FlexReduce: Flexible All-reduce for Distributed Deep Learning on Asymmetric Network Topology

Jinho Lee
Yonsei University
Seoul, Korea
leejinho@yonsei.ac.kr

Inseok Hwang
IBM
Austin, TX, USA
ihwang@us.ibm.com

Soham Shah
Brain Technologies Inc.
San Mateo, CA, USA
sshah@brain.im

Minsik Cho
IBM
Austin, TX, USA
minsikcho@us.ibm.com

***Abstract*—We propose FlexReduce, an efficient and flexible all-reduce algorithm for distributed deep learning under irregular network hierarchies. With ever-growing deep neural networks, distributed learning over multiple nodes is becoming imperative for expedited training. There are several approaches leveraging the symmetric network structure to optimize the performance over different hierarchy levels of the network. However, the assumption of symmetric network does not always hold, especially in shared cloud environments. By allocating an uneven portion of gradients to each learner (GPU), FlexReduce outperforms conventional algorithms on asymmetric network structures, and still performs even or better on symmetric networks.**

I. Introduction

The blooming of AI today has led to ballooning of deep neural networks, with up to hundreds of millions of parameters per network [1]–[4]. Training such networks with massive data-sets can take several days or weeks. To tackle this, a popular approach is distributed deep learning by exploiting data parallelism with multiple GPUs [5]–[9]. A batch of data is split across many learners and the gradients from them are aggregated for a synchronized weight update. While distributed deep learning can significantly shorten training time, communicating across the network for collective gradient updates severely bottlenecks scalability [10]. The computation time is a function of batch size divided by the number of learners, while the communication time is only a function of the total learners and the number of learnable parameters. Therefore, as the number of learners (usually GPUs) increases, the ratio of communication to total training time skyrockets [10]–[13]. A larger batch size can make such ratio more favorable for distributed training, but may degrade the predictive power of the network [14]–[16]. Hence, the need for faster communication algorithms for distributed deep learning (DDL) is clearly apparent.

A well-known yet naive approach to enable large-scale data-parallel training is to deploy a parameter server that handles synchronization, update, and distribution of the parameters [17]. However, a parameter server is unscalable by design, which necessitates more scalable decentralized approaches primarily based on *all-reduce* schemes which are often implemented by ring algorithms [18], [19]. The gradients computed by learners are aggregated using an *all-reduce* operation – the functional equivalent of a *reduce* followed by a *broadcast*, or a *reduce-scatter* followed by an *all-gather*. The all-reduce has been optimized for decades [20] in the areas of HPC (High Performance Computing). However, most of its existing implementations assume that all learners are connected via uniform links [18]–[21], i.e., all links are of the equal bandwidth. This assumption is not true in typical multi-GPU multi-node DDL environments; the links between GPUs in the same node (e.g., PCI-e) are overwhelmingly faster than the links between GPUs across different nodes (e.g., Ethernet).

Recently, BlueConnect [22] was proposed to consider such hierarchy into account for DDL. BlueConnect decomposes the all-reduce into multiple dimensions to minimize data transfer through bottlenecking links. However, this strategy is beneficial only when the DDL topology satisfies the horizontal symmetry, i.e., the nodes in the same level have the same fan-outs. It means that, in multi-node multi-GPU environments, each node should have the same number of GPUs and the exact same number of nodes connect to the routers at the same level.

We argue that the assumption of horizontal symmetry is very unlikely to hold in shared resource environments, such as public cloud settings in which resources are divided into fine-grained virtual instances allocated in a dynamic and elastic manner. Furthermore, such asymmetric topology would be far more common in recently surging federated learning environments [23], [24]. Accordingly, the chance of horizontal symmetry would be slim in practice.

