Horovod: Distributed Machine Learning

Dusan Zivanovic
Department of Computer Science
University of Cambridge
dz308@cam.ac.uk

Viktor Toth
Department of Computer Science
University of Cambridge
vt289@cam.ac.uk

Abstract

In recent years, neural networks have been successfully applied to a wide variety of tasks at industrial scale. However, properly training large models can be extremely computationally demanding. To power the learning process, it is commonplace to distribute training over a large number of computing devices, which train the model independently on portions of data and periodically synchronise computed gradients. This creates a need to efficiently synchronise the computations of several workers. This project aims to implement Data Parallel Distributed Training using the efficient, decentralised Horovod all-reduce algorithm, and explore related optimisations.

1 Introduction

Over the last decade, neural networks have gained ever increasing popularity in numerous machine learning domains. While they have proven to be highly versatile, the training process often requires immense computational power – often only feasible with several interconnected processing units. However, efficiently synchronising tasks running on multiple devices can be greatly challenging.

One prevalent paradigm in distributed neural networks is Data Parallel Distributed Training. It describes a mode of co-operation where the training set is divided among the workers. Each process performs the feed-forward and back-propagation steps independently to calculate the local gradients of the optimisation problem. Finally, we average over the local gradients to compute the global gradient that is then used by each worker to update their model.

Data Parallel Distributed Training requires sharing and averaging gradients across all workers (usually referred to as an all-reduce operation with averaging or summing as the reduction function). This is often done in one of two ways:

- Collecting local gradients in one or more central servers which compute the averages and send them back to the workers.
- Workers directly communicating and sharing results with each other in a decentralised manner.

While a centralised server model is often easier to implement, it introduces a significant bottleneck: all communications must go through the main process(es). Moreover, parameter servers also carry the burden of computing the reduction function. For this reason, recently there has been extensive research done in decentralised all-reduce algorithms [1].

One such distributed algorithm is Horovod [2], which aims to optimally utilise the bandwidth between nodes. It has been shown to offer considerably better performance than centralised methods and has recently been incorporated into both TensorFlow and PyTorch (arguably the two most popular deep learning frameworks).

The aim of the project is to provide an overview and a re-implementation of the core features of the Horovod algorithm. First, we will present the main ideas behind Horovod. Then we discuss our implementation in detail and the different design choices. Finally, we will benchmark the performance of our implementation for training GoogLeNet [3] to categorise images based on the ImageNet dataset [4].

2 Horovod

For a distributed learning system, one of the biggest bottlenecks is the bandwidth between devices. Since modern neural networks have dramatically increased in size, nowadays the gradient information computed at each worker can easily be a several gigabytes. While data centres often accommodate 10-200 Gb/s data transfers between devices, moving computed gradients still remain one of the most time consuming part of the training pipeline.

Therefore, for an effective distributed learning system, we need implement an all-reduce algorithm that aims to minimise the communication between workers. Horovod employs the ring all-reduce method, which provides a bandwidth-optimal solution to synchronisation [1].

2.1 Ring All-reduce

Let us assume that each worker w_i has independently calculated some result r_i for $i = 1, ..., n$ (where n is the total number of workers). In distributed learning r_i is usually the gradient of the model parameters calculated using the portion of the training data provided to that worker. The goal of the all-reduce operation is to distribute $f(r_1, r_2, ..., r_n)$ among all the workers, where f is the reduction function (in our case we perform summation to calculate the averages). We also make the assumption that f is both associative and commutative.

It can be proven that we need at least $2 \cdot (n - 1)$ pairwise communications to calculate and distribute the result of the all-reduce operation [1]. One possible algorithm for the lower limit is as follows:

1. Worker w_1 sends r_1 to worker w_2.
2. Worker w_2 calculates $f(r_1, r_2)$.
3. We continue this pattern: worker w_{i-1} sends $f(r_1, ..., r_{i-1})$ to worker w_i which calculates $f(r_1, ..., r_{i-1}, r_i)$ (for $i = 2, ..., n$).
4. Now that worker w_n has the collected results $f(r_1, ..., r_n)$, we can distribute it in a ring manner: first sending it to w_1, which will send it to w_2, and so on.

While this algorithm only requires the theoretically minimum number of pairwise communications, it only allows for one pairwise communication at a time and does not leverage the fact that in most environments workers can communicate simultaneously. The ring all-reduce algorithm improves on it by dividing the data into n segments and performs all-reduction on them simultaneously.

