Accelerating Allreduce with in-network reduction on Intel PIUMA

Kartik Lakhotia[†], **Fabrizio Petrini**[‡], **Rajgopal Kannan**[§], **Viktor Prasanna**[†]
[†]Department of Electrical and Computer Engineering, University of Southern California,
[‡]Intel Labs, [§]U.S. Army Research Lab
[†]{klakhoti, prasanna}@usc.edu, [‡]fabrizio.petrini@intel.com, [§]rajgopal.kannan.civ@mail.mil

***Abstract*—The PIUMA system maps collective operations directly into the network switches and supports pipelined embeddings for high throughput collective computation. Utilizing these features and PIUMA's network topology, we develop a methodology to generate extremely low latency embeddings for in-network Allreduce. Our analysis shows that the proposed in-network Allreduce is highly scalable, with less than $1\mu s$ single-element latency on $16K$ nodes. Compared to host-based Allreduce, it exhibits $40\times$ less latency and up to $3.5\times$ higher throughput. With deep neural network training as an example, we further demonstrate the benefits of our in-network Allreduce on end-user applications.**

Introduction

Allreduce is a key primitive used in several parallel computing applications, including scientific computing, graph processing, artificial intelligence and others [1]–[3]. It computes a reduction of individual inputs (or input vectors) from participating processors called hosts, and returns the result to every host. Allreduce can become a performance bottleneck, even for computationally intensive applications such as Deep Neural Network training [2].

Several efficient Allreduce algorithms have been proposed for distributed-memory systems [1], [4]. However, host-based algorithms execute multiple rounds of software coordinated communication, resulting in high latency. The number of communication rounds further increase with the system size, deteriorating the performance scalability. In terms of throughput, host-based Allreduces are suboptimal as they communicate partial sums in addition to inputs and outputs. Furthermore, they execute numerous concurrent point-to-point transfers which may result in congestion issues.

This has motivated in-network aggregation methods that use switches to reduce data packets in-flight, to decrease the overall latency and network traffic [5]–[8]. Most current approaches either optimize short vector latency in message passing systems [7], [8], or are designed for small tightly knit clusters [6]. Mellanox's SHARP protocol for in-network computing supports high throughput streaming aggregation [5]. However, collective trees in SHARP are logical constructs that may incur high latency and potential congestion issues, especially if the logical tree does not exactly overlap with the network topology.

In this paper, we present a high-performance in-network Allreduce on the Intel Programmable Integrated Unified Memory Architecture (PIUMA), which features a multi-level HyperX interconnection network [9], [10]. We present novel architectrual features of PIUMA switches that enable pipelined embedding of logical topologies in direct networks, for high throughput computation. We develop a scalable methodology to implement extremely low latency Allreduce on large PIUMA systems with several thousand nodes.

Our analysis shows that the proposed in-network Allreduce significantly outperforms host-based Allreduces, both in terms of the standalone collective performance and the impact on user application. In the context of in-network collectives, we discuss directions for future research.

Background

Allreduce

Allreduce takes a binary associative operator $\bigoplus$ and an input x_i from a collection of host processors P_i ($0 \leq i < n$), and returns the reduction value $y = \bigoplus_{j=0}^{n-1}\{x_j\}$ to all hosts. The single value Allreduce can be generalized to multiple values when the input $\boldsymbol{x_i}$ is a vector of m elements $\{x_{i,0}, x_{i,1} ... x_{i,m-1}\}$, and the output $\boldsymbol{y} = \{y_0, y_1 ... y_{m-1}\}$ is an element-wise reduction of all input vectors, i.e. $y_k = \bigoplus_{j=0}^{n-1}\{x_{j,k}\}$. While traditional Allreduce implementations operate on a single element (typically a norm computation to determine numerical convergence) [3], [7], [8], emerging AI applications perform Allreduce on long vectors [2].

Performance of an Allreduce algorithm can be broadly characterized by two parameters:

1) Number of communication rounds, which affects critical-path computation latency.
2) Communication volume per host, which affects throughput for vector Allreduce.

The Recursive doubling algorithm can compute Allreduce in just $\log_2 n$ rounds. However, in all the rounds, hosts exchange an entire vector with one of their neighbors in a logical hypercube. Consequently, every host incurs $\mathcal{O}(m \log_2 n)$ communication volume.