In this paper, we propose FlexReduce, a bandwidth-optimized all-reduce framework for distributed deep learning for non-symmetric networks. FlexReduce eliminates the horizontal symmetry requirement from the all-reduce optimization, using uneven reduce-scatter/all-gather schemes where each learner takes a different portion of gradient data for reduce-scatter. As a result, we achieve a significant speedup even with asymmetric network topologies and maintain the same class of speedup as the prior art on symmetric networks. Our contributions include:

- The FlexReduce algorithm, introducing a decomposed, uneven reduce-scatter followed by an uneven all-gather – minimizing data transfer through bottleneck links in asymmetric networks.
- Theoretical cost analysis of FlexReduce with an upper bound for the number of reduce calls.
- A performance comparison of FlexReduce with other state-of-the-art distributed all-reduce communicative algorithms.

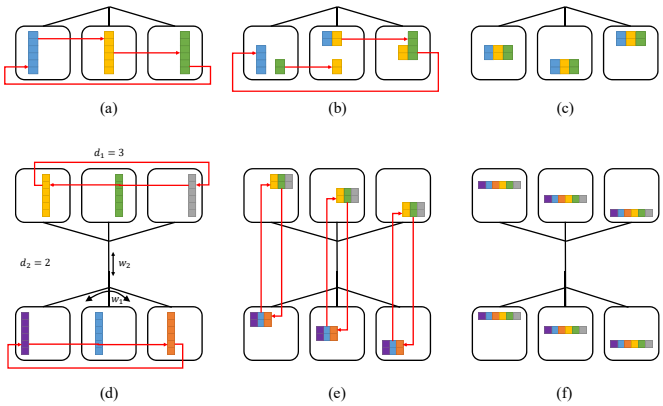

Fig. 1. Reduce-scatter done by (a)-(c) Ring algorithm with 3 learners and (d)-(f) BlueConnect [22] algorithm with 3×2 learners.

II. Preliminary

A. Prior Art

All-reduce in Distributed SGD

All-reduce operations are usually broken into two stages: a *reduce-scatter*, followed by an *all-gather*. One popular way of implementing these is with the ring algorithm. Fig. 1 (a)-(c) shows the reduce-scatter done by ring algorithm with three learners. Each of the d learners splits their data vector into d partitions, followed by sending a partition to its following ring-neighbor while receiving its preceding ring-neighbor's partition. This receive-reduce-send process is repeated over $d-1$ steps (2 in the example). As a result, each learner gets a portion of fully-reduced vectors at the end of the reduce-scatter. Then all-gather is performed simply by reverse-playing the reduce-scatter procedure where the reduces are replaced by overwrite. This algorithm is bandwidth-optimal as long as *all the links in the network have identical bandwidths*. Alternatively, when d is a power of 2, a recursive halving/doubling method can be used, which transfers data in a butterfly fashion and reduces the latency. The ring and recursive algorithms with their variants are often used in standard collective communication libraries such as MPI [20] and Nccl [25].

While the above algorithms are popularly employed in commercial settings, they are bandwidth-optimal only when the link bandwidths are uniform. This assumption usually holds for HPC, where each node is a machine in a cluster. However, in deep learning, the uniform-bandwidth assumption is no longer true. The GPUs used as learners often provide high local bandwidth at the scale of around 100GB/s with PCI-express or NVLink. Connecting network links on the other hand, can range from 0.1GB/s to 10GB/s with connections such as Ethernet or Infiniband. Therefore, the narrow-bandwidth network bottlenecks the entire system.

For this reason, a few all-reduce algorithms in favor of non-uniform hierarchies have been proposed [12], [22]. In [12], all-reduce is done in three phases: the first phase performs local reduce within each machine in correspondence to one master learner on each machine. The second phase performs all-reduce across each machine's master learner. Finally, a broadcast from the masters to their local learners completes the all-reduce process. This helps reduce the data transfer through the bottleneck links. However, this scheme introduces more communicative steps and also leaves the non-master learners idle, creating a significant bandwidth and latency penalty.

BlueConnect [22] provides an interesting solution by decomposing the reduce-scatter and all-gather further into multiple levels using the dimension-regularity of the network as shown in Fig. 1 (d)-(f) with six (3×2) learners where the global bandwidth w_2 is assumed to be much lower than the local bandwidth w_1. First, it performs a local reduce-scatter among d_1 learners in the first dimension. Next, the learners perform d_1 parallel reduce-scatters with the matching d_2 learners in the second dimension to finish the global reduce-scatter. This minimizes traffic through the bottleneck and reduces the number of steps, with all learners participating in every step. This scheme is a generalization of the recursive halving/doubling technique and will work as long as the learner network can be factored into multiple dimensions (3×2 in the example).