Similarly to the previous algorithm, ring all-reduce consists of two phases. In the first pass, we gather the reduced data on one device, and then the second pass distributes the reduced gradients. The algorithm is as follows (using cyclic indexing):

1. Each worker w_i divides the calculated data into n segments (denoted with $r_{i;1}, ..., r_{i;n}$).
2. At each iteration $t = 0, ..., n - 2$ each worker w_i sends $r'_{i;i-t}$ to w_{i+1} (where $r'_{i;j}$ denotes the partial reduction of all data segments $r_{w;j}$ already received by worker w_i).
3. At the end of the first pass, each worker w_i has the reduced value of segment $i + 1$, denoted with f_{i+1}.
4. We distribute the reduced gradients across all workers. At each iteration $t = 0, ..., n - 2$ each worker w_i sends f_{i+1-t} to w_{i+1}.

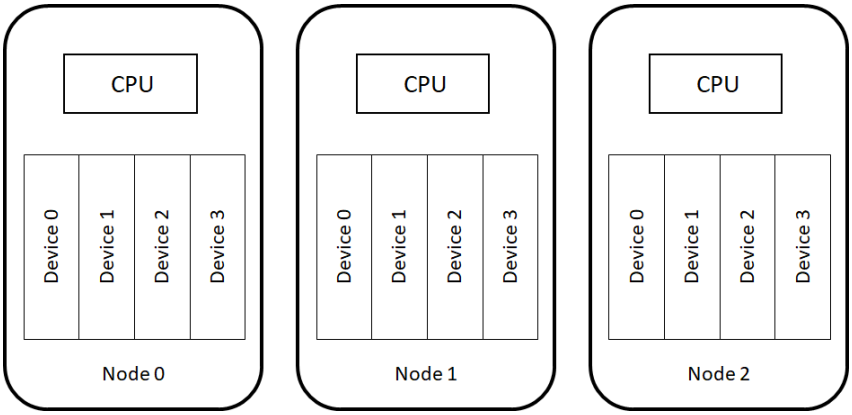

Figure 1: Computational model: Multiple connected computing nodes each equipped with a CPU and several devices.

A visual representation of the ring all-reduce method can be found in Appendix B.

A nice property of the all-reduce algorithm is that each worker sends data only to one other worker and, similarly, receives data from only one worker. Moreover, at every iteration each worker sends the same amount of data, hence the performance of the algorithm is only limited by the lowest bandwidth between two consecutive workers.

3 Implementation

The aim of this project was to implement a data parallel distributed neural network training system. In this section we, will discuss assumptions about the underlying computational model, arising design choices, optimisation techniques, and the frameworks and libraries we have chosen to use.

Data Parallel Distributed Training refers to a type of distributed learning where a number of workers execute the same training logic on separate partitions of the training dataset. To keep consistency between the different workers, we regularly synchronise the computed results.

A simplistic implementation augments the single-worker algorithm with only a single function call to average gradients across workers following back-propagation. To satisfy the requirements of the reduction function being associative and commutative (Section 2.1), averaging is often performed via summation. We first iterate over gradient buffers and invoke a distributed all-reduce function on each buffer with *sum* as the reduction function. Finally, the results in each buffer should be scaled to produce an average of the original workers' gradients.

3.1 PyTorch Distributed Message Passing

For our implementation, we have opted to use the PyTorch deep learning library [5] as it offers a robust and easily accessible distributed computing framework. PyTorch provides a universal abstract interface to several message passing libraries. Currently, it supports the Gloo and MPI backends for CPU-to-CPU communication as well as the NCCL backend for GPU-to-GPU communication. Capabilities of the framework include sending peer-to-peer messages[1], as well as collective function such as broadcast, scatter, gather, reduce and all-reduce functions. Since the objective of this project was to re-implement a Horovod style ring all-reduce, we only used PyTorch's distributed framework as a Python message passing library, and restricted ourselves to only using peer-to-peer functionalities.

3.2 Computational Model

In most modern data centres, the user has multiple computing nodes at their disposal. In our computational model, each node is equipped with a multicore CPU and AI accelerator devices – typically GPUs or TPUs. Figure 1 illustrates the model with 3 connected computing nodes. The nodes are interconnected with a data center network, such as Ethernet or InfiniBand, and the CPU and AI accelerators are connected with a high-throughput interconnect such as PCIe. This model lends itself to two different approaches based on which level the ring all-reduce operation is performed.