The Ring and Rabenseifner algorithms are throughput optimized algorithms that communicate approximately $4m$ elements per host [1], [4]. They compute Allreduce in two phases – (1) a reduce-scatter phase that computes disjoint $\frac{m}{n}$-element segments of the result $\boldsymbol{y}$, distributed across the hosts, and (2) a subsequent all-gather phase that collects these segments and delivers the entire vector $\boldsymbol{y}$ to every host. In the Ring algorithm, both phases require $n-1$ communication rounds, in which each host sends (and receives) an $\frac{m}{n}$-element vector segment to (and from) its neighbor in a logical ring. The Rabenseifner algorithm implements the two phases using recursive halving and doubling, respectively, and requires only $2\log_2 n$ rounds .

Allreduce can also be computed on hosts or switches arranged in a logical spanning tree. Inputs are reduced as they move up a reduction tree and the final result $\boldsymbol{y}$ computed at the tree root is distributed to all hosts using a broadcast tree. This approach is commonly adopted by in-network Allreduces [5], [7].

Direct Networks and HyperX topology

In direct or glueless networks, each switch is directly connected to a processor i.e. there are no external switches. A HyperX is a direct network that organizes compute nodes in a D-dimensional logical grid such that the peer nodes in each dimension are all-to-all connected [10]. Formally, if $N_{\boldsymbol{i}}$ denotes the node at grid coordinates $\boldsymbol{i} = \{i_0, i_1, ..., i_{D-1}\}$ in a HyperX, there is a bidirectional link between $N_{\boldsymbol{i}}$ and $N_{\boldsymbol{j}}$ ($\boldsymbol{i} \neq \boldsymbol{j}$) if and only if there is a dimension d such that $i_k = j_k$ for all $k \in \{0, \ldots, D\} \setminus \{d\}$. Such all-to-all patterns in each dimension result in small diameter and high bisection bandwidth. This makes HyperX a scalable interconnection topology for massively parallel systems.

Challenges for In-network Allreduce on Direct Networks

Existing in-network Allreduces typically map logical spanning trees onto physical tree networks [5], [7]. However, when there is a mismatch between the logical tree and the underlying network topology, the logical edges may not map to disjoint paths in the network, potentially leading to congestion. Furthermore, direct networks provide dense connectivity patterns that can be used to reduce the dilation and, in turn, the latency of Allreduce. However, embeddings of pre-defined logical trees may not efficiently utilize the physical network topology to optimize Allreduce latency.

Architecture for in-network reductions

In this section, we describe the topology of the Intel PIUMA network and the architecture of the

PIUMA switches that provide hardware support for embedding logical trees into the network [9]. The proposed architecture is applicable to general multi-level direct networks.

Network Overview

Each PIUMA node is a multi-socket processing unit with multiple cores per socket. The PIUMA system uses a multi-level direct network consisting of a HyperX topology to connect the nodes, and an on-chip mesh connecting compute cores within each node. Specifically, peer nodes in HyperX and sockets within each node are all-to-all connected. In the context of in-network collectives, such dense connectivity patterns provide opportunities for performance optimizations.

A core within a node represents the lowest level in the network and has an on-chip switch exclusively connected to it for network access. Thus, PIUMA has a direct network with one endpoint per on-chip switch. Memory within each node is organized in multiple modules, and each core is directly connected to a memory module. We refer to it as the *local memory* of the core. In this paper we assume that each participating core contributes an input x_i to the Allreduce.

For low-latency data transfer and complete network offloading of collectives, every switch has direct access to the core's local memory via one of the ports. The remaining ports on these switches collectively enable the required inter-socket and inter-node connectivity [11].

Switch Architecture for In-network Computing

The following components in PIUMA switches provide hardware support for in-network reduction:

Collective Engine

As shown in figure 1, each switch is equipped with a Collective Engine (CENG) that can reduce in-flight packets on multiple input ports. CENG supports integer and floating point addition, multiplication, min, max, and several bitwise operators such as AND, NOR, XOR. It is implemented as a pipelined tree of units to compute high radix reductions at line rate with logarithmic latency. Moreover, reductions are computed in a fixed port order when all required inputs are available. This ensures repeatability of results for a given embedding, even for non-associative operators such as floating point addition.