FlexReduce can be seen as a novel generalization on BlueConnect, which lifts the symmetry constraint from BlueConnect. FlexReduce provides an equivalent speedup compared to BlueConnect, even on non-factorizable network structures, where BlueConnect falls back to a naive ring scheme.

TABLE I
Notations

n	number of items	d_x	degree of node x
L	number of levels in the tree	δ_k	degree at level k (symmetric)
V_k	set of nodes in level k	w_x	bandwidth of links at node x
p_x^l	ancestor node of x at level l	ω_k	bandwidth at level k (symmetric)

B. Performance Models

As in [20], the time cost for a message of n items in collective communication can be written as the following:

$$\mathcal{T} = \alpha + \frac{n}{w} \tag{1}$$

where α is latency per message and w is the link bandwidth. With this, the time cost for a reduce-scatter on d learners using the ring algorithm can be modeled as :

$$\mathcal{T}_r(n, d, w) = (d-1)(\alpha + \frac{n}{d \cdot w}) \tag{2}$$

because each of the learner sends $\frac{n}{d}$ items over $d-1$ steps. With this, the cost for BlueConnect can be modeled as below:

$$\mathcal{T}_{blc} = 2\sum_{l=0}^{L-1} \mathcal{T}_r(\frac{n}{\prod_{i=0}^{L-1}\delta_i}, \delta_l, \min_{0\le k<L}\{\frac{\omega_j}{\prod_{j=0}^{k-1}\delta_j}\}) \tag{3}$$

where L is the number of levels, or dimensions, and ω_i and δ_i are the link bandwidth and the number of learners in level i, respectively. The sum is representative of one step per dimension. The number of items for each learner decreases through reduce-scatter, but the bandwidth usable per learner also decreases as multiple parallel reduce-scatter operators share higher level links. Finally, it is multiplied by 2 for the all-gather operations performed in reverse.

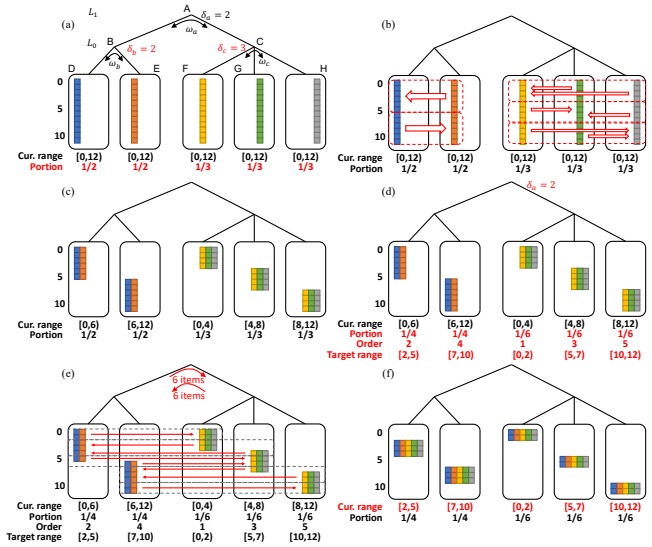

Fig. 2. An example of FlexReduce on 5 learners with 12 items.

III. FlexReduce Algorithm

A. Illustrative Example

The core idea of FlexReduce is uneven decomposed reduce-scatter and all-gather. Rather than performing uniform reduce-scatter by splitting the item vector equally to every learner in the system, we decide to differ the portion of data each learner owns after the reduce-scatter phase. Consequently, we get a balanced, minimized data transfer across the bottleneck link at the root. We use widely implemented reduce() and broadcast() functions [20], [25] as our core building blocks.