3.2.1 Node Reducer

The node reducer (or two-step reducer) exploits the fact that intra-node communications have a larger bandwidth compared to inter-node communications. It consists of two steps:

1. Each CPU collects the local gradients from its devices hence computing a node-reduced gradient.
2. CPUs perform the ring all-reduce operation and copy the collected global gradient to the devices.

This approach can be considered as a hybrid method between hierarchical and decentralised distributed learning and is especially useful when the gradients need to be moved to the CPU before being sent over the network. However, entrusting the CPUs with computing the node-reduced gradient – i.e. summing the gradients received from its devices – can create a bottleneck. Therefore, it is imperative to maintain a balance between the number of CPUs and the number of corresponding devices.

3.2.2 Device Reducer

The device reducer (or one-step reducer) performs ring all-reduce directly among devices. It aims to eliminate the possibility of a bottleneck forming around a single computing unit. It also delegates all tensor calculations to the AI accelerator devices, which are better suited to perform them. In this setup, the CPU is only responsible for facilitating network communications and feeding training data to its devices.

3.3 Optimisations

Previous work details multiple possible improvements to the distributed training pipeline. In this section, we present two extension and will later examine the effects they have on the overall performance.

3.3.1 Tensor Fusion

The authors of existing distributed learning packages observed that all-reduce has a higher throughput for larger buffers [2][6]. This is likely due to constant overhead costs of running all-reduce, including latency of communication between nodes. Tensor fusion optimisation merges small gradient tensors into batches before invoking all-reduce. Gradients are fused in the order in which they are reported by the network model. The assumption is that this will merge gradients from the same or adjacent layers together, therefore the whole bucket should become available soon after the first gradient is computed.

3.3.2 Pipelining

To fully leverage parallel computing, we should aim to overlap back-propagation and gradient synchronization. The pipelining optimisation begins synchronizing gradients as soon as they become available. This means that final-layer gradients (first to be computed) can be synchronised while the back-propagation is still being performed on earlier layers.

[1] PyTorch does not expose the peer-to-peer message passing functionalities of NCCL. This is circumvented by manually defining a computation group for each pair of consecutive devices and using the broadcast functionalities.

PyTorch's AutoGrad package provides a hook utility [7] used to define callbacks which are called when the gradient of a layer is computed. Our implementation registers hooks and keeps track of how many gradient tensors in a fusion batch have been computed. Once enough gradients have been collected, we mark the batch ready for all-reduce.

These optimisations have previously been implemented in Uber's Horovod [2] and PyTorch's `DistributedDataParallel` [6] distributed learning packages.

Finally, we have not implemented or evaluated the system using GPU RDMA (Remote Direct Memory Access). This technology allows network cards to directly access GPU memory, without being copied into main memory.

3.4 System Architecture

Our distributed learning system was in part built to evaluate different all-reduce algorithms and optimisation techniques. For this reason we split the system into specialised modules that can be combined into different configurations. A diagram of the components is presented in Figure 2. There are three component types: Distributed Model Wrappers (colored blue in the figure), Reducers (colored green) and all-reduce algorithms (colored yellow). Any Model Wrapper can be combined with any reducer and any all-reduce algorithm. The only exception is PyTorch's built in DistributedDataParallel which contains all the above functionalities in itself.

A Distributed Model Wrapper is a thin wrapper for user defined neural network models and its purpose is to submit fused batches of gradients to a reducer for averaging as soon as they become available. They provide a *sync_gradients* function which blocks until all gradients are reduced. The *P+F* wrapper implements overlapping back-propagation and gradient synchronization, and gradient fusion, there are also wrappers implementing one or none of these optimisations.

Reducers take gradient buffers from Model Wrappers and return references to the same buffers, now containing values of gradient averages across all workers. Two reducers have been implemented. The per-worker reducer immediately calls all-reduce on submitted buffers, and queues the buffers to be retrieved by the wrappers using the *get* function. The two-step reducer queues buffers to a different thread where buffers from all workers of this node are averaged before calling all-reduce.

Finally, the all-reduce algorithm to be used can be given as a parameter to reducers. Two have been implemented: a Horovod style ring all-reduce, and a parameter server like central all-reduce.

