2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071117

Logical/Physical Topology-Aware Collective Communication in Deep Learning Training

Jo Sanghoon, Hyojun Son, John Kim
KAIST
Daejeon, Republic of Korea
{scho1118, processor, jjk12}@kaist.ac.kr

***Abstract*—Training is an important aspect of deep learning to enable network models to be deployed. To scale training, multiple GPUs are commonly used with data parallelism to exploit the additional GPU compute and memory capacity. However, one challenge in scalability is the collective communication between GPUs. In this work, we propose to accelerate the AllReduce collective. AllReduce communication is often based on a *logical* topology (e.g., ring or tree algorithms) that is mapped to a *physical* topology or the physical connectivity between the nodes. In this work, we propose a logical/physical topology-aware collective communication that we refer to as C-Cube architecture – Chaining Collective Communication with Computation. C-Cube exploits the opportunity to overlap or chain different phases of collective communication as well as forward computation in a tree algorithm AllReduce. We exploit the communication pattern in a logical tree topology to overlap the different phases of communication. Since ordering is maintained in the tree collective algorithm, we propose *gradient queuing* to enable chaining of communication with forward computation to accelerate overall performance while having no impact on training accuracy. We also exploit the physical topology characteristics to further improve the performance, including proposing detour connections for collective communication while leveraging the additional connectivity to enable a double-tree C-Cube implementation. We implement a C-Cube proof-of-concept on a real system (8-GPU NVIDIA DGX-1) and show C-Cube results in performance improvement in communication performance compared to non-overlapped tree algorithms as well as overall performance.**

I. Introduction

GPUs are widely used for machine learning to train neural networks [7], [26]. As the size of input data increases, the number of GPUs used for training is increased to scale the amount of memory and computation [10]. To enable parallel training across a large number of GPUs, data parallelism approach is widely used where all GPUs have a copy of the same neural network model. Local gradients are calculated within each GPU and are aggregated such that each GPU has an updated set of model parameters. As a result, one critical component to enable efficient scalability of multi-GPU training is the communication of the gradients between the GPUs. Communication between the GPUs is often done through collective communication [27]. There have been different collective communication algorithms/libraries that have been proposed, including Nvidia NCCL [2], RCCL [4], GLOO [1], and Baidu-AllReduce [17]. The collective communication algorithms for AllReduce are often based on a ring collective algorithm [17], [46], [26] or tree collective algorithm [45].

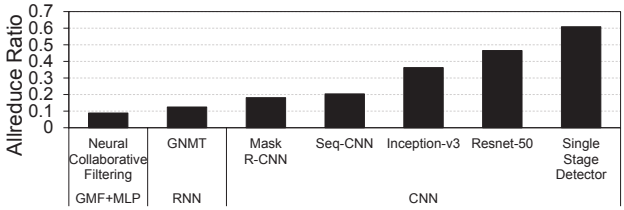


Fig. 1: Ratio of execution time from AllReduce.

To understand the impact of collective communication on overall execution time, we evaluated different workloads from MLPerf [37] with Pytorch [39] and plotted the ratio of AllReduce as a fraction of overall execution time (Figure 1). [1] For some CNN workloads, AllReduce can represent up to 60% of the total workloads (i.e., Single Stage Detector). For other workloads such as Neural Collaborative Filtering that have memory-intensive components with embedding tables, communication itself represents a small fraction. However, collective communication still represents approximately 10% of the total execution time. These results are measured on an 8-node system but as the system size scales, the ratio will increase as more communication is needed if the batch size does not scale. These results are consistent with prior work [15], [18], [57] that have also identified how collective communications can represent a significant fraction of the entire execution time due to the Stochastic Gradient Descent (SGD) approach.

There have been many prior works on accelerating communication in deep learning training [19], including overlapping communication with computation [59], [9], [24]. However, many prior work focus on *asynchronous*, distributed training with parameter servers where communication is only required between workers and parameter server; in comparison, communication in synchronous distributed training requires communication between all workers through collective communication and is used in systems such as Google TPU [28], Nvidia DGX [3], and Facebook Zion [50] systems. In addition, most prior work overlaps communication (or begin to overlap communication) with backward propagation since gradients are generated during backprop; thus, as soon as the gradient

[1]Performance was measured on an 8-GPU Nvidia DGX-1 and NCCL was used for the collective communication.

data are available, communication is initiated. In comparison, we propose an alternative approach where we overlap communication with the forward computation of the *following* iteration.

In this work, we address how deep-learning training using the tree-based AllReduce collective algorithm can be accelerated. In particular, we propose a *logical/physical* topology-aware approach to accelerate the tree algorithm. The *logical* topology in the tree algorithm enables overlap of the different communication phases of AllReduce since the reduction and the broadcast move data in different directions. This overlap reduces the gradient turn-around latency to enable chaining of the communication with the computation. Based on this observation, we propose *C-Cube* architecture – **C**haining **C**ollective **C**ommunication with **C**omputation. C-Cube consists of two components – chaining within the collective communication and chaining across collective communication and computation. Unlike many prior works, the communication overlap is initiated with the *following* iteration of forward propagation, instead of the current iteration of the backward propagation. This approach ensures that the amount of non-overlap communication time can be minimized.

In addition, we exploit the *physical* topology to further accelerate deep learning training. We propose how detour routes can be enabled to support tree algorithm while the additional connectivity or physical channels of the physical topology can be exploited to implement a double-tree C-Cube architecture on a hybrid-mesh cube topology. We demonstrate the proof-of-concept of C-Cube on an 8-node Nvidia DGX-1 system. C-Cube is implemented as CUDA persistent kernels without host-side interrupt by using device-side P2P (Point-to-Point) synchronization. The same synchronization approach used in the overlapped tree collective algorithm is used to implement gradient queuing (and computation overlap) to demonstrate the performance benefits of C-Cube. In particular, the contributions of this work include the following.

- We propose a tree collective algorithm that exploits the *logical* topology where the reduction and broadcast phases are chained or overlapped to improve communication throughput.
- We propose C-Cube that exploits the reduced gradient turn-around latency and in-order computation to overlap communication with the forward propagation computation. We also propose gradient queuing to enable communication and computation overlap.
- We propose a *physical* topology-aware implementation of C-Cube that includes detour routes to implement efficient tree algorithm while exploiting the additional physical connectivity to enable double tree organization.
- We implement a proof-of-concept implementation of C-Cube on an 8-node GPU real system and evaluate C-Cube is able to achieve up to 61% improvement in overall performance, compared to baseline two-tree algorithm.

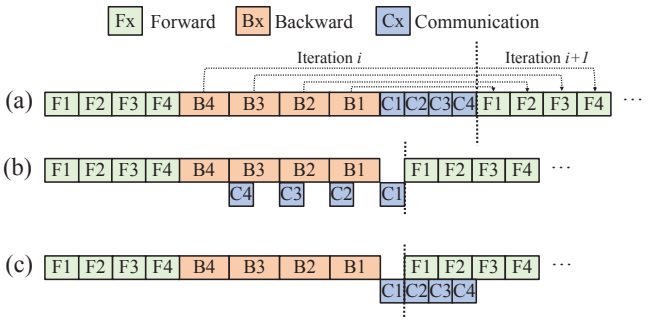


Fig. 2: (a) Deep learning training and the data dependency shown by the arrow, (b) communication overlap with backward computation, and (c) communication overlap proposed in this work. For simplicity, we assume the DNN consists of 4 layers.

II. Background/Motivation

A. Collective Communication/Deep Learning

Deep learning training consists of two components – forward and backward propagation [34]. The forward propagation involves moving input data through the hidden layers to generate the output. In the backward propagation, the errors are sent back from the output to the input, through the hidden layers, and updating the parameters. The gradients are computed using the activations with errors to update the weights of each layer. To scale training, there are two common types of parallelism used in deep learning – model parallelism that partitions the model (or the weights) across multiple GPU nodes and data parallelism [14] that distributes the input data across the different nodes and maintains the same model parameters between nodes. In this work, we focus on data parallelism in distributed deep-learning training and in particular, the challenges in the communication between the nodes.

