Parallel Primitives for Domain Decomposition in Neural Networks

Russell J. Hewett, Thomas J. Grady II, and Jacob Merizian

Abstract—Training deep neural networks (DNN) in distributed computing environments is increasingly necessary, as DNNs grow in size and complexity. Local memory and processing limitations require robust data and model parallelism for crossing compute node boundaries. We propose a linear-algebraic approach to model parallelism in deep learning, which allows parallel distribution of any tensor in the DNN using traditional domain decomposition strategies. Rather than rely on automatic differentiation tools, which do not universally support distributed memory parallelism models, we show that classical parallel data movement operations are linear operators, and by defining the relevant spaces and inner products, we can manually develop the adjoint, or backward, operators required for gradient-based optimization. We extend these ideas to define a set of data movement primitives on distributed tensors, e.g., broadcast, sum-reduce, and halo exchange, which we use to build distributed neural network layers. We demonstrate the effectiveness of this approach by scaling ResNet and U-net examples over dozens of GPUs and thousands of CPUs, respectively.

1 Introduction

Training deep neural networks (DNNs) in extreme-scale computing environments is a challenging yet increasingly necessary component of modern computational and data science workflows. Large problems in classical data-driven machine learning (e.g., large multi-dimensional or volumetric data processing problems, such as video processing or seismic data processing, as well as natural language processing [1]) and in scientific machine learning (SciML [2]; e.g., those in physics-guided ML [3] which require integration of parallel partial differential equation (PDE) solvers [4], [5]) require robust parallelism models to cross the compute-node boundary to get around limitations on local memory and processing power (even with modern large-memory GPUs). Data parallelism is ubiquitous in deep learning, but model parallelism has been harder to achieve. In particular, this is because the "model" in large deep neural networks is highly irregular and has no uniform spatial structure to induce the sparseness that is typical in large-scale parallel simulation and inverse problems. In this paper, we seek to expose parallelism in neural network training and inference by studying the impact of domain-decomposition strategies on any tensor in a network, including learnable parameters, inputs, and outputs, on algorithms and computational performance.

Recently, multiple frameworks have been developed which partially address the distributed deep learning problem. These frameworks build from, or into, popular frameworks such as PyTorch [6] and Tensorflow [7], which natively support data parallelism, to add support for pipelining [8], or limited support for some model parallelism over some aspects of the network [9], [10], [11], [12]. Native support for distributed learning is also slowly appearing in popular ML frameworks. Parallelism in individual aspects of deep learning, such as convolutional layers has also been investigated [13], as well as aspects of some applications to parallel physics-driven network structures in, e.g., seismic inversion [14]. However, current approaches only provide partial solutions and we lack a complete, integrated framework for treating the distributed learning problem. Here, we aim to provide such a framework. While this manuscript generally addresses "model" parallelism within a single training or inference step, our framework readily admits classical data parallelism and pipelining, as well.

Automatic (or algorithmic) differentiation (AD) is among the most important tools that computational science has contributed to the democratization of deep learning. Given an implementation of a computer algorithm for evaluating a non-linear function $\mathcal{F}$, forward-mode AD produces an implementation of an algorithm for evaluating the action of F, the Jacobian of $\mathcal{F}$, and backward- or adjoint-mode AD produces an algorithm for evaluating the action of F^*, the adjoint of the Jacobian of $\mathcal{F}$ [15]. In computational science problems, such as partial differential equation (PDE) constrained optimization [16], AD is frequently applied to forward computation kernels to develop the correct adjoint kernels necessary for numerical optimization. In deep learning, it is used for similar tasks in the construction of gradient calculations needed to invert for the parameters in composite non-linear functions, such as DNNs.

However, AD tools, especially those in widely used deep learning frameworks, have limited support for the data movement[1] operations required to run codes on distributed memory supercomputers, inhibiting the development of fully parallel deep learning codes. Historically, some AD tools [17], [18] have provided limited support for differentiating distributed memory parallel codes, e.g., enabled via the Message Passing Interface (MPI) [19]. However, such support is not ubiquitous. Fortunately, as we will show, the data movement

- *R. J. Hewett is with the Department of Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA, 24061. E-mail: rhewett@vt.edu*
- *T. J. Grady is with Georgia Tech and was with Virginia Polytechnic Institute and State University.*
- *J. Merizian was with Virginia Polytechnic Institute and State University.*

Manuscript received September 1, 2021.