Figure 3 illustrates how gradients are synchronised using a two-step reducer and the Gloo MPI backend. First, PyTorch's Autograd framework computes a gradient and the hook callback is invoked. In step two, the hook notices that a whole batch of gradients is available and marks it as ready for synchronization by setting an event [2]. A separate thread goes through gradient batches, starting from the ones closest to the output. When a given batch becomes ready, the gradients are fused into a buffer in the main memory (step 3). In step 4 the fused tensor's handle is submitted to the reducer. A per-node thread collects tensors from workers, averages them, and invokes an all-reduce function (steps 5 and 6). Finally, a gradient retrieval thread in each worker retrieves tensors from the reducer and unfuses them into original gradient tensors (steps 7 and 8).

4 Evaluation

Our evaluation efforts were aimed at determining how well our implementation compares against PyTorch's `DistributedDataParallel`, whether two-step (node based) reduction performs better than one-step (device based), as well as determining how effective the two optimisation techniques were. In the case of tensor fusion, we were also interested in how large the fused tensors should be.

We evaluated the performance of the distributed training systems, by measuring the throughput of training the GoogLeNet [3] neural network designed for image classification. Training data was compiled by selecting 25 classes – each with 1000 samples – from the ImageNet [4] dataset. We also performed experiments with synthesised (random) inputs which omits the data loading step of the training pipeline. Performance of the learning systems is measured by the number of images processed in a second.

[2] https://docs.python.org/3/library/threading.html#threading.Event

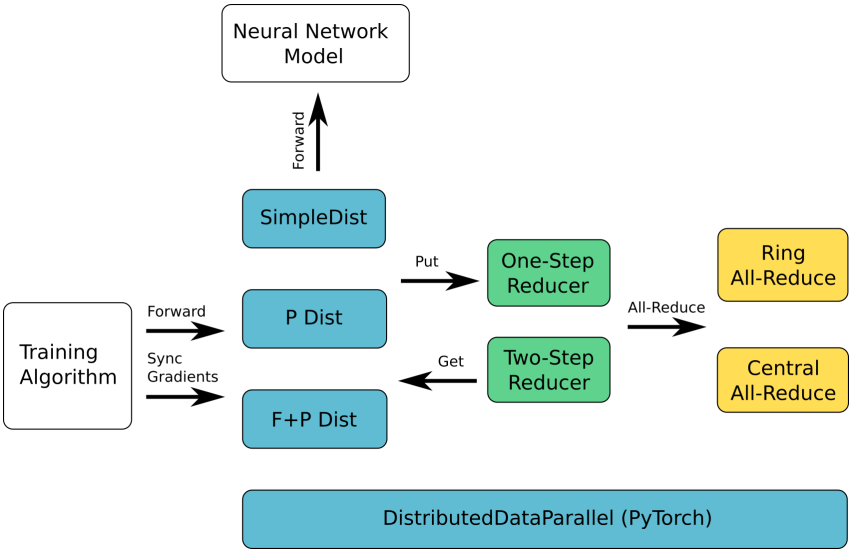

Figure 2: Overview of the modules of our distributed learning implementation. Different Model Wrappers (blue), reducers (green) and all-reduce algorithms (yellow) can be combined. DistributedDataParallel is a pre-existing part of PyTorch.