Stochastic Gradient Descent (SGD) [11] is commonly used in deep learning optimization by calculating the gradients with a subset of total input data called a "mini-batch." There are two approaches to average the gradients across all GPUs – parameter server and AllReduce. Parameter servers average gradients at the "root" node and is often done asynchronously; however, there can be scalability challenges that impact convergence [13]. AllReduce collective communication is commonly used to communicate the gradients across multiple GPU nodes for deep-learning training. AllReduce often consists of two phases – reduction phase (or ReduceScatter) and broadcast phase (or AllGather) [45] and provides a distributed approach where all nodes are involved in averaging the gradients in a synchronous manner. However, as the number of nodes increases, the amount of communication increases and can limit scalability [15], [18], [57].

B. Overlapping of Collective Communication

The dependency between the gradients generated in backward propagation and the next iteration update and forward

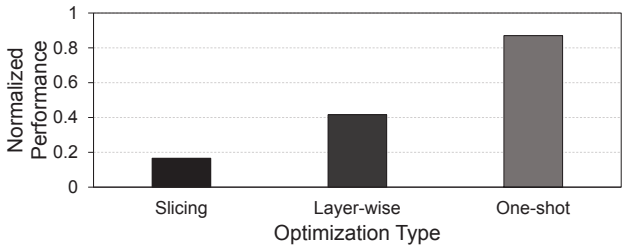


Fig. 3: NCCL AllReduce performance comparison between different optimization types (Resnet-50). Results are normalized to the hardware peak bandwidth of NVLinks

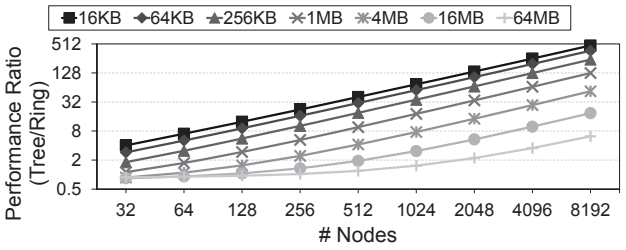


Fig. 4: Performance model comparison between ring and tree AllReduce algorithms.

propagation is shown in Figure 2(a). To reduce the overhead of communication, overlap of the communication with computation have been proposed [59], [20], [39] (Figure 2(b)) to hide communication latency. The overlap of communication begins during the backward computation and continues with the forward computation as well. While these approaches enable communication to begin as soon as possible, the next iteration cannot begin until the last layer of backward (i.e., B1) finished and the gradients are communicated (i.e., C1).

In this work, we take a different approach to overlap communication with computation as shown in Figure 2(c) where the **communication overlap is done with the forward calculation of the *following* iteration**. The start of communication overlap is delayed compared to Figure 2(b); however, because of data dependency, the forward propagation for iteration $i+1$ cannot begin until $C1$ for the first layer is completed. Note that the communication latency for $C1$ cannot be overlapped since it is the last computation of backward but also the first computation in the forward computation. Thus, the only difference is whether the rest of the communication is overlapped with backward propagation of the current iteration or with the forward propagation of the next iteration. Figure 2(c) approach ensures that the amount of non-overlap communication is kept minimal while that might not be the case for Figure 2(b). [2]

More importantly, ***overlapping with forward computation enables the communication to be done in-order and exploit "one-shot" collective communication*** [54], [41], [31] that calls collective communication only once after backward computation ends. Prior works have proposed to hide communication overhead through "layer-wise" (coarse-grain) [59], [9] or "slicing" (fine-grain) [24] approaches on asynchronous parallel training with the parameter server. Layer-wise communicates when gradients are ready from each layer after backward computation whereas slicing divides communication into smaller granularity to better hide the parameter update in the parameter server. While fine-grain communication can improve overall performance in parameter servers where communication is between the parameter server and the workers, slicing data (and increasing the number of times collective communication is invoked) results in higher performance overhead and degrades the performance for collective communication [31]. Figure 3 shows communication performance when "layer-wise" and "slicing" approaches are applied to NCCL AllReduce on DGX-1 with the parameter size of Resnet-50 compared to "one-shot." Layer-wise or slicing requires multiple invocations of Allreduce. Layer-wise results in approximately $2\times$ loss in performance compared to one-shot while slicing results in over $4\times$ reduction in performance. In this work, we use the one-shot approach as our baseline and do not partition the data into any finer granularity but leverage the optimal chunk size in communication. [3] In addition, we also pipeline data communication between the GPUs, similar to NCCL [2] and other prior work.

C. Scale-out Comparison

To understand the benefits of a tree-based AllReduce (compared to a ring-based AllReduce), we use a linear communication cost model [52] ($\alpha + \beta N$) and the notations used for the modeling are the following:

- N – size of the message.
- K – number of chunks.
- P – number of processors.
- α – latency component.
- β – bandwidth component ($= \frac{1}{bandwidth}$)

The AllReduce ring algorithm consists of two stages (Reduce-Scatter and AllGather) and the communication done within each phase are identical. Message size of N is split into P chunks of size $\frac{N}{P}$, each chunk traversing every node, except the node from which it originated, and results in $(P-1)$ steps. The cost or latency for AllGather can be summarized as follows.

$$T_{AllGather} = (P-1)(\alpha + \beta\frac{N}{P}) \quad (1)$$

Ring AllReduce execution time (T_{ring}) is effectively twice the AllGather time, as shown below.

$$T_{ring} = 2(P-1)\alpha + 2((P-1)/P)\beta N \quad (2)$$

The first component is the latency component while the second component depends on the bandwidth. The tree AllReduce

[2]For example, if the earlier communication takes longer, $C1$ communication might not be able to begin immediately after $B1$ finishes. As a result, prior work have proposed different scheduling mechanism to prioritze the critical data [24] [40].

[3]Chunk is the amount of data that is communicated between neighboring nodes in each *step* of AllReduce [2].

algorithm is the combination of Reduction and Broadcast but number of communication steps is $\log(P) + K$ and the performance for each phase in the tree algorithm is the following:

$$T_{reduction} = T_{broadcast} = (\log(P) + K)(\alpha + \beta N/K) \quad (3)$$

where the chunk size is N/K. Based on Equation (3), the optimal number of chunks (K_{opt}) can be calculated to be as follows:

$$K_{opt} = \sqrt{\log(P)\beta N/\alpha} \quad (4)$$

The optimized performance of tree can be calculated by substituting K_{opt} back into Equation (3) and be expressed as follow:

$$\begin{aligned} T_{tree} &= T_{reduction} + T_{broadcast} & (5)\\ &= 2\log(P)\alpha + 2\beta N + 4\sqrt{\alpha\beta N \log(P)} & (6) \end{aligned}$$

The main difference, compared to the ring algorithm, is that the latency component is $O(log(P))$, not $O(P)$, since the depth of the tree is proportional to $log(P)$. To compare the performance of the two algorithms, we use the α, β, P, N parameters from [25] and plot the performance ratio of $1/T_{tree}$ and $1/T_{ring}$ as a function of P and N, in Figure 4. Thus, ratio higher than 1 shows that tree algorithm outperforms the ring algorithm and vice-versa. As shown in Figure 4, for smaller message sizes, the tree algorithm provides better performance since latency (α) is the dominant term. However, for large message sizes, ring algorithm shows slightly better performance (by up to 14%) for smaller number of nodes. For the large message sizes, where the performance is bandwidth-dominated, ring-algorithm has been shown to be bandwidth-optimal [52]. However, as the number of nodes increases, the performance (or the inverse of latency) for communication using the tree algorithm exceeds that of the ring algorithm as the performance impact from the latency component increases – resulting in the tree algorithm being more scalable. In this work, we explore improving the performance of tree AllReduce algorithm.

III. C-Cube Architecture

In this section, we describe our proposed C-Cube architecture that includes the overlapped tree algorithm to improve the performance of AllReduce tree algorithm. We describe how performance can be improved by overlapping the different phases of the AllReduce algorithms and also propose a novel gradient queuing architecture that enables C-Cube to chain communication with computation to improve overall throughput.