1. We avoid the term *communication* because our model applies beyond classical distributed memory settings.

operations necessary for distributed memory parallelism are linear operators. Consequently, we do not need to appeal to AD to generate the adjoint operations needed for gradient calculation. Instead, we exploit the definition of the adjoint operator, careful definitions of the spaces they act upon, and the inner products on those spaces, to build a set of primitive operations, and their adjoints, to describe data movement in computers and distributed memory supercomputers.

As we will demonstrate, these operations can be embedded into a deep learning framework using the framework's native interface for specifying new functions, and composited with existing network layers or functions. Thus, the data movement operations, and their adjoints, become merely another function for the framework's automatic differentiation engine to operate on. By posing the data movement operations mathematically, we aim to expose parallelism in neural networks through the mathematics. This will aid optimal implementation in environments where end-users, developers, and researchers do not have full control over both the hardware, compiler, and software stacks and where other, well established HPC software needs to be integrated. We have implemented a proof-of-concept in our distributed deep learning tool, DistDL [20], using MPI (via mpi4py [21]) and PyTorch [6].

The remainder of the manuscript is organized as follows. In Section 2 we define the memory model and the linear algebraic foundation for our framework, including in Section 2.2, where we show that classical distributed data movement primitives fit our framework. In Section 3, we develop data movement primitives for partitioned high-order tensors in deep learning. We use these tools to assemble distributed layer functions in Section 4. In Section 5 we realize some example distributed neural networks and provide empirical demonstrations of the effectiveness of this approach on large CPU and CPU-GPU clusters. Finally, we offer our perspectives on the ramifications of these results and future developments in Section 6.

2 Linearity & Data Movement

Let $\mathbb{F}$ be the space of relevant computer numbers, e.g., integers, reals, or floating point numbers. If $\mathcal{F} : \mathbb{F}^m \to \mathbb{F}^n$ is a linear operator, then $F = \mathcal{F}$ is its Jacobian and the adjoint of the Jacobian, F^*, is defined by the adjoint relationship,

$$\langle F\boldsymbol{x}, \boldsymbol{y}\rangle_{\mathbb{F}^n} = \langle \boldsymbol{x}, F^*\boldsymbol{y}\rangle_{\mathbb{F}^m}\,, \tag{1}$$

where $\mathbb{F}^k$ represents a k-length subset of a computer's memory. For simplicity in this development, we take the inner product to be the standard Euclidean inner product,[2]

$$\langle \boldsymbol{a}, \boldsymbol{b}\rangle_{\mathbb{F}^k} = \sum_{i=0}^{k-1} a_i b_i \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k. \tag{2}$$

Thus, with a concrete implementation of F we can derive and implement concretely the coherent associated F^*, and we can exploit these implementations in a deep learning framework's AD tool. We do not rely on the AD tools themselves to generate the required kernels or to build the computation graph through the data movement operators. While the graph-based approach would produce results identitical to our linear-algebraic approach, we find that our approach is semantically more useful.

In general, the data realized in the subsets of the memory are subsets of tensors. In defining these operations, we make no assumptions about the order of the tensor or the dimension-ordering, size, or storage layout of the underlying multi-dimensional array representing it, though these do matter in a practical implementation. To build parallel primitives for deep learning, we must first understand the nature of $\mathbb{F}^k$ and of the operators on it. In the ensuing discussion, we consider the concept of "a computer's memory" to be broad. While it is easiest to consider $\mathbb{F}^k$ to be the main memory of a single CPU of a single compute node (or worker), this framework admits auxiliary memories, such as those attached to GPU accelerators, remote memory on other compute nodes or cloud instances, or even local or remote disk.

