity identified in our work. On the other hand, research on optimizing data center flows examined generic flows categorized by size or deadline, failing to take advantage of the domain-specific knowledge of DDNNs.

Our work is first of its kind to bridge the gap between both worlds to examine the impact that this increasingly popular tenant (DDNN) has on other applications in the data center. Through ns-3 simulation, we identified the boundaries of de-prioritizing network-optimized DDNN traffic in favor of latency-sensitive flows (web search) without compromising the performance of DDNNs. Specifically, we can expect to see a reduction in average flow-completion-time of small flows up to 136% and 66% when the link utilization of the background traffic is below 3.12%. As the background flows double to 6%, we can still expect as high as a 73% FCT reduction for small flows and 40% for large flows. This could enable higher utilization of machines by allowing latency-sensitive applications to use leftover CPU and RAM present on GPU training nodes that would otherwise have an unsuitably busy network. It also allows DDNN clusters to use the same network infrastructure as latency-sensitive applications without harming the performance of either. More importantly, we can achieve significant performance gains for latency-sensitive applications under identified operating boundaries, effectively making DDNN a friendlier tenant in data centers.

Bibliography

[1] Mpi: A message passing interface. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 878–883, 1993.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 19, USA, 2010. USENIX Association.

[4] M. Alizadeh and T. Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 71–74, 2013.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013.

[8] AWS. Amazon ec2 p3 instances, accelerate machine learning and high performance computing applications with powerful gpus. `https://aws.amazon.com/ec2/instance-types/p3/`. Online; accessed 2020-08-23.

[9] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), August 2019.

[10] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[11] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: design, analysis and applications. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1653–1664 vol. 3, 2005.

[12] 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, 2015. cite arxiv:1512.01274Comment: In Neural Information Processing Systems, Workshop on Machine Learning Systems, 2016.

[13] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[15] Andrew Gibiansky. Bringing hpc techniques to deep learning. `https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/`. Online; accessed 2020-08-20.

[16] Igor Gitman and Boris Ginsburg. Comparison of batch normalization and weight normalization algorithms for the large-scale image classification. *CoRR*, abs/1709.08145, 2017.

[17] Richard Graham, Timothy Woodall, and Jeffrey Squyres. Open mpi: A flexible high performance mpi. pages 228–239, 09 2005.

[18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[19] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 4854–4858. AAAI Press, 2017.

[20] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *SysML 2019*, Apr 2019.

[21] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, Brighten Godfrey, and Roy H. Campbell. Caramel: Accelerating decentralized distributed deep learning with computation scheduling. *ArXiv*, abs/2004.14020, 2020.

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

[23] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[24] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 127–138, New York, NY, USA, 2012. Association for Computing Machinery.

[25] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[26] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *CoRR*, abs/1905.03960, 2019.

[27] Simon Kassing. Networkload. `https://github.com/snkas/networkload.git`. Online; accessed 2020-08-20.

[28] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Sigcomm '15*, 2015.

[29] C. Li, Abdul Dakkak, Jinjun Xiong, W. Wei, Lingjie Xu, and W. Hwu. Xsp: Across-stack profiling and analysis of machine learning models on gpus. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 326–327, 2020.

[30] Peter R. Mattson, Christine Cheng, Cody A. Coleman, Greg Diamos, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David M. Brooks, Dehao Chen, Debojyoti Dutta,

Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David E. Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. Mlperf training benchmark. *ArXiv*, abs/1910.01500, 2020.

[31] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild! a lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 693–701, Red Hook, NY, USA, 2011. Curran Associates Inc.

[32] Cyprien Noel and Simon Osindero. Dogwild! – distributed hogwild for cpu & gpu. 2014.

[33] ns-3. ns-3 network simulator. `https://www.nsnam.org/`. Online; accessed: 2020-08-15.

[34] ns-3. pfifo_fast queue disc. `https://www.nsnam.org/docs/models/html/pfifo-fast.html`. Online; accessed 2020-08-20.

[35] ns-3. Traffic control layer. `https://www.nsnam.org/docs/models/html/traffic-control-layer.html`. Online; accessed 2020-08-20.

[36] NVIDIA. Nvidia collective communications library (nccl). `https://developer.nvidia.com/nccl`. Online; accessed 2020-08-20.

[37] Adam Paszke, S. Gross, Soumith Chintala, G. Chanan, E. Yang, Zachary Devito, Zeming Lin, Alban Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[39] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[40] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yi xin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[41] The Chromium Projects. Quic, a multiplexed stream transport over udp. `https://www.chromium.org/quic`. Online; accessed 2020-08-22.

[42] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, August 2011.

[43] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[44] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[45] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. *CoRR*, abs/1603.05201, 2016.

[46] K. Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2015.

[47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[48] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 115–126, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[50] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 50–61, New York, NY, USA, 2011. Association for Computing Machinery.

[51] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 139–150, New York, NY, USA, 2012. Association for Computing Machinery.