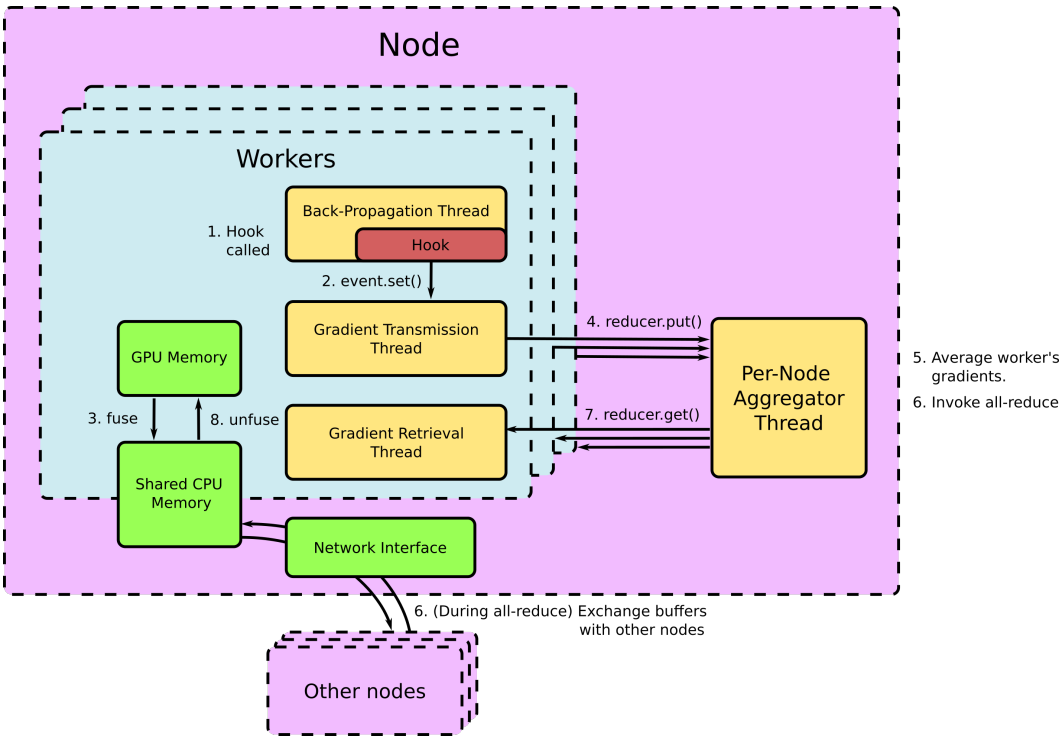

Figure 3: Synchronization of gradient tensors using a two-step reducer, beginning with a hook being invoked when the gradient is computed and ending with a synchronised batch tensor being unfused into individual global gradient tensors on the GPU.

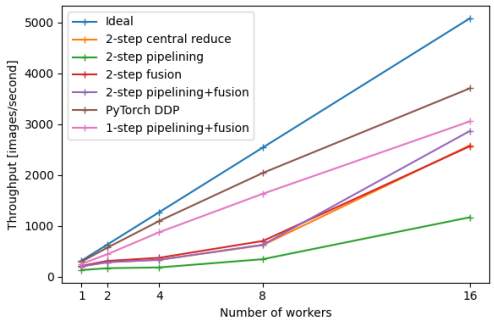

Figure 4: Scaling of throughput of training using synthesised data for different configurations.

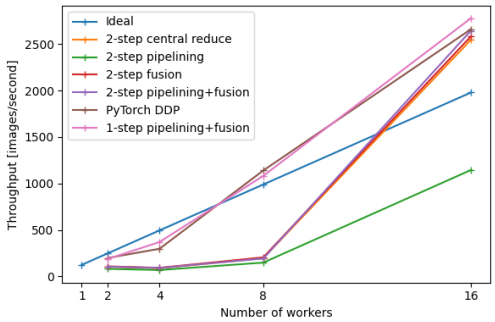

Figure 5: Scaling of throughput of training using real data for different configurations.

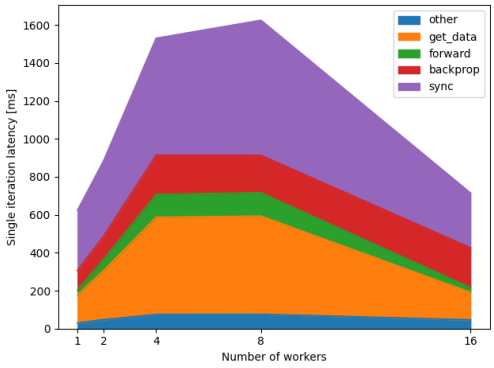

Figure 6: Latency breakdown for the 2-step P+F configuration when using synthesised data

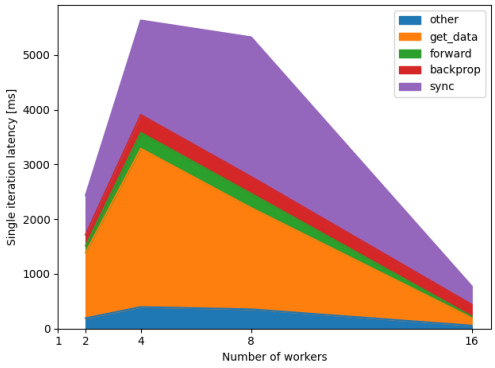

Figure 7: Latency breakdown for the 2-step P+F configuration when loading data from a remote data store.