Collective Virtual Channel

Collective packets in PIUMA are routed on dedicated, higher priority vs point-to-point traffic, virtual channels. The arbitration unit in each switch prioritizes packet movement on this channel, providing a performance equivalent of an empty network to in-network collectives, which is *unaffected* by background network traffic.

Configuration Registers

Configuration registers within each switch determine the connectivity between the collective input and output ports, and the operation performed in the CENG. This provides fine-grained control over the paths traversed by packets and computations performed in a collective embedding. Effectively, the problem of embedding Allreduce boils down to *setting the switch configurations in such a way that resulting embedded paths and reduction patterns represent a logical Allreduce topology*.

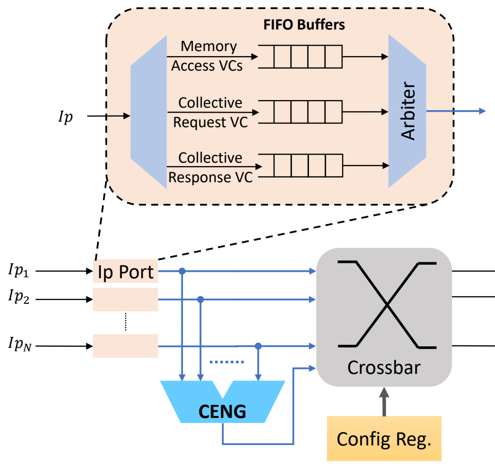


Figure 1: PIUMA switches provide hardware support for in-network collectives.

Multiple In-network Collectives

PIUMA's configuration register mechanism forbids congestion on the collective virtual channels i.e. logical edges in a topology must map to disjoint physical paths. Every embedding in PIUMA is associated with a collective identifier and configuration registers are replicated for each identifier. Therefore, multiple collectives can be mapped on a given identifier (and can be simultaneously operational) provided their embeddings do not share physical links or reduction engines at switches (spatially disjoint).

Additionally, collectives can be mapped across multiple identifiers. Since each identifier uses a unique replica of configuration registers, overlapping embeddings can be simultaneously configured on different identifiers. However, on a given switch, only one of these collectives can be executed at any time (temporally disjoint). Hence, in-network collectives in PIUMA never incur congestion. In this paper, we focus on a global Allreduce that spans the entire system.

Request-response Mechanism

Two categories of packets are supported for collective operations – *request* packets that carry inputs and partial sums, and *response* packets that carry collective output. Separate virtual channels and configuration registers are provided for movement of request and response packets, as shown in figure 1. This mechanism provides flexibility for embedding collectives with varied communication patterns (all-to-all, all-to-one and one-to-all).

Pipelined Reduction

A vector Allreduce may generate numerous packets depending on the vector size. Movement of these packets in the Allreduce embedding must be pipelined for high throughput computation.

For efficient pipelining, PIUMA switches employ a credit signal based mechanism to track the buffer space on the neighboring switches. If the buffer on a neighbor is full, the corresponding output port is labeled unavailable, and the input port(s) forwarding to that output port are *stalled* by the scheduler. This enables pairwise synchronization of adjacent switches (equivalent to consecutive pipeline stages), thereby preventing packet drops without the need of complex reliability or reservation protocols.

In-network Allreduce on PIUMA

Logical Computation

We use a logical *reduction tree* to compute the output $\boldsymbol{y}$, followed by a *broadcast tree* to distribute it. The leaves of the trees represent the local memory endpoints of the participating cores.

Computation is initiated when participating cores call the collective intrinsic, specifying the input vector address and the number of elements m. Thereafter, switches fragment input elements into request packets and insert them into the reduction tree. These packets travel towards the root vertex, getting reduced with other packets at intermediate switches. The root computes the final reduction and packetizes output elements into response packets that are broadcast to the leaves. A counter in each PIUMA switch tracks the number of output elements written to the local memory endpoint. When all m elements are written, it notifies the core that the collective computation is locally completed.

Allreduce Embedding

Reduction Tree

Allreduce latency is a function of the cumulative link latencies in the embedded paths between leaves and the root vertex. Therefore, the key to achieving a low-latency Allreduce in PIUMA is to shorten the longest leaf-to-root path.