A. Conventional Collective Algorithms

Example of AllReduce using the ring algorithm [17], [46], [26], [38], [56] and the tree algorithm [45] is shown in Figure 5, where we assume there are 4 nodes and the data that needs to be reduced are partitioned into 4 "chunks." The reduction phase or Reduce-Scatter for the ring algorithm is shown in Step 1–4 (Figure 5(b)). Each node is responsible for reduction on different data chunks that needs to be accessed from all other nodes – i.e., N1 is responsible for chunk 1, N2 is responsible for chunk 2, etc. Each node sends a data chunk to its neighbor in each step, around the ring during the reduction phase. After Step 4, each node has a different completely reduced data chunk. The remaining steps are AllGather or broadcast phase where each node is responsible for sharing its reduced data with all other nodes. After Step 7, all nodes have the reduced data for all chunks and AllReduce is completed. The physical topology of the system does not necessarily need to be a ring topology as a ring can be embedded to the physical topology to enable a "logical" ring algorithm-based communication.

In comparison, the tree algorithm performs reduction as data moves up the tree and broadcast as data moves down the tree (Figure 5(a)). Each data chunk is first reduced at the leaves in Step 1 and the data is continuously reduced as it moves up the tree. The reduction phase is pipelined across the tree – thus, while data chunk 2 is being reduced between the two leaves in Step 2, data chunk 1 is sent up to root node (N4) for the final reduction. At the end of Step 5, all of the reduced data is located at the root node and the reduction phase is completed. The broadcast phase is next performed where each data chunk is broadcast or sent down the tree, one chunk at a time. Similar to reduction, broadcast is also pipelined – i.e., in Step 7, while data chunk 2 is broadcast from N4 to N2, N2 is a performance broadcast of data chunk 1 to N1 and N3. By the end of Step 10, the reduced data is available across all nodes.

B. Key Observations for C-Cube

We first describe the following key observations that enable the proposed C-Cube architecture in this work.

Observation #1: *Because of pipelining in the tree algorithm, some data chunks complete reduction and wait for other data chunks to be reduced before moving on to the broadcast phase.*

As shown in Step 2 of Figure 5(a), data chunk 1 reduction is completed across all nodes but the data is held in the root note (N4) until the entire reduction phase is completed which occurs at the end of Step 5. Instead of waiting for the reduction phase to complete, C-Cube initiates broadcast for the data chunk 1 while the rest of the data is still being reduced.

Observation #2: *While physical topology channels (i.e., inter-GPU channels) are bidirectional consisting of two unidirectional channels in each direction, the "downlink" of tree connectivity is not utilized in the tree algorithm during the reduction phase.*

In the tree-algorithm, the reduction up the tree is followed by the broadcast down the tree (Figure 6(a)). Thus, the "downlinks" are not utilized during reduction and the "uplinks" are not utilized during broadcast. Since network channels in most topologies are bidirectional, consisting of two unidirectional channels, we exploit this observation to enable overlapping of the communication.

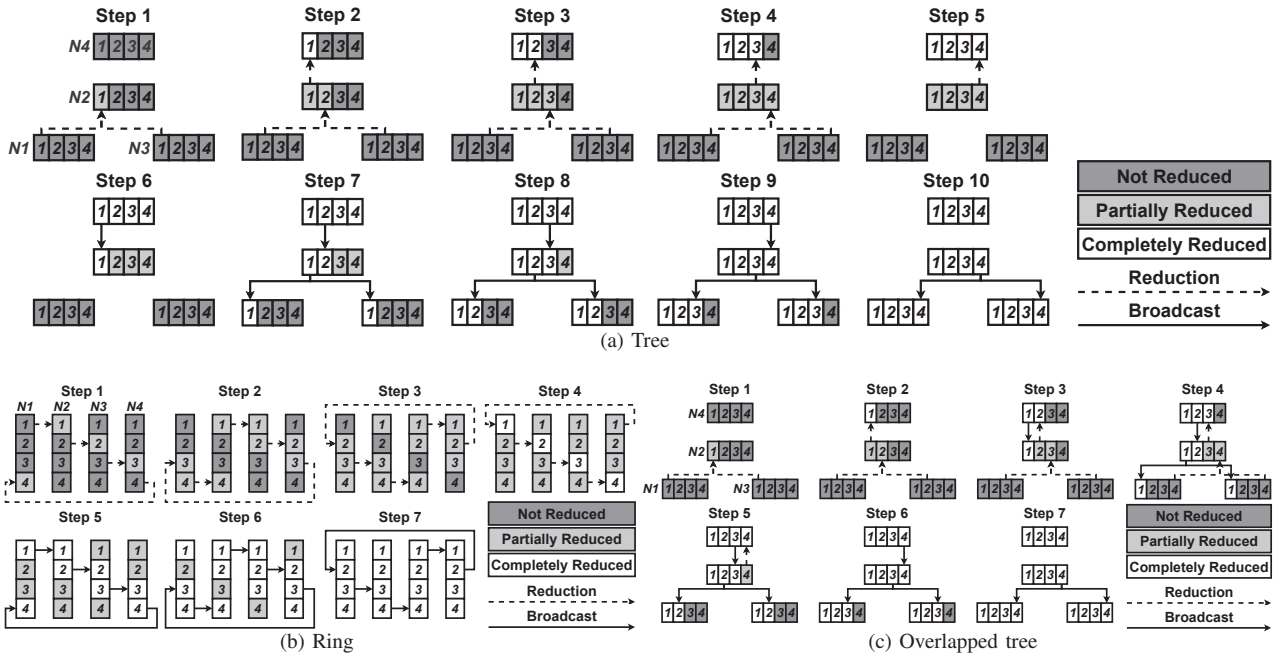


Fig. 5: High-level block diagram illustrating the conventional (a) tree algorithm (Step1-5: Pipelined reduction, Step6-10: Pipelined broadcast), (b) ring algorithm (Step1-4: Reduce-Scatter, Step5-7: AllGather), and (c) our proposed overlapped tree algorithm for AllReduce collective communication.

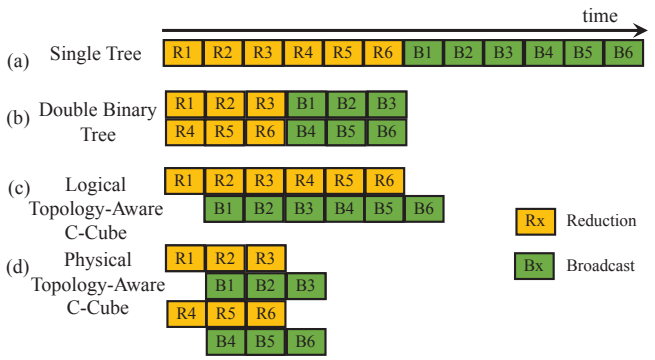


Fig. 6: Comparison of different AllReduce tree-algorithm implementations. For simplicity, different levels of the tree is not shown in the diagram. The diagrams illustrate the communication behavior for a single communication and we assume the communication amount is broken up into 6 "chunks."

To the best of our knowledge, the overlapping of communication within the tree algorithm has not been proposed. To maximize bandwidth, instead of a single tree, a double binary tree [45] is commonly used with two trees to fully utilize the bandwidth (Figure 6(b)) and reduce the communication time. [4] However, when a double tree algorithm is used, overlapping within the communication ***cannot*** be done since some of the physical channels are used in the other tree. For example, the downlink (Node 2-3) on the left tree is also used as the uplink for (Node 2-3) on the right tree (Figure 10(a)). If the communication phases are done separately (i.e., reduction phase completed and then followed by the broadcast phase), this is not a problem but is problematic when trying to overlap within communication. With our proposed logical-topology aware communication overlapping shown in Figure 6(c) with a single tree, the overall communication is not accelerated compared to the double binary tree. However, the first set of gradients needed for the forward calculation in the next iteration is available sooner and thus, can enable computation overlap. As we discuss in Section IV, we also exploit how *physical* topology can enable double-tree organization in our proposed C-Cube architecture (Figure 6(d)).