The illustrative example in Fig. 2 shows how FlexReduce is performed on a two-machine system, with 2 learners on the left machine and 3 learners on the right. Topologically, this is a depth-2 tree where the upper level's bandwidth (w_a) is significantly less than those of the lower levels (w_b, w_c). FlexReduce processes each level of the tree, from the bottom to the top. First, the target portion for each leaf node (i.e., the learners) is set to all n data items. Then, branch nodes in the first level (L_0) divide the target portions of their leaves by the branch nodes' degrees as in Fig. 2 (a). In the example, since node B has degree of 2, the target portions of leaves D, E will be set to $n/2$. Similarly, the target portions of $F-H$ will be set to $n/3$. Then each leaf is assigned a range sized to its portion, such that its range does not overlap with its neighbors' ranges. Since all nodes still contain the equivalent ranges, we assign the target range evenly without loss of generality. According to the assigned target range, a reduce operation as in Fig. 2 (b) is called so that each leaf will now have reduced data of their peers for all items within their target range (Fig. 2 (c)).

Processing L_1 level starts similarly by dividing the target portions of the leaf nodes ($D-H$) by 2, as node A has degree 2 (Fig. 2). For this, converting the portions to target ranges is non-trivial as all the nodes have different ranges of items and portion sizes. Accordingly, there are three goals in setting new target ranges:

1) Each node has non-overlapping ranges with its peers.
2) The target range of each learner should be continuous.
3) The target range should overlap as much as possible with the current range.

Goal 1 is trivial since we do not need multiple copies of a reduced item. Goal 2 implies minimal reduce calls – too many reduce calls would introduce computational overheads, making it hard to utilize available bandwidth. Goal 3 is to minimize data transfer at the lower level. Even though upper level data transfer is minimized, having a target range that does not completely overlap with the current range generates more lower-level traffic. Ideally, all the target ranges should be completely included in the current range. In an asymmetric topology as in the example, it is nontrivial and often impossible to find a set of target ranges that satisfy both goal 2 and 3.

We devise a heuristic that works well in practice. First, we sort all leaves under the same branch node by their current ranges. In this case, we sort all leaves since the only branch at L_1 is the root. We first sort by the range's end, breaking ties by the range's start. If both values are equal, we use lexicographical order. Then, visiting each leaf in the sorted order, we assign the range $[progress\_counter, progress\_counter + target\_portion)$ and increase the $progress\_counter$. The bottom two rows of Fig. 2 (d) shows the resulting sort order and the assigned ranges. Visiting all nodes guarantees that $progress\_counter = n$ as the target portions are assigned by accumulated degree along the path to the branch node.

After the ranges have been assigned, the reduce calls are made. The data within a single range can come from different leaves, with multiple reduce calls as in Fig. 2 (e). As the transfers are global and the amount of data transferred at the root is equal at both directions, time cost is minimized.

After the uppermost level procedure, each leaf has a non-overlapping, fully reduced range of items(Fig. 2 (f)). Finally, for the uneven all-gather, the aforementioned procedure is performed in reverse, calling broadcasts instead of reduces to complete the all-reduce.

B. Algorithm Details

The pseudo-code of the procedures described in Section III-A is shown in Fig. 3. There are two executable functions:

- **FlexReduce_plan()** is called only when the network structure changes. It computes the execution plan and puts it into $planner$.
- **FlexReduce_execute()** is called every time the system calls all-reduce. It performs the all-reduce operation by executing the entries in the planner.

While the two functions can be merged into one and be called every all-reduce operation, we decouple them since the planning is a serial job and causes substantial overhead if it is calculated every time all-reduce is called.

FlexReduce_plan() takes the network structure as input and outputs the plans to the planner. The planner is a two dimensional array, where the first dimension is the levels in the network tree, and the second dimension is the entries for the calls to make. It works as follows.

```
1  Procedure FlexReduce_plan (root) :
2    planner = ∅
3    for leaf in root.leaves() :
4      leaf.init(portion = 1.0, range = [0, 1.0) )
5    for level in root.bfs_levels().reverse():
6      level_plan = ∅
7      for node in level.nodes():
8        for leaf in node.leaves():
9          leaf.portion /= node.degree
10       assign_range(node, level_plan)
11     planner.add(level_plan)
12     for leaf in root.leaves():
13       leaf.range = leaf.next_range
14
15 Procedure FlexReduce_execute(items):
16   for level_plan in planner: #uneven reduce-scatter
17     for entry in level_plan:
18       reduce(items, entry.participants, entry.subrange)
19     sync()
20   for level_plan in planner.reverse(): #all-gather
21     for entry in level_plan:
22       broadcast(items, entry.participants, entry.subrange)
23     sync()
```

