Distributed deep neural network training: A measurement study

Forrest Iandola
UC Berkeley
forresti@eecs.berkeley.edu

1. Introduction and Motivation

Deep neural networks (DNNs) have become a staple of state-of-art approaches to a broad variety of machine learning problems in areas such as computer vision, speech recognition, and text analysis. However, the process of training (i.e. converting a collection of data into a machine-learned model) deep neural networks is much more computationally expensive than previous approaches to computer vision. As a result, researchers often wait for weeks or months for models to train. Researchers often explore a broad variety of DNN models and configurations, so long training times impede research progress.

Accelerating DNN training promises to enable at least two major changes to the workflows of DNN researchers, and to DNN research as a while: First, with sufficiently fast DNN training, researchers and developers could train models on today's datasets in *interactive* timescales, rather than weeks or "summer vacation" timescales. Second, acceleration of DNN training could be leveraged to ingest more training data in a given number of hours, often leading to models that are more robust and are more accurate when deployed to classify data.

In our recent paper, *FireCaffe* [9], we addressed this problem by scaling the training of the popular GoogLeNet [16] DNN model over a cluster of GPU-enabled servers. In that work, we reported a 47x speedup (from 3 weeks to 10.5 hours) on 128 GPUs, compared to a well-tuned single-GPU baseline. Beyond 128 GPUs, we observed diminishing returns – e.g. only a 60x speedup on 256 GPUs. On 128 GPUs, we found that approximately half of the execution time is spent communicating among servers. On 256 GPUs, substantially more than half of the execution time is spent on communication. What can be done to alleviate this bottleneck to further scale and accelerate DNN training? On a fixed hardware platform, there are two main options:

- Option 1. find a way to communicate data faster (e.g. by modifying the implementation), or
- Option 2. find a way to communicate less data (e.g. by modifying the algorithm)

In the *netmap* paper, Rizzo found that the off-the-shelf FreeBSD software interface to the network card is so inefficient that it transmits data 15x slower than the contemporary line rate [15]. Rizzo addressed this with *netmap*, which has a collection of software optimizations that bring the transmission rate up to the peak line rate. Similarly, a naive implementation of a network switch on commodity servers is quite inefficient (substantially slower than line rate), and the RouteBricks authors performed a substantial amount of software tuning before their system was able to approach line rate [7]. Like the baselines in the Netmap and RouteBricks papers, could FireCaffe be suffering from a similarly inefficient communication software implementation? Or, put an other way, what is FireCaffe's network transmission efficiency with respect to the maximum achievable line rate? Answering this question will motivate further research on one of the following:

- If FireCaffe's communication is an order of magnitude slower than line rate, there is probably low-hanging fruit to be obtained by tuning the communication implementation (Option 1).
- Alternatively, if we discover that FireCaffe's communication is approaching the peak line rate, then rethinking the DNN training algorithm to communicate less is the next problem to target (Option 2).

The rest of this paper is organized as follows. In Section 2, we introduce some preliminaries and terminology. We review some of the related work in distributed DNN training in Section 3. In Section 4, we describe four different strategies for implementing the communication needed for distributed DNN training, and we assess the tradeoffs in execution time. In Section 5, we describe the hardware platform that we use for evaluating distributed DNN training, and its communication overhead. In Section 6, we conduct our study of the theoretical and measured runtime of the communication in distributed DNN training. We conclude in Section 7.

2. Terminology

Deep neural network (DNN): A family of machine learning algorithms that unfortunately have the word "network" in the name. Ignore the word "network;" a DNN is purely a software construct. That said, the DNN does indeed have its own computational topology, which can be tuned or rearchitected in concert with other aspects of the system.

Network hardware: Instead of saying "network," we do our best to say "network hardware" to avoid confusion with the term mentioned above.

Bytes vs. bits: To keep things simple, we report all bandwidth numbers in terms of Bytes per second. Read "GB" as Gbytes.

3. Related work

3.1. Algorithmic approaches to accelerating DNN training

When performing distributed DNN training, it is necessary to communicate information across servers. There are several ways to slice the problem up, including pipeline parallelism, sub-image parallelism [3], model parallelism [4], and data parallelism [9]. We offer a more comprehensive review of the tradeoffs among these approaches in the FireCaffe paper [9]. But, briefly, data parallelism requires 360x less communication across servers than model parallelism when training the popular GoogLeNet [16] DNN architecture for computer vision. If we have to pick only one type of parallelism, data parallelism is by far the biggest parallelism opportunity for computer vision applications of DNNs.