Observation #3: *Data reduction (and broadcast) are done "in-order" in the tree algorithm.*

In the root node (N4) in Figure 5(b), the reduction for each data chunk occurs ***in-order*** – i.e., data chunk 1 reduction is completed first in Step 4 and it is also the first data to be broadcast. This observation is critical to enable chaining of the communication with the computation. In comparison, for the ring algorithm, when the reduction is completed in Step 4, each node has a different data chunk completely reduced but since the data chunk that each node has differs, the ordering of data is not maintained.

Observation #4: *Physical topology can provide additional connectivity that can be exploited in C-Cube.*

As we discuss in Section IV, the physical topology can provide additional channels or connectivity that can be ex-

[4]For the two tree algorithm, the first tree is flipped to invert the nodes and leaves to create the second tree (Figure 10(a)) and is also used in Nvidia NCCL tree implementation [2].

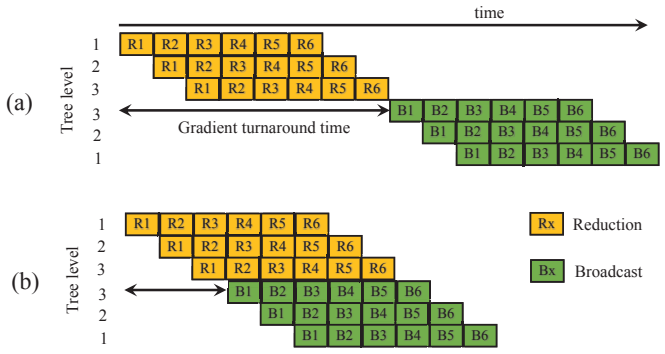


Fig. 7: Timing diagram of (a) baseline tree algorithm and (b) our proposed overlapped tree algorithm that chains reduction and broadcast together. Each row represents a different level in the tree and communication message is partitioned into 6 "chunks" and pipelined.

ploited. We observe how the extra channels can be used to enable *detour* routes to implement a tree algorithm. The extra channels can be leveraged to implement a two-tree C-Cube architecture, and further reduce the communication time as shown in Figure 6(d).

C. Overlapped Collective Tree Architecture

Based on **Observation #1** and **#2**, we propose an overlapped tree algorithm where both the reduction and the broadcast phases are overlapped to exploit the idle network channels. An overview of the proposed architecture is illustrated in Figure 5(c). Compared to the conventional tree algorithm, the initial steps of the overlapped-tree algorithm remain identical until the first data chunk reaches the root of the network. Once the first data chunk is fully reduced, the remaining data chunk reduction continues. However, *in parallel*, the broadcast phase begins for the first chunk since the data has been fully reduced (**Observation #1**). In Step 3, while the reduction continues up the tree, the broadcast for chunk 1 begins *down* the tree using the downlinks (**Observation #2**). In Step 4, the overlap continues as the broadcast for chunk 2 begins. By overlapping reduction with the broadcast, AllReduce is completed in 7 steps, instead of 10 steps for the conventional tree algorithm (Figure 5(a,c)).

A generalized time diagram is shown in Figure 7 that compares the conventional tree algorithm without overlapping to our proposed overlapped tree algorithm. Each box represents the amount of time to communicate a chunk of data. For the conventional tree algorithm without overlapping (Figure 7(a)), the broadcast begins only when the last data chunk reduction completes; however, with the proposed overlapped tree collective, broadcast can begin while reduction continues ((Figure 7(b)).

Leveraging the cost model of the baseline tree algorithm described earlier in Section II-C, the cost model of the proposed overlapped tree algorithm can be summarized as follows:

$$T = 2\log(P)\alpha + \beta N + 3\sqrt{\alpha\beta N \log(P)} \quad (7)$$

Overlapped tree algorithm effectively doubles the height of the tree, compared to the conventional tree algorithm, but requires only a single pass – i.e., the number of steps is $2\log(P)+K$, not $2(\log(P)+K)$.

One important benefit from the overlapping communication is reduction in the *gradient turnaround time* that we define as the time required for the first gradient chunk to finish collective operation and "turns around" – thus, is ready to be used for computation (Figure 7). This is critical when chaining communication and computation described in the following subsection since short gradient turnaround time enables computation to begin earlier and improve overall performance.

D. Chaining Communication with Computation

While the proposed overlapped tree communication accelerates AllReduce, overall performance benefit can be limited since the forward computation still needs to wait until the entire communication phase completes. For example, in Figure 8(a), the top two rows illustrate the communication overlap but computation does not begin until the entire communication is finished – thus, there is an opportunity to further accelerate the workload with chaining (**Observation #3**).

A key component of C-Cube is to chain communication with computation – i.e., instead of waiting for all communication to finish, begin computation as soon as some of the intermediate values are available. To enable such chaining, we propose ***gradient queuing*** – buffering or temporarily storing fully reduced gradient chunks and enable next layer computation to begin before current communication completes. High-level block diagram of computation chaining (or computation overlapping) is shown in Figure 8(b). In this example, we assume there are 5 layers in the neural network and each layer has multiple gradient chunks. $Li-j$ in the top two rows represent the j^{th} gradient chunk in the i^{th} layer. While not drawn to scale, the figures show the common trend of the difference in computation and communication across the different layers in deep learning workloads – the earlier layers have a limited amount of communication and more computation while the latter layers have limited amount of computation but more communication. This is shown with more chunks in the communication for latter layers. Note that *C-Cube does not introduce any data partitioning (or overhead) but rely on existing chunks commonly used in AllReduce communication.*

In Figure 8(b), we illustrate how computation is overlapped with communication. When the first data chunk is finished, instead of waiting for the entire overlapped communication to finish, we leverage gradient queuing to overlap computation and communication. Layer 1 is assumed to have only 1 data chunk and thus, the reduced gradients can be immediately enqueued and then, dequeued for computation. For Layer 2 that consists of two data chunks, once the second data chunk finishes reduction, the reduced gradients can be enqueued and then, can be dequeued when Layer 1 finishes computation. Gradient queuing allows the reduced gradient to be stored

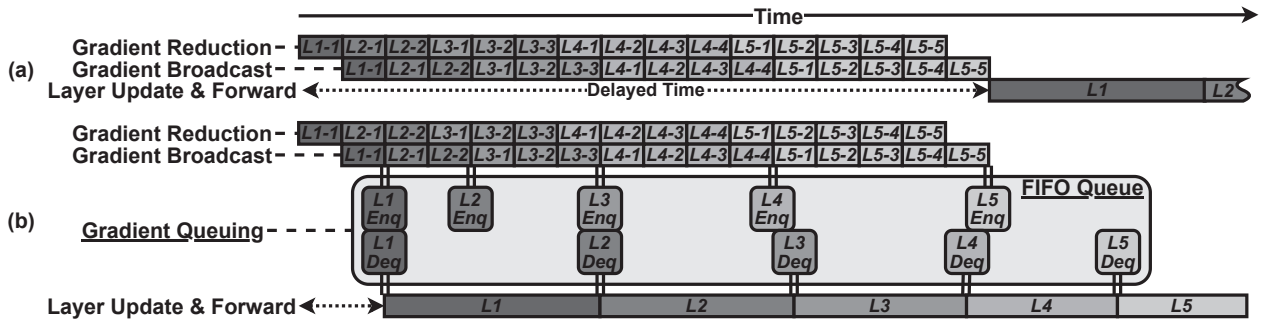


Fig. 8: High-level block diagram of (a) communication overlap and no overlap with computation and (b) gradient queuing that enables overlapping of computation with communication in C-Cube. x-axis represents time and for simplicity, the computation for Layer 3,4,5 is not shown in (a). We assume L1 has 1, L2 has 2, L3 has 3 chunks, etc. – thus, Li-j refers to the j^{th} junk of the i^{th} layer.