(a)

```
1  # assign range to each leaf, partition range into subrange,
2  # and add to level_plan
3  Procedure assign_range(node, level_plan)
4    progress=0
5    for leaf in sort_by_range(node.leaves()):
6      leaf.next_range =
7            [progress, progress+leaf.portion)
8      for (subrange, participants) in
9            find_bounds(leaf.next_range, node):
10       level_plan.add(subrange, participants)
11     progress+=leaf.portion
12
13 Procedure find_bounds(range, node):
14   subrange.end = range.begin #init
15   while(subrange.end < range.end):
16     subrange.begin = subrange.end
17     participants = ∅
18     for leaf in node.leaves():
19       if subrange.begin ∈ leaf.range :
               participants.add(leaf)
20     subrange.end = min(range.end, leaf.range.
           end for leaf in participants)
21     yield (subrange, participants)
```

(b)

Fig. 3. Pseudo-code of FlexReduce. (a) shows the executable functions and (b) shows the helper functions.

The portion and the range of each leaf under the root are initialized to all items ((a) line 3-4). Then the graph is visited in reverse-bfs level order ((a) line 5-7). Visiting each node, all its leaves get their portion divided by the current node's degree ((a) line 8). Then each leaf is assigned a new range while building the plan for the current level ((a) line 10, *assign_range()*), and the plan is added to the planner ((a) line 11).

On assigning the range, the leaves from the current node are sorted by the end of their range, and then by the start of the range. Starting from a variable $progress$ at 0, the leaves are visited in the sorted order, where each leaf sequentially takes the range equal to its portion and sets it as its $next\_range$ ((b) line 5-7). Each time a leaf is assigned a $next\_range$, all other leaves that currently hold the data are tracked ((b) line 9, $find\_bounds()$). Advancing through the $next\_range$, each time the set of data holder changes, a $subrange$ is formed, associating the data holders as reduce() participants ((b) line 14-21). The subrange and the participants are added into the current level's plan ((b) line 10) to finalize the range assignment.

Following the planning, execution is done through *FlexReduce_execute()*. Iterating through the planner, reduce() calls are made for each entry fully parallel within a level ((a) line 16-18). After waiting for the reduce() calls from current level to complete ((a) line 19), we proceed through the subsequent levels in the planner to finish the uneven reduce-scatter.

Finally, uneven all-gather is performed by calling broadcast() function in the exact opposite order as reduce() calls were made. As a result, the planner is iterated in reverse order ((a) line 20) while performing broadcasts.

IV. Cost Analysis

The notations used in this section are listed in TABLE I. The depth of the tree L is counted from 0 for the bottom-most branch node and $L-1$ in the root, with the leaf nodes uncounted as it will make the equations slightly clearer.

In the first step of local reduce-scatter, the time cost is:

$$\mathcal{T}_0 = \max_{x \in V_0} \mathcal{T}_r(n, d_x, w_x) \tag{4}$$

Since all learners under each branch node in V_k perform reduce-scatter in parallel, the slowest reduce-scatter will dominate the time cost. In the second step, the data transfer goes up one level, and either of the two levels can become the bottleneck.

$$\mathcal{T}_1 = \max(\max_{x \in V_1} \mathcal{T}_r(n, d_x, w_x), \max_{x \in V_0} \mathcal{T}_r(\frac{n}{d_x}, d_{p_x^1}, w_x)) \tag{5}$$