Data parallelism consists of giving a subset of the training data to each worker in a compute cluster (see Figure 1). At the beginning of each iteration of training, each server is dealt a subset of the current batch of training data. Each server computes a forward and backward pass on the data, yielding gradient updates (one gradient update for each model parameter). Even on a single-server implementation, it is standard to sum the gradient updates over all data samples in the batch. In a distributed data-parallel implementation, each server sums the gradient updates over its subset of the batch, and then the gradient updates are summed over across servers' subsets of the training data batch. Finally, the summed gradients are collected on all servers, which use the gradients to update their local copy of the model. In synchronous data parallelism, after each iteration of training, all servers have an identical model. In terms of data movement, the communication that we have described consists of summing equal-length vectors of gradient updates from multiple servers, and distributing the results back to the servers. The high-performance computing community has a word for this type of communication: *allreduce*.

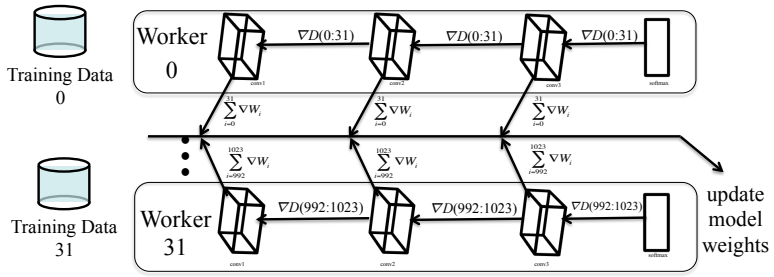

Figure 1. Data-parallel DNN training.

3.2. Current state-of-art

Recent results distributed computer vision DNN training results Baidu [18], NVIDIA [5], and Google [2] use synchronous data parallelism, as described in the previous section. All of these deliver respectable scaling results up to 64 GPUs, beyond which communication across servers dominates the execution time. These are built around collective communication implementations of allreduce, such as we will discuss in the later subsections of Section 4.

In the "early" days of distributed deep learning, parameter-server implementations of synchronous or asynchronous communication were more prevalent – such as those implemented at Microsoft [3] and Google [6], as well as SparkNet [13]. For reasons that will become apparent in the next section, parameter servers appear to be falling out of favor.

4. Communication patterns in synchronous distributed DNN training

Math notation:

BW = Bandwidth in terms of *bytes/sec*
latency is in terms of *seconds to send one byte*
p = Number of servers

n = Message size contributed by *each server* in the allreduce. All servers contribute the same message size, dictated by the number of parameters in the DNN model that we are training.

4.1. Parameter Server Allreduce

A parameter server can be used to implement the allreduce operation in synchronous data-parallel DNN training as follows. We appoint one server as the "parameter server," while all other servers are "workers." In data-parallel training, each worker gets a subset of each batch of training data. To begin an iteration of training, each worker applies its local copy of the DNN model to run forward- and backward-propagation its subset of the training data batch. Then, the workers all send their model updates to the central parameter server, which sums up the updates, and then sends the summed updates back to the workers. Finally, the workers use the information received from the parameter server to update their local copies of the DNN model. After each iteration, all workers have an identical model. We illustrate parameter server communication in Figure 2.

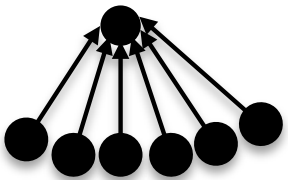

Figure 2. **Parameter server** implementation of the *reduce* communication pattern. All workers send updates to a central parameter server.

What is the execution time of performing an allreduce communication using a parameter server approach? Consider the case where there are 8 servers, all with same hardware architecture. Server 0 is the parameter server, and servers 1-7 are workers. When server 1 sends updates to the parameter server at the peak line rate, it fully saturates its own network card and the parameter server's network card. Since the parameter server's network capacity can be fully saturated by receiving from one server at a time, the communications from workers to the parameter server are *serialized*. The execution time of an allreduce operation can be calculated as a function of the network hardware's bandwidth and latency. Since the communications are serialized, the bandwidth term is $\frac{2p*n}{BW}$, and the factor of 2 is due to the fact that the parameter has to receive updates from all workers, and then send the summed updates to all workers. The latency term, due to the serialized communication, is $2p * latency$. We put all of this together in the following equation to calculate best-case execution time for allreduce implemented with a parameter server.

$$T_{param\_server} = \frac{2p * n}{BW} + 2p * latency \tag{1}$$

Note that p is the number of workers, not counting the parameter server. We derived this equation from first principles. Somewhat surprisingly, we have not been able to find a reference that writes down the overhead of synchronous allreduce communication using a parameter server.