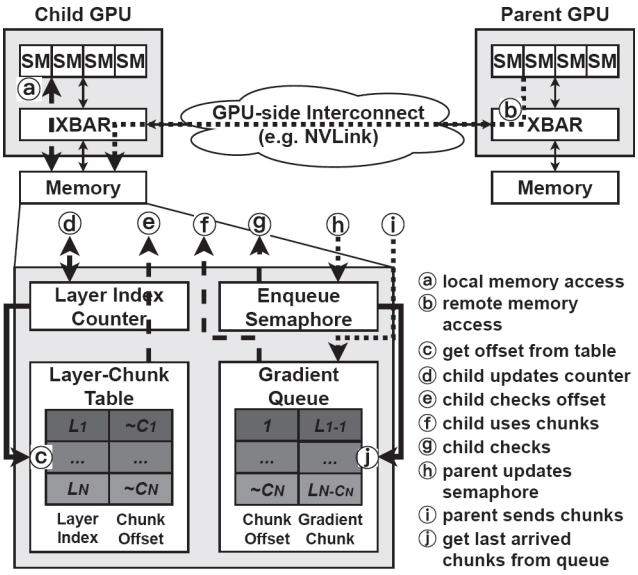


Fig. 9: Detailed architecture of gradient queuing.

or enqueued when reduction is complete. Once all the gradients of a layer are available, the gradients can be dequeued from gradient queuing and proceed with computation while communication for the other layers continue. The same synchronization method of overlapping communication can be used when enqueuing reduced chunks and dequeuing each layer's gradients since both communication and computation are done in the same GPU with different streams. This unified synchronization method between communication and computation minimizes overhead of gradient queuing.

In addition, since gradient queuing is done simply in first-in-first-out (FIFO) order and the same GPU memory is used, the cost of gradient queuing is very minimal. As gradient data completes reduction based on the tree algorithm, the reduced gradient chunks are stored in the same memory address as where they started reduction. As a result, the memory address of gradient data can also be used as the gradient queue due to *in-order property* of the tree collective algorithm. The number of gradient chunks is determined by Equation 7 and gradient queing is also done using the same granularity. Since the gradients are already partitioned in smaller chunks for optimal communication performance and we reuse the memory address of gradient data as gradient queue, gradient queuing has minimal impact on communication performance and GPU main memory usage.

Detailed overview of gradient queuing is shown in Figure 9 where we assume the parent GPU is broadcasting its gradient chunk (ⓑ) to the child GPU (ⓐ) in the broadcast phase of AllReduce communication for C-Cube. The main components of gradient queuing include the following.

- Enqueue Semaphore : points to last gradient chunk that arrived in the gradient queue ⓙ.
- Gradient Queue : stores fully reduced gradient chunks.
- Layer Index Counter : represents the index of the next layer that needs to begin computation.
- Layer-Chunk Table : stores the last gradient chunk offset of each layer.

During gradient queuing, the child GPU utilizes the gradient chunks (ⓐ) that the parent GPU sends (ⓑ). Parent GPU broadcasts its gradient chunks to the gradient queue of the child GPU ⓘ (through the GPU-side interconnect or NVlink) and updates the Enqueue Semaphore ⓗ. The Layer Index Counter (LIC) is initialized to the first layer and points to the next layer that needs to be begin computation for the next iteration. The child GPU checks whether Enqueue Semaphore is equal or larger than the output of the Layer-Chunk Table using the LIC as the input (ⓒⓔ). This determines which layer is finished with both phases of the AllReduce communication and ready to begin computation of the next iteration (ⓖ). When LIC^{th} layer's dequeue is called, Child GPU starts (Layer Index Counter)th layer computation using enqueued gradient chunks ⓕ and updates the LIC value such that in points to the next layer until computation for all layers are completed (ⓓ).

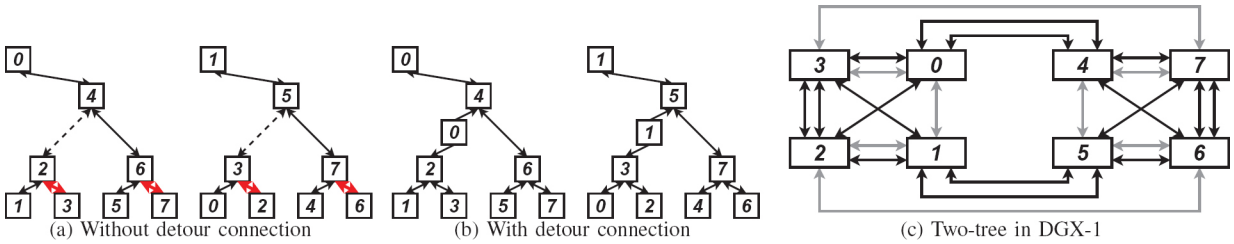


Fig. 10: Block diagram of the (a,b) *logical* tree topology and (b) the *physical* topology of the DGX-1. The dotted line in (a) shows connectivity that does not exist in the physical DGX-1 topology and detour route is leveraged as shown in (b). The black channels in (c) are channels used in our implementation while the grey channels were not used.

```
def lock(lock):
  while atomicCAS(lock,0,1)!=0: # compare and swap
  __threadfence()
```

```
def unlock(lock):
  __threadfence()
  atomicExch(lock,0) # read and write
```

```
def post(lock,
         cnt,
         value):
  lock(lock)
  while cnt==value:
    unlock(lock)
    lock(lock)
  ++cnt
  unlock(lock)
```

```
def wait(lock,
         cnt):
  # check empty
  lock(lock)
  while cnt==0:
    unlock(lock)
    lock(lock)
  --cnt
  unlock(lock)
```

```
def check(lock,
          cnt,
          value):
  lock(lock)
  while cnt<value:
    unlock(lock)
    lock(lock)
    # just check
  unlock(lock)
```

Fig. 11: C-Cube synchronization implementation for reduction-broadcast chaining and gradient queuing.

IV. Physical Topology-Aware C-Cube

In this section, we describe how C-Cube exploits the physical topology characteristics – in particular, how detour (or non-minimal) routes and how extra connectivity between the same nodes can be leveraged to enable two-tree implementation.

A. Tree Topology in DGX-1

To demonstrate the benefits of C-Cube, we implement a proof-of-concept of C-Cube on DGX-1 system with 8 GPUs. While tree algorithm is available in NCCL [2], a proper tree topology was not generated using NCCL for our small-scale system because of the limited physical channel connectivity – thus, we implemented our own CUDA-based tree-algorithm implementation, using the two-tree algorithm [45], with *detour* routes to overcome the limited physical channel connectivity. The baseline tree topology is shown in Figure 10(a) and the dotted line connectivity represents two nodes that are not directly connected in the hybrid mesh-cube topology and results in communication through the PCIe (and the CPU) and can cause significant performance degradation. To avoid this performance bottleneck, we propose a *detour* connection or non-minimal communication through an intermediate GPU without routing through the host. For example, communication from GPU2 to GPU4 is made through intermediate GPU (i.e., GPU0) as shown in Figure 10(b). Prior work [36], [35], [41] have also exploited such non-minimal or detour routes in GPUs; however, this is one of the first work to explore the detour routes for AllReduce collective communication acceleration.

By using detour connections, two binary trees can be created and embedded into the hybrid mesh-cube topology, as shown in Figure 10(c). Although there is some performance loss from using detour connections (discussed in Section V-C), it outperforms the baseline alternative where PCIe is used. The detour connections are implemented by using a separate CUDA kernels that perform *static* non-minimal routing. For example, on GPU 0 in Figure 10(b), one CUDA kernel is responsible for statically routing or forwarding data from GPU 2 to GPU 4 for reduction, and another CUDA kernel is used for routing in the other direction (from GPU 4 to GPU 2) for the broadcast phase of AllReduce.

The logical topology of double tree is shown in Figure 10(a); however, the communication overlap (or chaining) described earlier in Section III-C can not work for a double tree since some of the channels are used in both tree implementation. For example, the uplink from GPU3 to GPU2 (left tree in Figure 10(a)) is effectively the downlink from GPU3 to GPU2 in the second binary tree shown (right tree in Figure 10(a)). [5] If the two phases of collective communication are done separately, this is not a problem; however, when the communication is overlapped, this is problematic and double tree is not possible. In order to enable double tree, we exploit the physical topology of the DGX-1 or the additional connectivity – in particular, GPU2 to GPU3 has two separate (bidirectional) channels and we take advantage of this to enable a 2-tree implementation of our proposed C-Cube architecture (Figure 10(b)).