To this purpose, we adopt a *dynamic embedding* strategy that only maps the root of the reduction tree to a switch within the minimum eccentricity[1] participant node, and constructs rest of the logcal tree alongside the embedding. For a global Allreduce on the PIUMA HyperX where link latency is proportional to physical distance, root is mapped to the node at midpoint coordinates in each dimension, as shown in figure 2. When embedding Allreduce on a subset of nodes, it is advisable to map the root on a node participating in that Allreduce, to avoid overlaps with potential embeddings on other subsets of nodes.

All input vectors reach the root along the *shortest* network paths between their source and the root, thereby *minimizing the leaf to root latency*. Since Allreduce does not impose a specific order of aggregation, reduction vertices are *dynamically mapped* to the switches where multiple shortest paths intersect. These switches reduce the packets on all incoming paths and forward a single output towards the root.

First, we apply this strategy on the inter-node HyperX. Here, all nodes send their contribution (reduction of their internal inputs) towards

[1]Eccentricity of a node is the maximum distance to any other node in the network.

the root node along dimension ordered shortest paths in the HyperX [10]. Thus, the number of inter-node hops from a memory endpoint to the root node is *upper bounded by HyperX dimensionality*. For the 2D HyperX shown in figure 2, contribution of each node goes to the root via the row-wise peer node in column y_2, where it intersects and is reduced with inputs from other nodes in the same row. The root node reduces inputs coming from all peer nodes in every HyperX dimension, acting as a high-radix switch. Such high-radix reduction is feasible on a PIUMA node as there are several switches within each node.

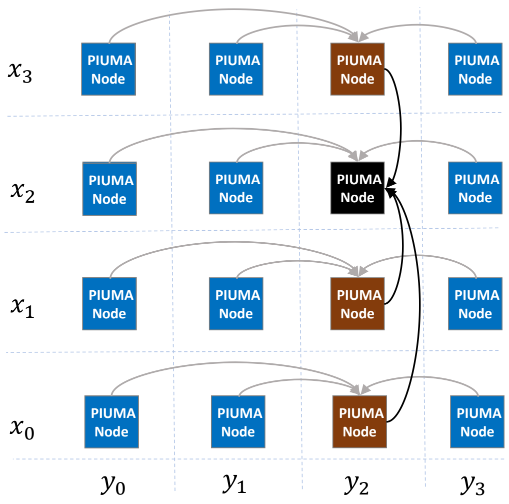


Figure 2: Dimension ordered Allreduce embedding in 2D PIUMA network

Next, we apply this strategy within each node. Here, the root of the intra-node reduction tree is mapped to a switch whose port(s) connect directly to the parent node in the inter-node tree. For example, in figure 3, the root of intra-node reduction tree is mapped to top left switch in socket 0. Inputs from memory endpoints reach this switch via the corresponding shortest paths in the intra-node mesh. They are reduced on the way with other inputs from within the node and external inputs from other nodes (if any). Note that the logical topology finally embedded in a node varies with the node's coordinates in PIUMA HyperX, and cannot be derived by an algorithm which is oblivious to the network topology. As an example, figure 3 shows the resulting topology for one such node.

The dynamic embedding approach has several benefits in addition to latency optimization. It induces numerous path intersections close to the input sources, thereby reducing the network traffic generated by Allreduce. Moreover, it naturally resolves congestion by dynamically mapping reduction vertices. This significantly reduces embedding complexity compared to embedding a pre-defined network oblivious logical tree.

Broadcast Tree

Allreduce output is broadcasted on the same embedding, albeit with packet flow in opposite direction on bidirectional PIUMA links. This is achieved by setting the response configuration registers to an inverse mapping of the request configuration registers. The switches where reduction vertices are mapped in the reduction tree, multicast the output to the corresponding ports in the broadcast tree.