As we increase the number of workers, the parameter server becomes a bigger bottleneck in application execution time. One approach to address bottleneck this is to appoint multiple servers as parameter servers. Each parameter server would be responsible for communicating updates with a subset of the workers. The pool of parameter servers can be implemented hierarchically, where the servers aggregate gradients in a tree-like fashion, as is proposed by Mai *et al.* [11]. In the next section we will take this idea to its logical conclusion, where all servers are involved in computation (like the "workers" in the parameter server scheme), and all servers work collaboratively to perform communication.

4.2. Binomial Tree Allreduce

In the parameter server approach to allreduce communication, serialized communication from all workers to the parameter server's network card creates a bottleneck. We now describe the strategy of implementing allreduce as a *binomial tree*, which enables many network cards to make progress concurrently. In the binomial tree strategy, every server participates in computation (like the "workers" in the parameter server approach), and the servers all collaborate on communicating updates. Let us organize the servers in a binomial tree.[1] In this scheme, each non-leaf server s receives communications from two children. Server s sums the data from its two children, and then sends the summed result to its parent; we illustrate this in Figure 3. To distribute the summed results from the root to the other servers, we simply send the results back down the tree.

What is the execution time of binomial tree allreduce, for a given number of servers p, message size per server n, bandwidth BW, and latency? Recall in the parameter server approach that the parameter server had to send and receive p messages of size

[1]The hardware network topology does not need to be a binomial tree. Binomial tree communication can run on various hardware network topologies, such as the full-crossbar that we will describe in Section 5.

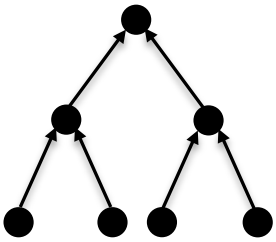

Figure 3. *Reduce* communication pattern implemented as a **binomial tree**. The arrows only show communication up the tree. For the *allreduce* communication pattern, the communication goes up to the root, and then back down the tree.

n. However, in this binomial tree approach, each server's network card sends and receives no more than 2 messages (and the leaves only send and receive one message). In addition, the height of a binomial tree with p nodes is $lg(p)$, so there are $lg(p)$ steps. Taking the height of the tree and the number of serialized messages per step in the tree into account, the bandwidth term is $\frac{2lg(p)*n}{BW}$ and the latency term is $2lg(p) * latency$. We put this together to calculate the best-case execution time of binomial tree allreduce in the following equation.

$$T_{binomial\_tree} = \frac{2lg(p) * n}{BW} + 2lg(p) * latency \tag{2}$$

We derived this equation from first-principles, and it matches the analysis by Hartmann *et al*. on binomial tree allreduce overhead [8].

Let us take a moment to contrast the parameter server best-case execution time (Equation 1) with binomial tree (Equation 2). In the parameter server approach, the execution time slows down linearly as we increase the number of servers. This is a big problem because (for a fixed problem size – i.e. *strong scaling*), doubling the number of servers also decreases the quantity of computation per server by 2x. So, with the parameter server, doubling the number of servers (workers) leads to half as much computation but double the time spent on communication. In contrast to the parameter server's linear increase in communication time w.r.t. the number of workers, the binomial tree approach experiences a *logarithmic* increase in communication time as we increase the number of workers. As you can see, the binomial tree scales much more efficiently than the parameter server approach to implementing allreduce communication.

4.3. Ring Allreduce

An other approach to implementing allreduce is to communicate updates in a *ring* fashion. As in the binomial tree approach from the previous subsection, all servers in the ring approach participate in both computation and collaborative computation. In this approach, we organize the servers into a ring, where each server has a `prev` and `next` server.[2] As in the previous sections, each server produces a DNN update vector of length n. But, to enable all network cards to work concurrently, each server breaks its length n vector into p messages that are each of length $\frac{n}{p}$. In each *step*, all servers send a length $\frac{n}{p}$ message to their respective `next` server. Within a step, all servers send a unique portion (range of indices) of their vector. We show an example with 4 servers in Figure 4.