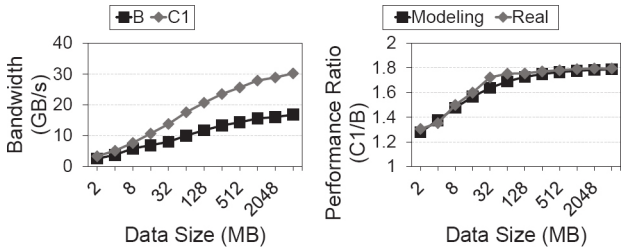
B. Overhead of Host Intervention

Because of high latency between the host and the device (GPU), synchronization using the host can result in a high performance overhead and the benefits of C-Cube will not be realized. To avoid this overhead, we implemented device-side synchronization, as summarized in Figure 11 with `lock` and `unlock` to work without host intervention. [6] Since device-side lock mechanism was not provided as native functions in the GPUs used in our evaluation, we used atomic operation for contention avoidance and thread fence for observability of the updated lock variable to enable lock [33]. Using `lock`

[5]The same is true for channels between GPU6 and GPU7 in Figure 10(a).

[6]The `lock` and `unlock` used in Figure 11 are shown in the pseudocode to demonstrate atomic accesses for the `count` variable.

and `unlock`, we implement semaphores (`post`, `wait` in Figure 11) to manage the receive buffers that are used for communication. In addition to semaphores, we also introduced `check` that does not update the count variable but just checks the state. Gradient queuing uses this functionality when each layer needs to check whether its own gradients are fully reduced and ready to update parameters before being forward computation. Broadcast phase updates count variable with `post` whenever one fully reduced chunk arrives with `enqueue` operations. Each layer then confirms that its chunks have arrived with `check` using `dequeue` and then, each layer can perform update and forward computation, as shown earlier in Figure 8, to enable chaining of computation and communication.



(a) Real implementation (b) Modeling vs. Real

Fig. 12: (a) Performance benefits of communication overlap in tree-based AllReduce on the DGX-1 system and (b) performance benefit analysis comparing measurement and modeling.

V. Evaluation

A. Methodology

To evaluate the benefits of the C-Cube, our evaluation was done on the NVIDIA DGX-1 that consists of 8 V100 GPUs interconnected with NVLink. The high-level block diagram of the system used in our evaluation is shown in Figure 10(c). Each V100 has 6 NVLink with each NVLink providing 25 GB/s of peak bandwidth – for a total peak bandwidth of 150 GB/s. We also complemented our evaluation with simulations for larger scale-out evaluation. We used ASTRA-sim [43] simulator to implement the communication overlap component of C-Cube and compare it with conventional algorithms (i.e., ring, tree) by varying the data size and the number of nodes. The neural network model that we use in our evaluation of C-Cube includes ZFNet [58], VGG-16 [48], and Resnet-50 [21]. ZFNet is a simple CNN architecture while VGG-16 or Resnet-50 is the backbone of some benchmarks in Figure 1 (i.e., VGG-16: Single Stage Detector, ResNet-50: Mark R-CNN). C-Cube was implemented using CUDA GPUDirect with persistent kernels, similar to NCCL, and the neural networks were also implemented with CUDA/cuDNN. [7] We also varied the batch size in our evaluation and to understand the impact of the interconnect bandwidth, we evaluated a "low" bandwidth and "high" bandwidth system. High bandwidth utilizes the full bandwidth of the NVlink for the AllReduce communication while low-bandwidth, which models lower bandwidth interconnect such as PCIe, is modeled by reducing the number of threads allocated to the AllReduce kernel by 4×. While it is known that batch size can impact algorithm accuracy [58], [48], [21], the *proposed C-Cube architecture does not have any impact on the accuracy or the convergence for a given neural network* since C-Cube does not change the ordering of any computation.

B. Results

For real system evaluation, we compare performance results with the baseline (`B`) that uses a two-tree algorithm but with our detour-route implementation (Figure 10(c)). We compare baseline with our proposed overlapped tree algorithm (reduction-broadcast overlap) (`C1`), chaining of communication and computation (`C2`), and our proposed C-Cube architecture (`CC`) which combines `C1` and `C2`. We also compare with results from using the NCCL ring algorithm (`R`). For fair comparison, the hardware interconnect bandwidth provided is identical across the comparison.

1) Communication Performance: The results of overlapping communication are shown in Figure 12(a) as the data size is increased. The overlapping tree algorithm (`C1`) always exceeds the performance of the baseline tree algorithm (`B`) – by 75% for 64MB data size and up to 80% for larger data size. Both `B` and `C1` implement a double tree but the benefits of overlapping are clearly expected since we overlap communication and enable better pipelining of the communication stages to achieve higher performance (or bandwidth). In Figure 12(b), we compare our measurement with the modeling that we described earlier in Section II-C. The expected benefit of `C1` over `B` from modeling closely matches the measured benefits from the real 8-node GPU system.

2) Overall Performance: The performance evaluation comparison of C-Cube and alternatives are shown in Figure 13 for different workloads and batch sizes. The evaluation results are normalized to ideal speedup – i.e., baseline throughput from a single GPU multiplied by the number of GPUs. Thus, normalized performance ratio of 1 represents a system where the cost of collective communication on performance is ignored and linear speedup is achieved. The results are shown for "low-bandwidth" to represent a system with limited bandwidth (i.e., PCIe) as well as "high-bandwidth" where the NVlink bandwidth is fully utilized.

`C1` provides 10% performance improvement on average, and up to 20% improvement, compared to `B`. Chaining computation with communication without overlapped tree algorithm (`C2`) results in slightly higher performance improvement compared to `C1`. However, when both `C2` and `C1` are combined, shown as `CC`, there is approximately 32% improvement on average (and up to 61%). In comparison, `R` shows better performance than `C1` (up to 27%) since DGX-1 is a relatively small system and the ring algorithm is bandwidth-optimal. However, except for small batch size for ZFNet, `CC` exceeds `R` in terms of performance, up to 31% since `CC` can effectively hide

[7] Since C-Cube was implemented with CUDA, high-level framework could not be used and thus, the neural networks had to be written in CUDA.

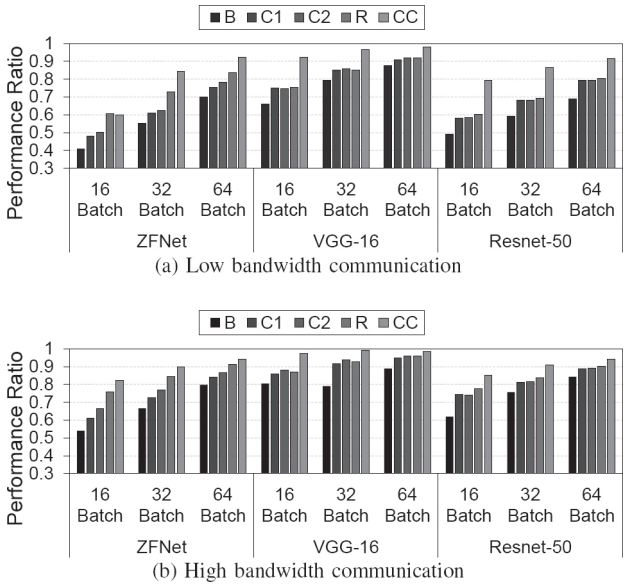


Fig. 13: Normalized performance of C-Cube with different batch size, neural networks, and low/high bandwidth communication. `B`: baseline tree, `C1`: overlapped tree, `C2`: computation chaining, `R`: NCCL ring, `CC`: C-Cube

communication overhead using gradient queuing. [8] Efficiency can be defined as when the performance ratio value approaches 1 in Figure 13. In general, as the batch size increases or with "high" bandwidth communication, the efficiency increases since the communication is effectively lower (with higher interconnect bandwidth) or more compute provides the ability to better hide the communication. The results demonstrate the potential benefits of C-Cube across different configurations and C-Cube can chain computation/communication with up to 98% efficiency and obtain advantages of both better bandwidth utilization and communication hiding together.