From the perspective of nodes in V_1, the collection of data from all their children covers all the items (i.e., the range is [0,n).) Therefore, it can be seen as doing another local reduce-scatter at level 1. Since the same traffic is split in the lower level, the lower level links transfer n/d_x items each with their bandwidth. Extending up to the root level, the cost for the FlexReduce can be modeled as follows:

$$\mathcal{T}_{FR} = 2\sum_{i=0}^{L-1} \mathcal{T}_i = 2\sum_{l=0}^{L-1} \max_{0 \le k \le l} \max_{x \in V_k} \mathcal{T}_r(\frac{n}{\prod_{i=k}^{l-1} d_{p_x^i}}, d_{p_x^l}, w_x) \tag{6}$$

This easily outperforms the ring algorithm. Also, when the network is symmetric, Eq. 6 meets the conditions below stating that the degrees and bandwidth only depend on the tree level:

$$\forall x \in V_k \quad w_x = \omega_k, \forall x \quad d_{p_x^k} = \delta_k \tag{7}$$

By injecting Eq. 7 into Eq. 6, $\mathcal{T}_{FR}$ becomes equivalent to the cost of BlueConnect [22] $\mathcal{T}_{blc}$ in Eq. 3 as follows:

$$\mathcal{T}_{FR} = 2\sum_{l=0}^{L-1} \max_{0 \le k \le l} \{\mathcal{T}_r(\frac{n}{\prod_{i=k}^{l-1} \delta_i}, \delta_l, \omega_k)\} \tag{8}$$

$$= 2\sum_{l=0}^{L-1} \max_{0 \le k \le l} \{(\delta_l - 1)(\alpha + \frac{n}{\delta_l \omega_k \prod_{i=k}^{l-1} \delta_i})\} \tag{9}$$

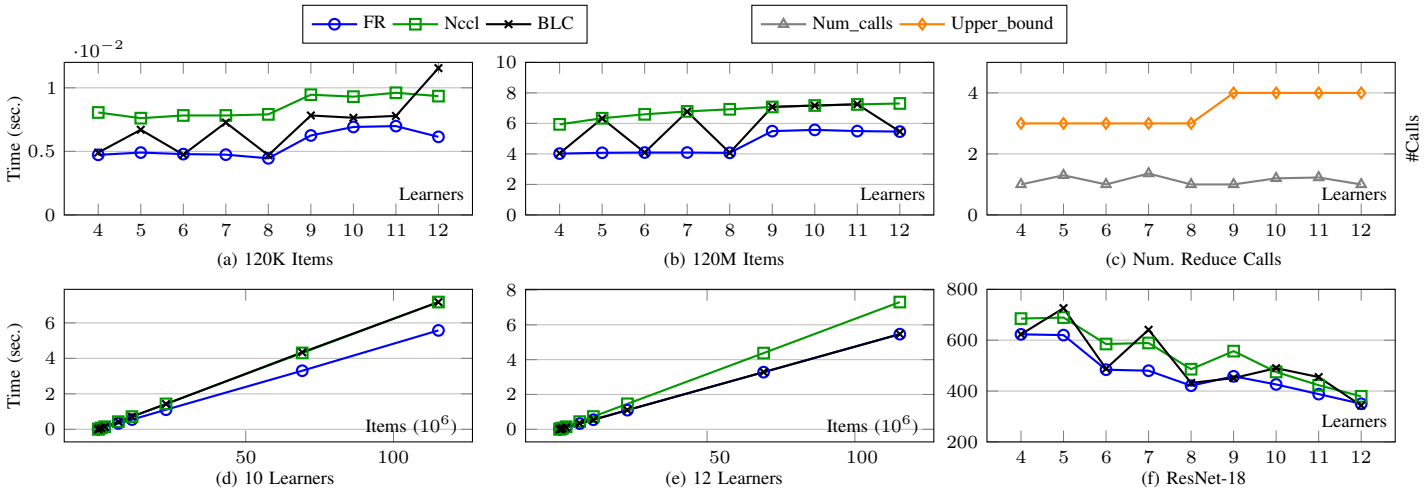

Fig. 4. Experimental results.