4.1 Cambridge HPC cluster

For our measurements, we have used the University of Cambridge's High Performance Computing (HPC) Wilkes2 cluster. This network is equipped with nodes containing one Intel Xeon E5-2650 v4 2.2GHz 12-core processor, four Nvidia P100 (16BiB) GPUs, and 96 GiB main memory. We have conducted experiments with 1, 2, and 4 GPUs on one single node as well as 8 and 16 GPUs on 2 and 4 nodes respectively.

In our first experiment, we compared the performances of different setups using synthesised (random) data (Figure 4) and ImageNet images (Figure 5) as input for the neural networks. The figures show how the throughput of training images scales with the number of GPUs for different distributed training configurations. The "Ideal" scaling (shown in blue) is simulated by multiplying single GPU performance with the number of GPUs. We also measured the latency breakdown of our most complex configuration, the "2-step P+F". Results are shown in Figure 6 and Figure 7 respectively.

For synthesised data, we can see that "DDP" and our best configuration, namely "1-step p+f" have comparable performance, with "1-step p+f" having 18% lower throughput at 16 GPUs. Both configurations are significantly worse than theoretical performance, with DDP achieving 72% and "1-step p+f" 60% of the theoretical performance. Furthermore, the poor performance of "2-step p" shows that the most important optimisation is tensor fusion, since this configuration only achieves 23% of the theoretical performance at 16 GPUs. Additionally, all 2-step configurations perform noticeably worse, especially with 8 or fewer GPUs. This might be because on a single node ring reduce parallelises gradient averaging, whereas 2-step reduce sums all gradients in a single thread. Finally, the improvements brought by overlapping back-propagation with synchronization were not observed.

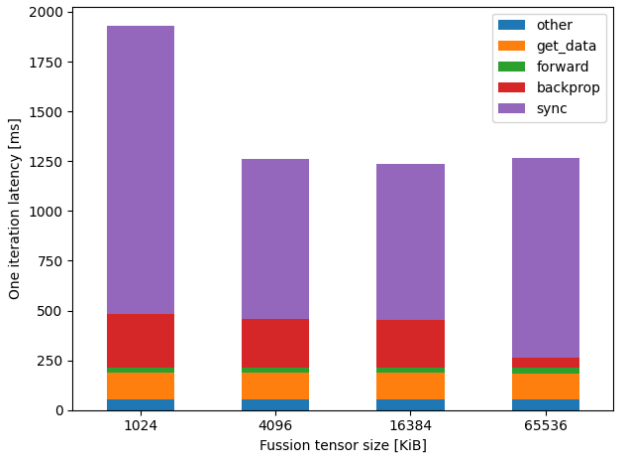

Figure 8: Single training iteration (batch) latency for the 2-step P+F configuration, with real training data, as a function of fusion buffer size.

Figures 5 and 7 show the throughputs we measured when loading real data from a remote file store, and the latency breakdown for the 2-step P+F configuration. Achieved throughputs are (somewhat peculiarly) over-perform the ideally scaled throughputs. Looking at the latency breakdown in Figure 6 provides an explanation. Jobs with fewer workers, including the 1-GPU Ideal job, spend much more time loading data. This is due to the fact that increasing the number of worker processes lowers the per-worker portion of the training dataset, and for 16 workers these parts completely fit into the workers cache. Therefore, with more workers the penalty for retrieving remote data is not paid. We recognise that this experiment is peculiar because of the remote data access and the artifacts of caching, however, we include it as an interesting case study. The experiment with synthesised data is more representative of the actual performances.

To investigate the effect of the tensor fusion batch sizes, we have performed charted the single iteration latencies using the "2-step P+F" configuration at 16 GPUs with real training data (see Figure 8). Latency is measured for loading data, forward and backward propagation, and synchronization. As the graph clearly shows, for this particular neural network the optimal fusion tensor size is between 4 and 64 MB, with significantly worse performance for smaller fusion batch sizes.

Finally, Figure 6 shows that even with synthesised data, the input generation, and synchronisation phases are slower with 4 and 8 than with 2 and 16 workers. This contributes to 2-step methods performing poorly with 4 and 8 workers. Given that these issue only impact 2-step configurations, a possible, but not certain explanation is that workers are bottlenecking the CPU resources. This could be because 1-step configurations use four MPI tasks per node whereas 2-step configurations use a single MPI task and manually spawn additional threads. Even though we explicitly requested more CPU cores per task for 2-step runs, this difference could have resulted in a different allocation of resources.