Analysis

Model

In-network Allreduce behaves like a *single round* of communication between any pair of endpoints (cores) P_i and P_j. The input packets are inserted into the network from local memory module of P_i and traverse a given path into the network (undergoing computations on the way), before being written back to the memory module of P_j as the Allreduce output. In a vector Allreduce, packets containing individual elements are pipelined, effectively behaving like a large point-to-point message transfer. Therefore, the cost of an m-element vector Allreduce $T(m)$ can be modeled as per the Hockney's model i.e. $T(m) = \alpha + \beta m$, where α is the maximum latency of the embedded path between any two endpoints and β is the transfer time per element.

Note that there is no additional computation cost because Allreduce reductions are a part of the embedded pipeline. Therefore, computations over any given message element overlap with concurrent transfer of other elements on the network.

System Parameters

We assume a $D-$dimensional (HyperX) system with N_P nodes, N_S sockets per node and N_C cores per socket. For the given PIUMA system, $n = N_P N_S N_C$ cores participate in the Allreduce collective. For simplicity, we assume that the latency of a link in a multi-level direct network

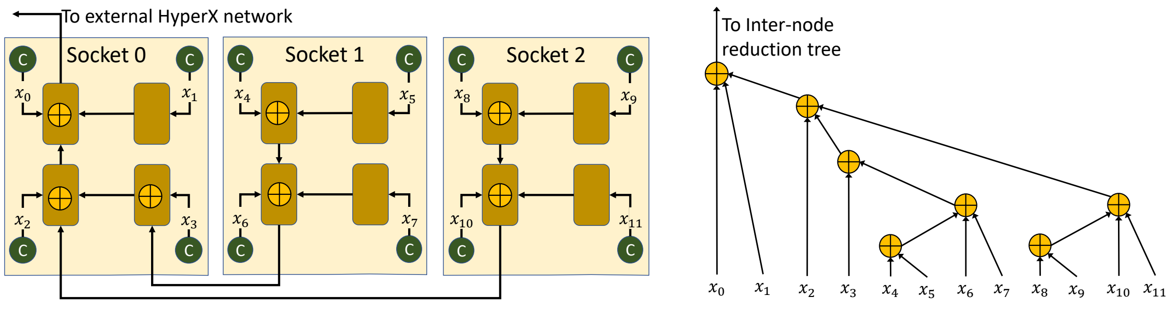

To external HyperX network
Socket 0
Socket 1
Socket 2
C
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
To Inter-node reduction tree
a) Reduction tree embedding within a node
b) Logical topology of embedding

Figure 3: Dynamic reduction tree embedding in a three socket PIUMA node and the resulting logical Allreduce tree.

depends on its level – L_P, L_S and L_C denote the latency of inter-node links in HyperX, inter-socket links within a node and the inter-core links within a socket, respectively. The intra-socket mesh has a diameter of d_{max}. BW_{link} and BW_{mem} denote the bidirectional link bandwidth and memory bandwidth per core, respectively.

Latency
To compute Allreduce latency, we analyze the longest embedded path from a leaf to the root of the reduction tree. In the worst case, this path may incur D inter-node hops. For each inter-node hop, at most one inter-socket hop is required to reach the port that connects to the destination node. An additional inter-socket hop may be needed to arrive at the root switch on the final node, resulting in a total of $D+1$ inter-socket hops. For every inter-node or inter-socket hop, the path may traverse at most d_{max} links within the socket to reach the destination port. Therefore, Allreduce latency which is twice that of the reduction tree, is given by:

$$\alpha_{ar} \leq 2(L_P D + L_S(D+1) + L_C d_{max}(2D+1)) \tag{1}$$

Note that the upper bound on α_{ar} is twice the maximum point-to-point latency in PIUMA, and is independent of the number of nodes in each HyperX dimension.

Throughput
Allreduce throughput is a function of link bandwidth which determines transfer rate over network, and memory bandwidth which determines ingestion rate into the network. Our Allreduce embedding uses duplex communication for simultaneous data transfer on both reduction and broadcast trees and incurs no congestion. Further, m elements are concurrently read from and written to the memory endpoints. Therefore, for s Byte elements, per element transfer time is given by:

$$\beta_{ar} = \max\left(\frac{2s}{BW_{link}}, \frac{2s}{BW_{mem}}\right) \tag{2}$$

Evaluation

Setup

To evaluate the in-network Allreduce, we use an in-house simulator specifically designed to model the switch architecture and network topology of PIUMA.[2] In addition to the architectural parameters given in table 1, the simulator also takes the configuration register values of switches as an input to build a precise rendition of the Allreduce embedding in PIUMA. Since collective channel is prioritized above point-to-point communication, our simulator ignores any background network traffic.