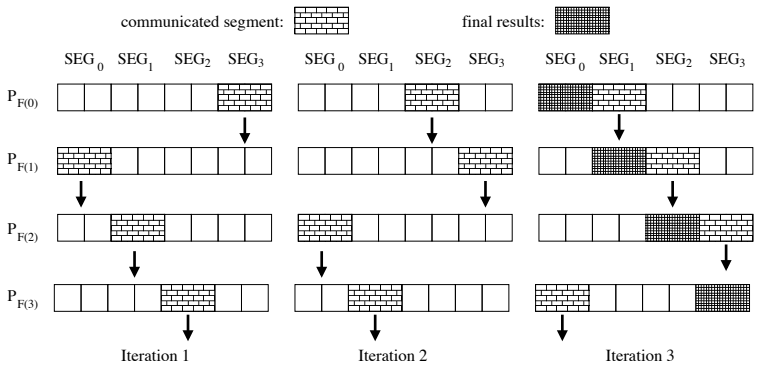

Figure 4. *Reduce-scatter* communication pattern implemented as a **ring**. Figure borrowed from [14].

[2]The ring allreduce approach is compatible with various network hardware configurations, and it does not require the network hardware topology to be a ring.

What is the execution time of ring allreduce, for a given number of servers p, message size per server n, bandwidth BW, and latency? Patarasuk and Yuan [14] offer a well-organized discussion of this, so we summarize their analysis as follows. Patarasuk and Yuan divide their analysis into two pieces – *reduce-scatter* (that is, for each index of the length n vector, the summed result is resident on at least 1 server), followed by allgather (getting a full copy of the summed vector onto all servers). To complete a reduce-scatter, $(p-1)$ steps are required, and in each step all servers send and receive a message of length $\frac{n}{p}$. The allgather also requires $(p-1)$ steps, and again in each step all servers send and receive a message of length $\frac{n}{p}$. Over all $2(p-1)$ steps, the total volume of data sent by each server over all steps is slightly less than $2pn$; more precisely $\frac{2(p-1)*n}{p}$. Assuming full-duplex communication, the bandwidth term is $\frac{2(p-1)*n}{p*BW}$, which is less than the parameter server and binomial tree approaches. However, the ring latency term is larger than the binomial tree latency term, because the ring sends many tiny messages. Adding up the total number of messages sent per server, the ring latency term is $2(p-1)*latency$, which is approximately as bad as the parameter server's latency term. This presents an interesting tradeoff: the ring approach is a good choice when using hardware that provides low latency and high bandwidth, but on heavily latency-limited hardware, the binomial tree may execute faster. We summarize this analysis in the following equation for the execution time of the ring allreduce communication.

$$T_{ring} = \frac{2(p-1)*n}{p*BW} + 2(p-1)*latency \tag{3}$$

4.4. Butterfly Allreduce

In the allreduce approaches that we have discussed so far, we have had to choose between minimizing the bandwidth term (ring) or minimizing the latency term (binomial tree). We now describe the *butterfly* approach to allreduce communication. Butterfly offers the best of both worlds – the small bandwidth term from the ring allreduce approach, and the small latency term from the binomial tree approach. There are a number of allreduce approaches that could be described as "butterfly," but here we focus on the recursive-halving butterfly approach described by Thakur *et al*. [17]. In their landmark paper, Thakur *et al*. require several pages worth of backstory to fully describe the recursive-halving butterfly approach. See Figure 5 for a visual overview of how the butterfly approach works. These are the key things to understand the butterfly approach:

- In the first step, the messages are of size $\frac{n}{2}$, and each server of index i communicates with the server whose index is $\frac{p}{2}$ away
- In the second step, the messages are of size $\frac{n}{4}$, and each server of index i communicates with the server whose index is $\frac{p}{4}$ away
- ...
- In the final step, the messages are of size 1, and each server of index i communicates with its immediate neighbor.

At this point, we have completed a reduce-scatter (which we defined in the previous subsection). The same quantity of communication needs to be performed a second time to perform the allgather (this time with recursive-doubling instead of recursive-halving), after which the allreduce is complete.

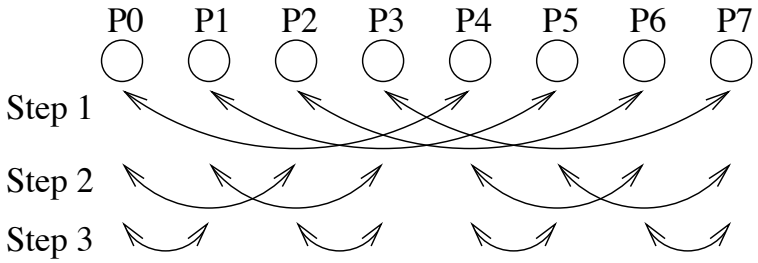

Figure 5. *Reduce-scatter* communication pattern implemented as a **butterfly**. Figure borrowed from [17].