5 Conclusion

As neural network models rapidly increase in size, so does the need for distributed approaches to training. In this project, we have provided an implementation for a Horovod-style distributed learning system using the PyTorch framework. We have presented that our implementation is close to existing state-of-the art solutions as well as investigated the performance impact of different approaches and optimisation techniques. We have shown that the one-step (device-level) tends to outperform the two-step (node-level) variant and illustrated the importance of the tensor-fusion method.

References

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

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

[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

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

[5] 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 8024–8035. Curran Associates, Inc., 2019.

[6] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[7] 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.

A Accessing the Codebase

Our implementation can be accessed on GitHub.

`https://github.com/zdule/distributed_learning`

Description and usage information can be found in the repository's readme file.

B Graphical Representation of Ring All-reduce

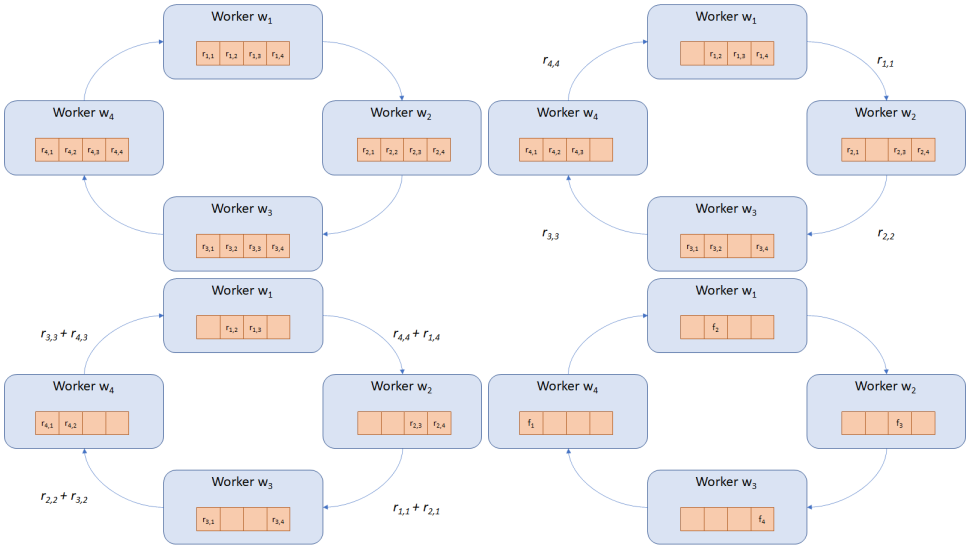

Figure 9: The first pass of the ring all-reduce algorithm

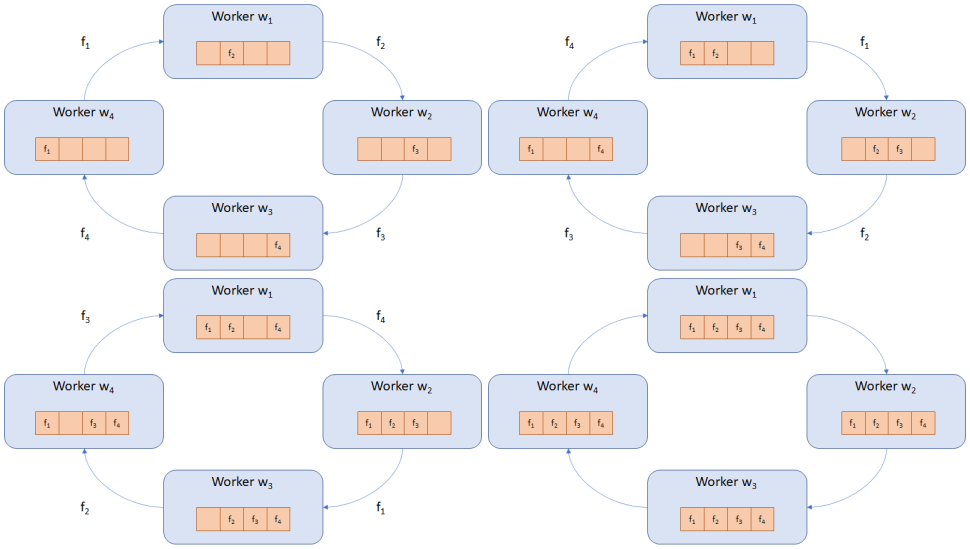

Figure 10: The second pass of the ring all-reduce algorithm

C Acknowledging CSD3

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (`www.csd3.cam.ac.uk`), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (`www.dirac.ac.uk`).