3) Scalability Simulations: Results from simulations are shown in Figure 14 to evaluate the scalability of the proposed C-Cube. In ASTRA-sim, we implemented `B` with double-tree [45] AllReduce, as well as `C1`. In the comparison, we assumed constant interconnect bandwidth as `R`. For these evaluations, we evaluate a hierarchical, indirect topology (i.e., intermediate switches) as the number of nodes increases and the results follow trend similar to the modeling shown earlier in Figure 4, in terms of benefit of tree compared to the ring algorithm.

In Figure 14(a), the performance ratio of the proposed overlapped tree (`C1`) and ring (`R`) algorithm is shown. For small data size (i.e., 16kB, 1MB), `C1` provides up to 20× improvement communication performance since latency dominates; however, as data size increases (i.e., 64MB), the benefit

[8]Overlapping can be done with ring algorithm. However, as discussed earlier in Sec II-B, one-shot collective is not possible with a ring algorithm. We evaluated overlap provided by Pytorch [39] with different bucket size and overlap did *not* provide any significant performance improvement in our system, which has also been observed in prior work [31].

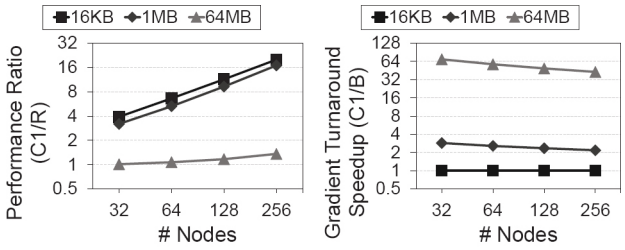


Fig. 14: Scalability analysis using simulations on the potential benefits of communication overlap (`C1`).

of `C1` decreases – achieving up to 35% improvement as the bandwidth dominates for larger data size. However, as the number of nodes increases, performance of `C1` outperformance the ring algorithm and C-Cube (and tree-algorithm) is more scalable implementation of AllReduce. Since the simulator did not have ability to provide detailed modeling of compute in deep learning, overlapping compute with communication and gradient queuing could not be modeled; thus, we measured the *gradient turnaround time* to evaluate the potential benefit of C-Cube as the network size increases in Figure 14(b). The improvement in the gradient turnaround latency of `C1` compared to `B` is plotted. For small data with the limited number of chunks, there is no benefit but as the data or message size increases, the improvement in the gradient turnaround time is significant – 29× improvement on average (and up to 69×). As the number of chunks increases (i.e., 256 chunks for 64MB), the first chunk does not need to wait until the rest of the chunks finish communication and results in significant improvement in performance. As a result, the benefits of C-Cube will likely be higher as both the network size and the message size increases.

C. Discussion

Detour overhead: Overhead from the detour routes are shown in Figure 15 where normalized performance (measured as the inverse of the execution time) of each GPU is shown and thus, higher is better. As discussed earlier, two GPUs (GPU0 and GPU1 in Figure 10(b)) in our implementation are used as detour or intermediate nodes. Since the GPUs are responsible for forwarding the data through GPUDirect, the detour routes end up using GPU resources and can negatively impact performance of the intermediate GPUs. However, our results show that the impact of detour routes is relatively small as the performance degradation of the detour nodes (i.e. GPU0, GPU1) results in only up to 3-4% loss in performance compared to the other GPUs that are not utilized as detour nodes (i.e., GPU2-7). The detour routes does introduce higher latency of communication but the performance is dominated by bandwidth and not latency. As a result, the detour route implementation enables C-Cube to be implemented with minimal impact on overall performance.

Communication-computation pattern: The amount of computation (and communication) differs for each layer and differs across different neural networks. To fully exploit the benefits

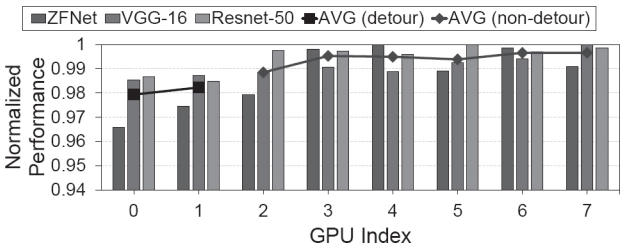


Fig. 15: Normalized performance comparison between between detour nodes (GPU 0, 1) and non-detour nodes (GPU 2-7), using batch size of 64 and high-bandwidth configuration.

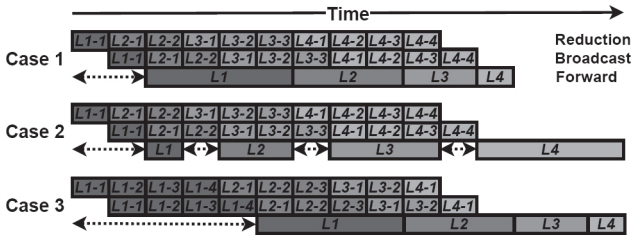


Fig. 16: Communication-computation patterns in deep-learning training. The dotted horizontal arrows illustrates time when the layer update and forward stages cannot proceed.

of C-Cube, the amount of computation should decrease for the latter layers while the communication data size increases for efficient chaining (Figure 16-Case 1). This is necessary to avoid any performance degradation since C-Cube overlaps with forward and not backward propagation – i.e., the longer computation time of the earlier layers can be leveraged for efficient overlapping of the remaining communication. However, for different computation-communication patterns, the benefits of C-Cube can drop. For example, if the amount of computation *increases* for the latter layers as shown for (Case 2), "bubbles" appear where the forward computation for L1 finishes but the subsequent forward computation cannot begin since communication for L2 is not finished. In addition, if the amount of data communicated is higher in the early layers (Case 3), the gradient turnaround time is pushed back.

However, most CNNs follow Case 1 and an example of ResNet-50 is shown in Figure 17. As the layer index increases, the computation time decreases while the parameter size increases. Since CNN increases the number of filters to extract more features from each layers' input feature map when doing forward computation, parameter size increases with each layer. Computation time of each layer also decreases since the feature map of each layer, which is a major bottleneck of CNN [8], becomes smaller to extract the feature of input. As a result, C-Cube takes advantage of a common computation/-communication pattern that is found in CNNs to maximize performance and efficiency.

VI. Related Work

Collective Communication in HPC: There has been a significant amount of work done on collective communications

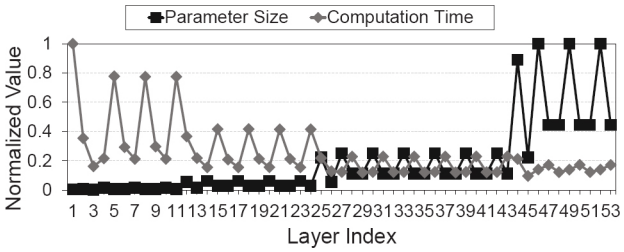


Fig. 17: Comparison of parameter size and computation time for different layers in Resnet-50.

in high-performance computing (HPC). Different collective communication algorithms have been proposed in HPC that are optimized for different message sizes, minimizing latency for short messages and optimize bandwidth for long messages [52]; these algorithms are still applicable to communications in deep-learning workloads. For example, NCCL tree algorithm [25] is implemented by using the two-tree algorithms [45] approach to better utilize bandwidth in common HPC environments and enable logarithmic scalability. Alternative implementations of collective algorithms have also been proposed to exploit new architectures. Faraj et. al [16] showed that collective communications must adapt to the system architecture. Since new architectures are more common but traditional collective communication algorithms are not necessarily optimal, different implementations of collective communication libraries have been proposed [12], [32], [22].