What is the execution time of butterfly allreduce, for a given number of servers p, message size per server n, bandwidth BW, and latency? Let us begin by analyzing the butterfly reduce-scatter operation.[3] Since the server index offsets in each step are $\frac{p}{2}$, $\frac{p}{4}$, ..., 1, there are $lg(2)$ steps, so there is a latency cost of $lg(p)*latency$. As in the ring approach presented earlier,

[3]The following analysis assumes that the number of servers p is a power-of-two. Non-power-of-two sizes can also be implemented efficiently, but the analysis of execution time becomes slightly more complex.

the butterfly reduce-scatter requires each server to send and receive just $\frac{(p-1)*n}{p}$ bytes, so the reduce-scatter bandwidth cost is $\frac{(p-1)*n}{p*BW}$. The recursive-doubling butterfly allgather has the same number of steps and the same progression of message sizes (though done in reverse order), so we can simply double the bandwidth and latency terms to calculate the total execution time for the butterfly allreduce. We summarize this in the following equation to compute the best-case execution time of butterfly allreduce.

$$T_{butterfly} = \frac{2(p-1)*n}{p*BW} + 2lg(p)*latency \tag{4}$$

4.5. Theoretical Comparison of Allreduce Approaches

To conclude this section, we summarize what we have learned so far about allreduce approaches (see Table 1). First, parameter servers create a bottleneck and are less efficient at synchronous allreduce than all other approaches that we have considered. Next, binomial tree allreduce has bandwidth and latency terms that grow logarithmically with scale, which enables more efficient scaling than the linearly-increasing parameter server communication time. Alternatively, the ring approach requires less bandwidth than the binomial tree, but the ring approach pays a higher cost in the latency term. Finally, at least in this theoretical analysis, the butterfly approach offers the best-of-both-worlds: the small bandwidth term of the ring approach, and the small latency term of the binomial tree approach. With this in mind, from now on in this paper, we focus on the butterfly allreduce approach.

Table 1. Summarizing theoretical overheads in algorithms for implementing the Allreduce communication pattern. Terms in **bold** are the best (i.e. lowest-overhead) known solutions.

Algorithm	Bandwidth term (smaller is better)	Latency term (smaller is better)
parameter server	$2p*n$	$2p$
binomial tree [8]	$2lg(p)*n$	$\mathbf{2lg(p)}$
ring [14]	$\mathbf{\frac{2(p-1)*n}{p}}$	$2(p-1)$
butterfly [17]	$\mathbf{\frac{2(p-1)*n}{p}}$	$\mathbf{2lg(p)}$

5. Experimental testbed

The math in Equations 1-4 assume that links between any two servers are equally fast. We want to put our best foot forward on comparing experimental execution time to theoretical execution time, so we have selected a hardware platform that meets this requirement. We run our experiments on the FireBox-0 cluster in the UC Berkeley Aspire Lab. This cluster is a rack containing 16 compute nodes, connected with a top-of-rack switch. The switch is a Mellanox SX6036 Infiniband switch, which is a full-crossbar. Mellanox claims that all ports can communicate concurrently at 7 GB/s [12].

In the context of a larger datacenter, it's relatively common to have numerous racks like ours. Within a rack, servers are connected by a top-of-rack switches. Inter-rack connections are commonly made by connecting the top-of-rack switches to one or more additional switches, possibly in the form of a fat tree.

In our cluster, each server has a GPU. When executing FireCaffe, we perform the computation on the GPUs, and we use the Infiniband fabric to communicate among servers. The GPUs are connected to the Infiniband cards (without going through the CPU) by PLXLink PCIe hubs. The PCIe bus is offers 14 GB/s communication, and NVIDIA has released even higher intra-server communication in the form of NVLink. Thus, the main bottleneck in our communication stack is the communication of data across multiple servers, and we focus our benchmarking and analysis on this aspect of the communication.