Table 1: PIUMA Network Parameters

Parameter	**Value**
No. of cores per socket (N_C)	8
No. of sockets per node (N_S)	16
Intra-socket link latency (L_C)	5 ns
Inter-socket link latency in a node (L_S)	25 ns
Inter-node link latency (L_P)	50 ns
Memory bandwidth per core (BW_{mem})	51.2 Gbps
Unidirectional link bandwidth (BW_{link})	64 Gbps

For host-based Allreduce, we individually determine the latency in each round as a function of the shortest path between communicating PIUMA

[2]Single node behavior was also functionally verified on Intel's production-level simulator.

cores. Further, given the shared-memory model of PIUMA, we assume that each pair of communicating cores exchange a flag between themselves to indicate validity of data in their local collective buffers. We ignore any potential congestion issues in host-based Allreduce.

Baselines: For throughput comparison of vector Allreduce, we use the throughput optimized Rabenseifner algorithm as the baseline [1]. For latency comparison, we use the Recursive Doubling algorithm which requires only $\log_2 n$ rounds.

Results

Allreduce Performance

Figure 4 shows a comparison of the latency and unidirectional bandwidth achieved by in-network and host-based Allreduces.

Latency: In-network Allreduce achieves more than $40\times$ reduction in latency over host-based Allreduce. This is primarily because in-network Allreduce requires single transfer of data from memory to network and vice-versa. Thus, it avoids synchronization costs and overheads of multiple data transfers between network and memory, as seen in host-based computation. Overall, in-network Allreduce is highly scalable with $< 1\mu s$ latency on a cluster of 16K nodes.

Note that as expected, the latency of in-network Allreduce is almost constant for a given HyperX dimensionality. In contrast, the latency of host-based Allreduce increases logarithmically with the number of nodes due to an increase in the number of communication rounds.

Throughput: The link bandwidth in PIUMA is designed to be higher than local memory bandwidth to support a large number of remote memory accesses [9]. Hence, the optimal Allreduce throughput is limited by ingestion rate from local memory endpoints. Another consequence of this design is that the vector reduction in software contributes significantly to the execution time of host-based Allreduce.

On a cluster of 16 PIUMA nodes, our in-network Allreduce achieves *near optimal bandwidth* on large vectors, with a $3.5\times$ increase in the maximum throughput[3] compared with the host-based implementation. This is because it communicates approximately $2\times$ less data per core compared to host-based Allreduce, and does not incur computation cost for vector reductions.

For a (moderately long) 128KB vector, our in-network Allreduce achieves near optimal bandwidth even on extreme-scale systems. On a single node, it is $3.9\times$ faster than the host-based Allreduce. As the system size increases from 1