The detour route proposed in this work is similar to non-minimal routing, such as Valiant's routing [53] or global adaptive routing [49], that increases the hop count but enable higher throughput by utilizing all network channels. However, the detour route is not dynamically utilized on a per-packet basis but statically allocated to provide connectivity for collective communication or effectively providing software-based adaptive routing [5], [51]. The physical topology that we exploited in this work was a hybrid mesh-cube topology but it remains to be seen how alternative physical topologies in large-scale systems [29], [30], [6] can be exploited for efficient collective communications.

Communication Optimization in Training: Most of the prior work are often applicable to asynchronous training (or parameter-based servers) while C-Cube addresses training with synchronous servers and collective communication. Recent work (e.g., MGS-SGD [47]) have proposed overlap for AllReduce (or synchronous training). However, the key difference is that overlap is commonly done with the *backward* propagation; in comparison, C-Cube proposes overlap with *forward* propagation. This enables C-Cube to achieve overlap without any partition or re-ordering of the gradient data. ByteScheduler [40] provides a generic framework that is applicable to asynchronous or synchronous training. However, similar to other prior work, it requires re-ordering and partitioning of data; in comparison, C-Cube does not require any partitioning or re-ordering. In addition, ByteScheduler [40] was demonstrated to have significant benefits for parameter

servers but performance improvements with synchronous training was relatively modest. In addition, to the best of our knowledge, C-Cube is the first technique that proposes within communication overlapping. C-Cube also proposes overlap with *forward* propagation (instead of backward propagation) by exploiting the communication pattern of the tree algorithm – thus, enables "one-shot" [54], [41], [31] method to be used that calls collective communication only once after backward computation – thus, minimizing overhead and not requiring any additional data partitioning or re-ordering. Overlapping of AllReduce was proposed for multiple invocations of AllReduce [55]; however, in this work we exploit "one-shot" collective such that AllReduce is only invoked once. Challenges in managing heterogeneous interconnect bandwidth for collective communication have been analyzed [44]. Co-design of the topology and algorithm was proposed through the Multi-tree organization [23]. Improving communication and computation overlap has been proposed through a dedicated hardware logic [42]; however, this work explores a multi-GPU system where communication and computation share the same GPU resources.

VII. Conclusion

In this work, we identified how tree collective algorithms present an opportunity for overlapping reduction and broadcast phases of the collective communication – and thus, improve collective communication throughput. In addition, we proposed communication-computation chaining with *gradient queuing* such that the computation of the workload can also be overlapped to improve overall system performance. To the best of our knowledge, this is one of the first works to exploit the *logical* topology (i.e., tree) of the communication algorithm to enable overlap within the collective communication while exploiting the *physical* topology to implement detour routes for collective communication.

ACKNOWLEDGMENT

We would like to thank Yassawe Kainolda and the anonymous reviewers for their valuable comments. This work was supported in part by IITP grant funded by MSIT (No. 2020-0-01303, 2020-0-01309) and in part by NRF-2020R1A2B5B0100168713.

References

[1] "Facebook. collective communications library with various primitives for multi-machine training (gloo)," 2019. [Online]. Available: https://github.com/facebookincubator/gloo

[2] "Nvidia collective communication library," 2021. [Online]. Available: https://developer.nvidia.com/nccl

[3] "Nvidia dgx systems," 2021. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-systems/

[4] "Rocm communication collectives library (rccl)," 2021. [Online]. Available: https://github.com/ROCmSoftwarePlatform/rccl

[5] D. Abts, G. Kimmell, A. Ling, J. Kim, M. Boyd, A. Bitar, S. Parmar, I. Ahmed, R. DiCecco, D. Han, J. Thompson, M. Bye, J. Hwang, J. Fowers, P. Lillian, A. Murthy, E. Mehtabuddin, C. Tekur, T. Sohmers, K. Kang, S. Maresh, and J. Ross, "A software-defined tensor streaming multiprocessor for large-scale machine learning," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 567–580.

[6] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–11.

[7] 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.

[8] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[9] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 193–205.

[10] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[11] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[12] E. Chan, R. Van De Geijn, W. Gropp, and R. Thakur, "Collective communication on architectures that support simultaneous communication over multiple links," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 2–11.

[13] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," in *International Conference on Learning Representations Workshop Track*, 2016.

[14] 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 *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[15] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[16] A. Faraj and X. Yuan, "Automatic generation and tuning of mpi collective communication routines," in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, pp. 393–402.

[17] A. Gibiansky, "Bringing hpc techniques to deep learning." Dec 2017. [Online]. Available: https://research.baidu.com/bringing-hpc-techniques-deep-learning

[18] P. Goyal, P. Dollár, R. 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.

[19] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. Cadambe, "Local sgd with periodic averaging: Tighter analysis and adaptive synchronization," in *Advances in Neural Information Processing Systems*, 2019, pp. 11 080–11 092.

[20] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 418–430, 2019.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[22] T. Hoefler and D. Moor, "Energy, memory, and runtime tradeoffs for implementing collective communication operations," *Supercomputing frontiers and innovations*, vol. 1, no. 2, pp. 58–75, 2014.

[23] J. Huang, P. Majumder, S. Kim, A. Muzahid, K. H. Yum, and E. J. Kim, "Communication algorithm-architecture co-design for distributed deep learning," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 181–194.

[24] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.

[25] S. Jeaugey, S. Jeaugey, S. Jeaugey, S. Jeaugey, S. Jeaugey, S. Jeaugey, Nvidia, and Nvidia, "Massively scale your deep learning training with nccl 2.4," Mar 2019. [Online]. Available: https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/

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

[27] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer, "How to scale distributed deep learning?" *arXiv preprint arXiv:1611.04581*, 2016.

[28] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, p. 67–78, Jun. 2020.

[29] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2007, pp. 172–182.

[30] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 77–88.

[31] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An in-network architecture for accelerating shared-memory multiprocessor collectives," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 996–1009.

[32] S. Kumar, S. S. Sharkawi, and K. N. Jan, "Optimization and analysis of mpi collective communication on fat-tree networks," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 1031–1040.

[33] W. Landau, "Cuda c: race conditions, atomics, locks, mutex, and warps," 2016.

[34] S. Lee, D. Jha, A. Agrawal, A. Choudhary, and W. K. Liao, "Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication," *Proceedings - 24th IEEE International Conference on High Performance Computing, HiPC 2017*, vol. 2017-Decem, pp. 183–192, 2018.

[35] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel & Distributed Systems*, vol. 31, no. 01, pp. 94–110, jan 2020.

[36] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: evaluating modern gpu interconnect via a multi-gpu benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 191–202.

[37] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "Mlperf training benchmark," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 336–349.

[38] H. Mikami, H. Suganuma, Y. Tanaka, and Y. Kageyama, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[40] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[41] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, 2019.

[42] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, "Enabling compute-communication overlap in distributed deep learning training platforms," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 540–553.

[43] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 22-26, 2020*. IEEE, 2020.

[44] S. Rashidi, W. Won, S. Srinivasan, S. Sridharan, and T. Krishna, "Themis: A network bandwidth-aware collective scheduling policy for distributed training of dl models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 581–596.

[45] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, vol. 35, no. 12, pp. 581–594, 2009.

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

[47] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, and X. Zhao, "Communication-efficient distributed deep learning with merged gradient sparsification on gpus," in *IEEE INFOCOM*, 2020.

[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[49] A. Singh, "Load-balanced routing in interconnection networks," Ph.D. dissertation, Stanford University, 2005.

[50] M. Smelyanskiy, "Zion: Facebook next- generation large memory training platform," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–22.

[51] W. Song and J. Kim, "A case for software-based adaptive routing in numa systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 684–693.

[52] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[53] L. G. Valiant, "A scheme for fast parallel communication," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 350–361, 1982.

[54] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 172–186, 2020.

[55] C. Yang, "Tree-based allreduce communication on mxnet," 2019.

[56] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[57] Y. You, I. Gitman, and B. Ginsburg, "Scaling sgd batch size to 32k for imagenet training," *arXiv preprint arXiv:1708.03888*, vol. 6, 2017.

[58] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[59] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.