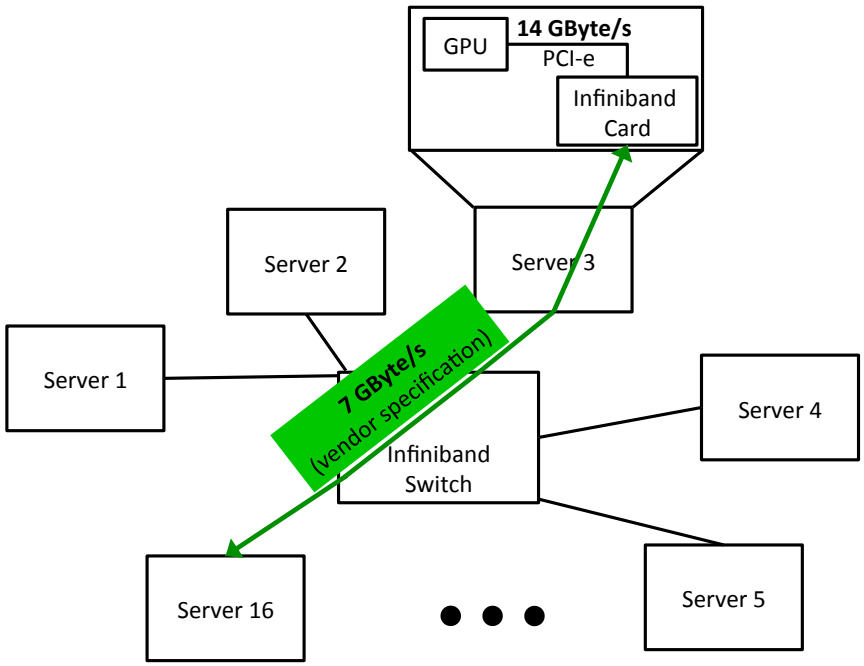

Figure 6. Communication fabric of the *Firebox-0* compute cluster that we use for our experiments in Section 6. Internally, each server follows the diagram shown for Server 3. The Infiniband switch is a full crossbar; Mellanox claims that all ports can run at 7 GB/s concurrently [12].

6. Measurement study

Now that we have a theoretical understanding of allreduce and its overhead, we now experimentally evaluate how close we are to the theoretical performance limits. Recall from Equations 1-4 that best-case allreduce overhead can be calculated as a function of the network hardware's *bandwidth* and *latency*. With this in mind, we proceed by measuring the line rate (i.e. bandwidth and latency) in Section 6.1. Then, we use these bandwidth and latency measurements to form a theoretical best-case execution time for the butterfly allreduce algorithm. Finally, we are have all the pieces in place to provide measurements and analysis of the butterfly allreduce execution time, and we present the theoretical and measured results in Section 6.2.

6.1. Measuring the line rate

To establish a best-case execution time for butterfly allreduce, we must first measure the bandwidth and latency of the network fabric in our hardware testbed. Mellanox claims peer-to-peer communication bandwidth 7 GB/s (56 Gbit/s) in the Mellanox SX6036 Infiniband switch that serves as the top-of-rack switch in our compute cluster [12]. Furthermore, the switch is a full-crossbar, so multiple peer-to-peer communications can take place concurrently without performance degradation.

So, can this Mellanox Infiniband switch actually deliver 7 GB/s? To find out, we simply measure the execution time of send/receive calls between two servers. In these measurements, we sweep over power-of-two message sizes, and we show the results in Figure 7. Observe that the latency – defined as the time to transmit a 1-byte message – is 2.5 microseconds. Up to a point, larger messages are communicated with higher bandwidth – that is to say, small messages are dominated by latency, and large messages are dominated by bandwidth. And, at a message size of 1MB, the bandwidth tapers off at slightly less than 6 GB/s, and we have found no message size for which communication executes faster than 6 GB/s. Mellanox claims that its switch can deliver 7 GB/s of bandwidth, and we have measured 6 GB/s. We are transmitting a peer-to-peer messages at 85% of the vendor's claimed line rate – this is close enough to the manufacturer specification that we are satisfied to move on.

In some papers, the authors would include error bars on a graph such as Figure 7. However, with repeated runs, the variance in these experiments is less than 1% – this is unsurprising since there are no other tenants performing communication on this datacenter network.