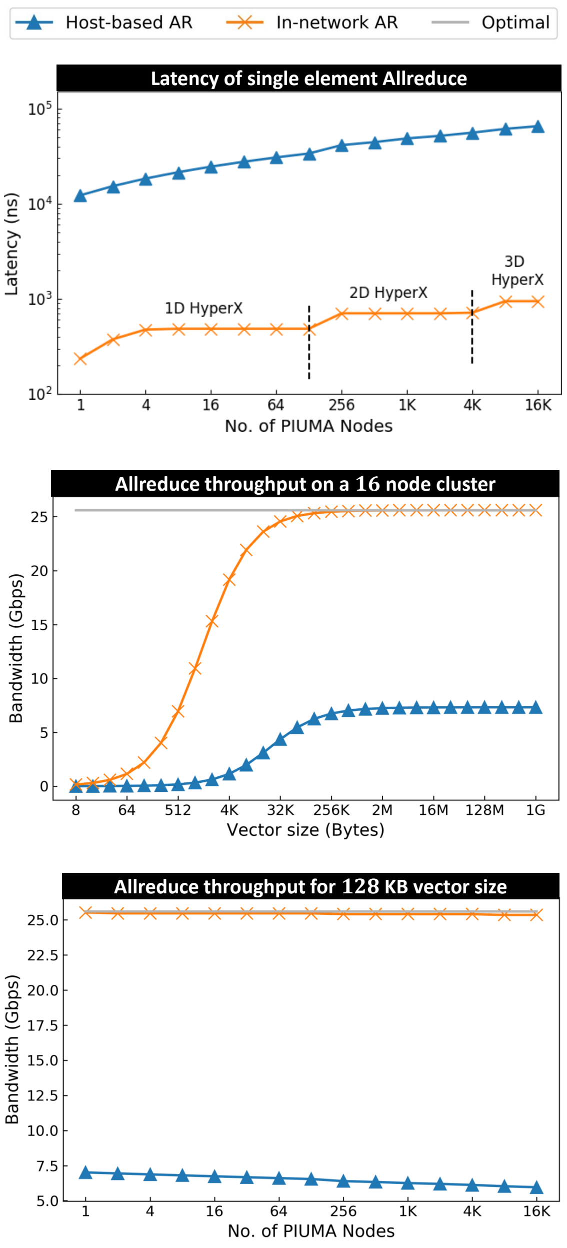


Figure 4: Performance of in-network and host-based Allreduce (AR)

[3]There is a typo in the abstract of our conference paper [11] which mentions a $3.6\times$ speedup.

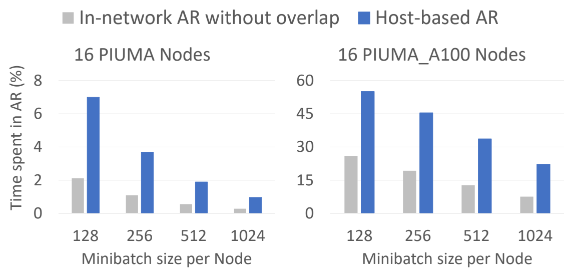
node to 16K nodes, its throughput drops by only 1.4%, unlike the host-based Allreduce whose throughput drops by 26%. Due to congestion issues in the host-based Allreduce, we expect its empirical throughput to be even lower than the analytical projections shown in figure 4.

Deep Learning Application Performance
Data-parallel Stochastic Gradient Descent (SGD) is commonly used for training Deep Neural Networks (DNNs) on distributed systems [2]. Each host stores a replica of the model and computes local gradients using a small mini-batch of training samples. These local gradients are then aggregated using *Allreduce* to compute the model updates in every epoch. On large systems, Allreduce constitutes a significant fraction of the overall DNN training time [5], [6].

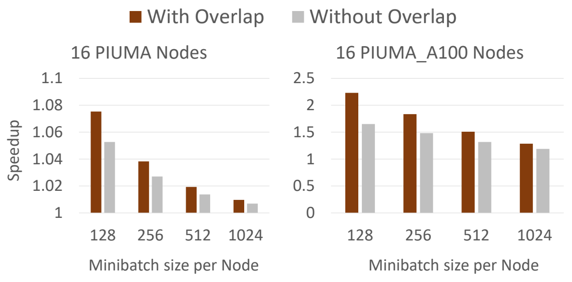
To evaluate the benefits of in-network Allreduce on deep learning, we model the performance of VGG16 training (4 Byte weights) on Imagenet dataset ($224 \times 224 \times 3$ images). Each training epoch for VGG-16 aggregates 550MB of local gradients using Allreduce. Due to the computationally demanding nature of the application, it is important to get a perspective on the device used for training. PIUMA is tailored for sparse data analytics and prioritizes memory and network over compute performance [9]. As a result, local gradient computation dominates training and we observe that a cluster of 16 PIUMA nodes spends only 7% of the training time in Allreduce (figure 5a). For a more meaningful analysis, we evaluate the impact that PIUMA's in-network collectives can have on deep learning accelerators such as GPUs. We envision a hypothetical device with a direct network similar to PIUMA (on-chip and external) but compute capability and memory bandwidth per node equivalent to the Nvidia A100 GPU – 156 TFLOPS for TF32 format and 1.56 TBps, respectively[4]. We assume that the switches on such a device would support in-network reduction of TF32 datatype elements. We refer to this device as PIUMA_A100.