$$= 2\sum_{l=0}^{L-1}\{(\delta_l - 1)(\alpha + \frac{n/\prod_{i=0}^{l-1}\delta_i}{\delta_l \min_{0\le k\le l}\{\omega_k/\prod_{j=0}^{k-1}\delta_j\}})\} \quad (10)$$

$$= 2\sum_{l=0}^{L-1}\mathcal{T}_r(\frac{n}{\prod_{i=0}^{l-1}\delta_i}, \delta_l, \min_{0\le k<l}\{\frac{w_j}{\prod_{j=0}^{k-1}\delta_j}\}) = \mathcal{T}_{blc} \quad (11)$$

One penalty not captured in the above model is that when the network is extremely unbalanced, there can be a large number of reduce calls (Alg. 3 (b), line 20). When executing the Lth step of uneven reduce-scatter, the upper bound on the number of reduce calls of a learner v can be expressed as below, assuming that all possible range boundaries do not align with each other.

$$O(N_{calls}(v, L)) = \sum_{m\in V_{L-1}} \left\lceil \frac{\prod_{l=0}^{L}\frac{1}{d_{p_v^l}}}{\min_{u\in leaves(m)}\prod_{l=0}^{L-1}\frac{1}{d_{p_u^l}}} \right\rceil + 1 \quad (12)$$

The numerator inside the sum represents the size of next_range of learner v, and the denominator represents the current range of other leaves. While this upper bound can be high when the network is significantly unbalanced, causing register pressures to GPUs for parallel reduce calls. However in practice, the number of reduce calls stayed within 1-3 in most cases.

V. Evaluation

We prototyped FlexReduce with C++. For primitive collective computations, we used *ncclReduce()* and *ncclBroadcast()* from Nccl [25]. *ncclAllreduce()* and BlueConnect [22] were used as baselines. We have tested on machines each with a i7-6900K CPU at 3.20GHz and four Nvidia GeForce 1080Ti GPUs per machine. The machines are connected via local Ethernet. To represent both horizontally symmetric and asymmetric DDL topology, we used two-machine configurations with 4 to 8 learners and three-machine configurations with 9 to 12 learners with at most one difference in the number of learners per machine. All numbers are averages of 10 runs.

A. All-reduce Latency Comparisons

Fig. 4 (a), (b), (d) and (e) shows the latency of the all-reduce operations. We tested around from 60K to 120M single-precision floating point gradient items. This range covers the from impractically small (LeNet-5 with 60K parameters) CNNs to twice of a very large (ResNet-152 with 60M parameters) CNNs. In the legend, *FR* represents the latency of FlexReduce, *Nccl* represents Nccl *allReduce()*, and *BLC* represents BlueConnect.

Fig. 4 (a) and (b) shows the latency according to different number of learners over 120K and 120M gradient items. As expected, FlexReduce outperforms Nccl in all configurations. With 4 to 8 learner configurations on two machines, the time saving over Nccl is around 32-42%, and for 9 to 12 learners on three machines, the saving is around 21-25%. The performance of FlexReduce also matches the ideal latency, obtained from Eq. 6 by multiplying the relative speedup to the latency of Nccl. We have achieved around 90% to 99% of the theoretical. We suspect the rest is coming from the implementation details, such as the increased number of reduce calls as we discussed in Section IV.

Compared with BlueConnect, FlexReduce also always wins, especially on horizontally asymmetric topologies (5, 7, 9, 10 and 11 learners), because BlueConnect falls back to the ring algorithm and performs similar to Nccl. FlexReduce still performs at least as fast as BlueConnect on symmetric configurations (4, 6, 8 and 12 learners).

Fig. 4 (d) and (e) show the latency by different numbers of items. In all cases, latency is linear to the number of items. On 10 learners, BlueConnect and Nccl almost overlap, as they both use the ring algorithm. On 12 learners, FlexReduce and BlueConnect come close, since the FlexReduce algorithm has same cost as BlueConnect on symmetric topologies as in Eq. 6.

Fig. 4 (c) shows the number of reduce/broadcast calls per leaf node (i.e., learner). In all cases, the upper bound on the number of reduce calls given by Eq. 8 is 3 or 4, as the topologies are not extremely unbalanced. Nonetheless, the average number of calls per learner is much less than the upper bound, not exceeding 1.5 in any cases. This aligns with the result that we were able to get very close to the theoretical speedup.