6.2. Allreduce benchmarks

6.2.1 Theoretical best-case Butterfly Allreduce speed

Now that we have measured the bandwidth (6 GB/s) and latency (2.5 microseconds) of our network fabric, we can pinpoint the theoretical best-case execution speed for the butterfly allreduce algorithm. Given a per-server message size n and a number

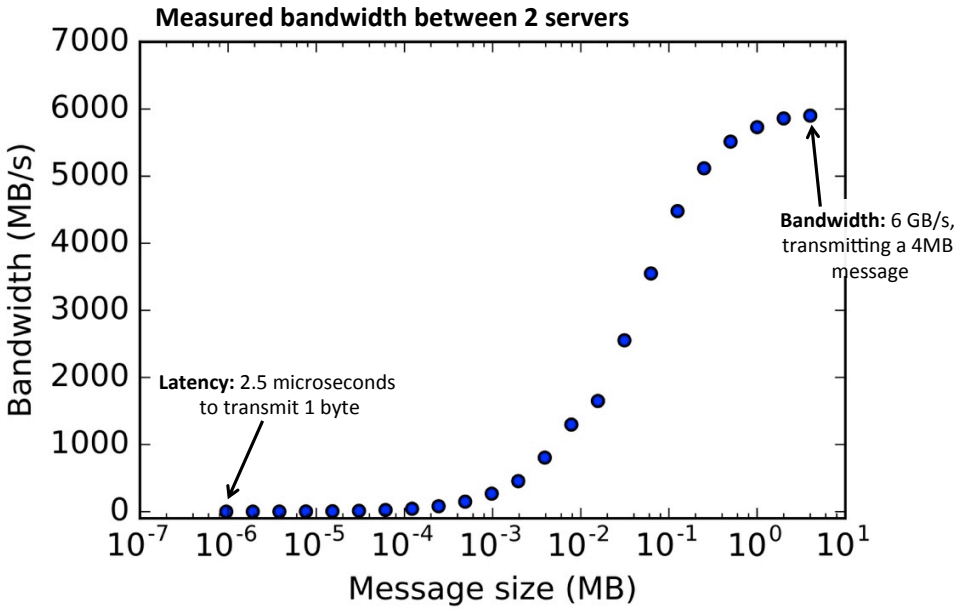

Figure 7. Measurement of line rate between 2 servers in our cluster. These peer-to-peer bandwidth and latency measurements are used for calculating the theoretical peak allreduce bandwidth in Section 6.2. The vendor (Mellanox) claims 7 GB/s of bandwidth, so observing 6 GB/s is reasonable.

of servers p, we can simply plug 6 GB/s of bandwidth and 2.5 microseconds of latency into Equation 4, and this gives us the best-case execution time of butterfly allreduce on our compute cluster.

6.2.2 Empirical measurements of Butterfly Allreduce speed

Is the theoretical best-case execution time achievable with a real implementation of the butterfly allreduce algorithm? A good way to assess this is to measure the execution time of a well-tuned implementation of butterfly allreduce. The authors of the butterfly allreduce study [17] (discussed in Section 4.4) released their code as part of the MPICH [1] library.[4] We measured this implementation on our compute cluster, with varying numbers of servers in Figure 8. As we increase the scale (number of servers), there are two key factors in play: (1) In the bandwidth term, $\frac{p-1}{p}$ grows asymptotically from $\frac{1}{2}$ in the 2-server case, toward 1 in the infinite-servers case. (2) In the latency term, $lg(p)$ grows logarithmically. We observe that, at the scale of 16 servers, the implementation is within a factor of 2x of the theoretical best-case. Compared to the "15x slower than line rate" situation in the netmap [15] paper's baseline, we think operating within 2x of line rate is pretty good.

[4] Conveniently, this butterfly allreduce implementation in MPICH is what we were already using for communication in our FireCaffe framework.