Figure 5a shows the fraction of training time (on a 16 node cluster) spent in computing Allreduce, as a function of minibatch size per node. Minibatch sizes of 256 or 512 images per GPU are commonly used in distributed training [12]. Scaling training to large clusters and large batch sizes is an active area of research [12]. Large minibatch sizes boost per epoch performance, but adversely affect the accuracy and convergence. Conversely, a small minibatch can improve accuracy but increase training time and relative cost of Allreduce per epoch. For example, with a minibatch of 128 images, the cluster of 16 PIUMA_A100 nodes could spend $> 55\%$ of the training time on host-based Allreduce. Using in-network Allreduce decreases this contribution to 25%.

[4]https://www.nvidia.com/en-us/data-center/a100



(a) Percentage of total training time spent in Allreduce (AR)



(b) Training speedup achieved by using in-network Allreduce

Figure 5: Impact of in-network Allreduce on DNN training

Apart from collective acceleration, network offloading also enables *overlap* between local computation and Allreduce collective. To study the impact of such overlap, we model a training algorithm that instantiates individual Allreduces for gradients of each layer, and concurrently executes backpropagation on other layers.

As shown in figure 5b, overlapping in-network Allreduce accelerates DNN training by upto $2.2\times$ compared to host-based Allreduce. Notably, we observed an *almost perfect overlap* between Allreduce and gradient computation. This is because backpropagation first processes dense layers that generate a large number of gradients but require little computation. Allreduce on these gradients completely overlaps with backpropagation on subsequent convolutional layers, which is computationally expensive but generates few gradients.

Discussion

Allreduce is widely used in parallel computing applications. In this paper, we presented an approach to realize high performance in-network Allreduce on multi-level direct network of the Intel PIUMA system. Our analysis showed that the proposed in-network Allreduce can achieve more than an order of magnitude reduction in latency and up to $3.5\times$ higher throughput compared to state-of-the-art host-based algorithms.

Our current approach employs a single leader design where each node communicates the entire reduction vector of its internal inputs on one of the HyperX links. Hence, we utilize a fraction of the available network bandwidth on each node. Allreduce throughput can be further improved by employing multiple embeddings that concurrently compute disjoint segments of the output vector.

An in-network Allreduce can be directly extended for Reduce, Broadcast and Barrier collectives. Additionally, PIUMA's generic hardware support for in-network computing can also be used to offload complex collectives, such as Prefix Scan. This can benefit several applications including sorting, sequence analysis, and load balancing. However, Prefix Scan requires a specific order of input aggregation unlike Allreduce. This makes congestion-avoidance challenging as reductions cannot be mapped on arbitrary path intersections.

Acknowledgement

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-17-3-0004, and the National Science Foundation (NSF) under grant OAC-1911229.

REFERENCES

1. R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*. Springer, 2004, pp. 1–9.
2. A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.
3. K. E. Prikopa, W. N. Gansterer, and E. Wimmer, "Parallel iterative refinement linear least squares solvers based on all-reduce operations," *Parallel Computing*, vol. 57, pp. 167–184, 2016.
4. P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.
5. R. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer *et al.*, "Scalable hierarchical aggregation and reduction protocol (sharp) streaming-aggregation hardware design and evaluation," in *International Conference on High Performance Computing*. Springer, 2020, pp. 41–59.
6. B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An in-network architecture for accelerating shared-memory multiprocessor collectives," in *International Symposium on Computer Architecture*. IEEE, 2020, pp. 996–1009.
7. A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "Mpi collective communications on the blue gene/p supercomputer: Algorithms and optimizations," in *Symposium on High Performance Interconnects*. IEEE, 2009, pp. 63–72.
8. A. Moody, J. Fernandez, F. Petrini, and D. K. Panda, "Scalable NIC-based Reduction on Large-scale Clusters," in *ACM/IEEE conference on Supercomputing*, November 2003.
9. S. Aananthakrishnan, N. Ahmed, V. Cave, M. Cintra, Y. Demir *et al.*, "Piuma: Programmable integrated unified memory architecture," *arXiv preprint arXiv:2010.06277*, 2020.
10. 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*, 2009, pp. 1–11.
11. K. Lakhotia, F. Petrini, R. Kannan, and V. Prasanna, "In-network reductions on multi-dimensional hyperx," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2021, pp. 1–8.
12. Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.