B. End-to-end Experiments on ResNet

For analyzing the overall effect of FlexReduce on the system, we conducted an end-to-end experiment on Pytorch [26]. We used the same setting as in Section V-A. The network tested is ResNet-18 [3] which has approximately 11M parameters. We modified DistributedDataParallel module of Pytorch by adding our own process group, so that it can utilize the FlexReduce as its communication backend. We have used a reduced-size ImageNet [27] with 64K images and compared the average time taken per epoch of FlexReduce, BlueConnect and Nccl.

Fig. 4 (f) shows the results. Overall, the time per epoch shows the same trends as the latency results in Section V-A; FlexReduce consistently outperforms BlueConnect and Nccl, while BlueConnect fluctuates, i.e., performing similarly to FlexReduce at symmetric configurations and similarly to Nccl at asymmetric conditions.

The average end-to-end relative speedup of FlexReduce over Nccl is shown to be 14.4%. We note that the end-to-end latencies include not only the gradient updates, but also the forward- and backward-pass computations which do not differ in FlexReduce, BlueConnect, and Nccl. Thus it accounts for the end-to-end relative speedup of FlexReduce appearing smaller than those from the all-reduce-only measurements in the previous subsection. There is an additional factor for this smaller relative speedup. Given the datapoints with asymmetric configurations (5, 7, 10, or 11 learners), BlueConnect underperforms Nccl by 6.2% while they are expected to be on par. This aligns with the fact that both BlueConnect and FlexReduce have more number of cuda kernel calls, consuming GPU resources. On the experiments from Section V-A, it did not affect the performance since the GPU resources were not the bottleneck. However in this case, Pytorch overlaps the computation with the communication by starting the all-reduce while some parts of the gradients are still being calculated and that incurs the competition of the GPU resources among the computation and the communication.

Since this penalty should also apply to FlexReduce, we believe we can further improve the performance by optimizing the FlexReduce implementation in a lower level instead of relying on *ncclReduce()* and *ncclBroadCast()*. This becomes one of our future work.

VI. Conclusion

We have proposed FlexReduce, an all-reduce algorithm that provides speedup, especially on asymmetric network topologies. The key idea is using uneven reduce-scatter/all-gather to balance and minimize the traffic through the bottleneck link. The experiments show up to a 42% latency saving, closer to the theoretical model we have derived. This algorithm for all-reduce in asymmetric networks has its value in shared environments such as cloud services or federated training. Consequently, users will have less control over accessible resources, with an increasing likelihood of a system with an irregular structure. This migration will further necessitate the need for optimizing performance in such asymmetrically structured systems. We envision that FlexReduce will provide an efficient communicative backbone for such an era.

References

[1] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *NIPS*, 2014, pp. 487–495.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[4] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017, pp. 4700–4708.

[5] S. S. Girija, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *tensorflow.org*, 2016.

[6] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[8] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012, pp. 1223–1231.

[9] M. Cho, U. Finkler, S. Kumar, D. Kung, V. Saxena, and D. Sreedhar, "PowerAI DDL," *arXiv preprint arXiv:1708.02188*, 2017.

[10] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[11] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[12] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[13] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *ICPP*, 2018, p. 1.

[14] L. Balles, J. Romero, and P. Hennig, "Coupling adaptive batch sizes with learning rates," *arXiv preprint arXiv:1612.05086*, 2016.

[15] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[16] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.

[18] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *SHPCC*, 1994, pp. 357–364.

[19] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *JPDC*, vol. 69, no. 2, pp. 117–124, 2009.

[20] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, Feb. 2005.

[21] R. Rabenseifner, "Optimization of collective reduction operations," in *ICCS*, 2004, pp. 1–9.

[22] M. Cho, U. Finkler, and D. Kung, "Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning," in *SysML*, 2016.

[23] H. B. McMahan and D. Ramage, "Federated Learning: Collaborative Machine Learning without Centralized Training Data," 2017, https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[24] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[25] Nccl, "https://developer.nvidia.com/nccl," 2017.

[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009, pp. 248–255.