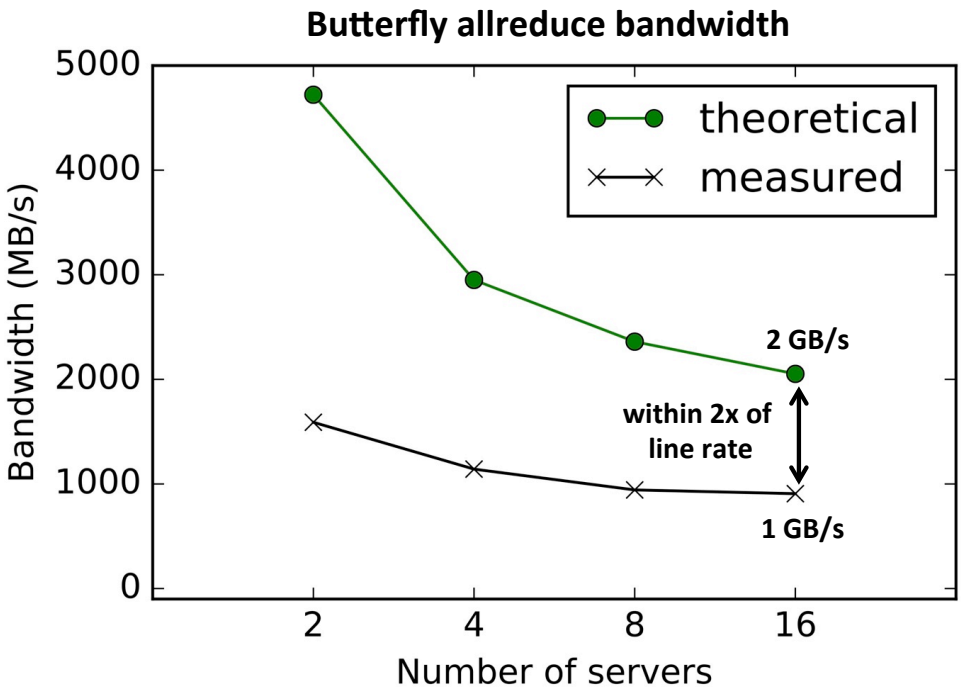

Figure 8. Allreduce bandwidth on the Firebox-0 cluster. Each server contributes a 53MB message. We choose 53MB because the GoogLeNet [16] DNN architecture has 53MB of parameters and therefore has 53MB of model updates provided by each server.

7. Conclusions

DNNs are used in the state-of-the-art approaches to many computer vision and machine learning problems. Training DNNs is computationally expensive and time-consuming. DNN libraries have been tuned to approach the peak FLOPS/s computation rate of an individual GPU, so the next frontier of acceleration is to scale over multiple GPUs. Beyond a certain scale (e.g. 128 GPUs in our FireCaffe [9] experiments), the execution time becomes dominated by server-to-server communication time. We have reviewed a number of approaches (parameter server, binomial tree, ring, butterfly) for implementing DNN communication.

An open question was, how much communication speed is FireCaffe leaving on the table? To begin, we measured peer-to-peer latency and bandwidth, and we plugged these terms into a formula for best-case butterfly allreduce execution time. In experimental results, the butterfly allreduce communication implementation that we use in FireCaffe runs at within 2x of the theoretical best-case. Compared to "15x slower than line rate" in the FreeBSD baseline of the Netmap paper, we think being within 2x of peak is pretty good. This is good news – we have been doing things the "right way" in FireCaffe all along.

7.1. Future Work

Now that we know we're within a factor of 2x of saturating line rate with the allreduce communication pattern, the following opportunities are worth pursuing:

1. different HW configuration (to communicate faster)
2. modify the algorithm or DNN topology/architecture (to communicate less) – in fact, we've made some progress on this: [10]
3. communicate lower precision operands (to communicate less)

References

[1] https://www.mpich.org. 8

[2] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *International Conference on Learning Representations (ICLR)*, 2016. 2

[3] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: building an efficient and scalable deep learning training system. In *OSDI*, 2014. 2

[4] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep learning with cots hpc systems. In *ICML*, 2013. 2

[5] W. Dally. High-performance hardware for machine learning (tutorial). In *Neural Information Processing Systems (NIPS)*, 2015. 2

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. 2

[7] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *SOSP*, 2009. 1

[8] O. Hartmann, M. Kuhnemann, T. Rauber, and G. Runger. Adaptive selection of communication methods to optimize collective mpi operations. In *Workshop on Compilers for Parallel Computers (CPC)*, 2006. 4, 6

[9] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *arxiv:1511.00175*, 2015. 1, 2, 9

[10] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv:¡too new to have an identifier¿*, 2016. 9

[11] L. Mai, C. Hong, and P. Costa. Optimizing network performance in distributed machine learning. In *HotCloud*, 2015. 3

[12] Mellanox. Sx6036: 36-port non-blocking managed 56gb/s infiniband/vpi sdn switch system. http://www.mellanox.com/related-docs/prod_ib_switch_systems/SX6036.pdf, 2013. 6, 7

[13] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. Sparknet: Training deep networks in spark. *arXiv:1511.06051*, 2015. 2

[14] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing (JPDC)*, 2009. 4, 5, 6

[15] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, 2012. 1, 8

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv:1409.4842*, 2014. 1, 2, 9

[17] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications (IJHPCA)*, 2005. 5, 6, 8

[18] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *arXiv:1501.02876*, 2015. 2