2.1 Primitive Memory Operations

Here, we develop linear representations of primitive memory operations and their adjoints, which we will use to develop parallel data movement primitives and more complex distributed neural network layer structures. We are careful to point out that the manual procedure that we outline below is essentially how adjoint-mode AD works. However, we find it useful to view these operations from a linear-algebraic perspective, rather than from the typical computation-graph perspective used in AD. In general, most AD tools do not handle *all* possible data movement operations (e.g., worker-to-worker, host-to-device, disk-to-host, etc.) within our inclusive memory model, so we must be able to build the operations manually. Thus, our framework provides the theoretical glue necessary to implement these operations when they are not available natively. Moreover, in manual implementations we can make some optimizations that AD-generated codes cannot make, as some operations appear only implicitly in forward codes, but must appear explicitly in adjoint codes, or vice versa. In this manuscript we err on the side of being explicit, while practical implementations may not explicitly include all operations, except perhaps during validation.

2.1.1 Allocation & Deallocation

The *allocation* of a new subset of memory, to be realized by $\boldsymbol{x}_b = \boldsymbol{0}_b$, for a program that already has space for $\boldsymbol{x}_a$ available, is a linear operation $A_b : \mathbb{F}^m \to \mathbb{F}^n$,

$$A_b\boldsymbol{x} = \begin{bmatrix} I_a \\ O_b \end{bmatrix} [\boldsymbol{x}_a] = \begin{bmatrix} \boldsymbol{x}_a \\ \boldsymbol{0}_b \end{bmatrix}, \tag{3}$$

where I_a is an identity operator on the original subset and O_b is a zero operator on the new subset. The adjoint of allocation,

2. When $\mathbb{F}$ is the space of floating point numbers, the inner product must be constructed carefully, especially in parallel environments, because floating point arithmetic is not associative.

A_b^*, is derived through the standard inner product, as follows. Assume $\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^m$ and $\boldsymbol{y} = \begin{bmatrix}\boldsymbol{y}_a \\ \boldsymbol{y}_b\end{bmatrix} \in \mathbb{F}^n$. Then,

$$\begin{aligned}\langle A_b\boldsymbol{x}, \boldsymbol{y}\rangle_{\mathbb{F}^n} &= \sum_{i=0}^{n-1}(A_b\boldsymbol{x})_i y_i = \sum_{i=0}^{m-1}(I_a\boldsymbol{x}_a)_i y_i = \sum_{i=0}^{m-1} x_i y_i \\ &= \sum_{i=0}^{m-1} x_i (I_a \boldsymbol{y}_a)_i = \sum_{i=0}^{m-1} x_i (I_a\boldsymbol{y}_a + O_b^T\boldsymbol{y}_b)_i \\ &= \left\langle \boldsymbol{x}, A_b^T\boldsymbol{y}\right\rangle_{\mathbb{F}^m} = \langle \boldsymbol{x}, A_b^*\boldsymbol{y}\rangle_{\mathbb{F}^m}\,. \end{aligned} \tag{4}$$

Thus, A_b^* is the transpose[3] of A_b, and acts on a realization $\boldsymbol{y}$ from $\mathbb{F}^n$,

$$A_b^*\boldsymbol{y} = A_b^T\boldsymbol{y} = \begin{bmatrix} I_a & O_b\end{bmatrix}\begin{bmatrix}\boldsymbol{y}_a \\ \boldsymbol{y}_b\end{bmatrix} = \begin{bmatrix}\boldsymbol{y}_a\end{bmatrix}. \tag{5}$$

The adjoint of allocation is *deallocation*, and similarly the deallocation primitive D_b has allocation as its adjoint, $D_b^* = A_b$.

We use a liberal definition of "allocation," which goes beyond classical memory allocation operations (e.g., `malloc()` in C) because these operations are describing the semantics of an implementation, not syntax. Allocation is any operation which brings memory into scope, including formal allocation on the heap, acquisition of resources from a memory pool, the addition of data to the stack, creation of a reference, etc. In the context of a neural network layer, this means that if data is not checkpointed for use in the adjoint phase during the forward phase and it goes out of scope, then we consider it to be "deallocated" when the forward function completes.

2.1.2 Clear

The *clear* operator, K_b, sets a realization of a subset of $\boldsymbol{x}$, $\boldsymbol{x}_b$ to $\mathbf{0}$. The operation, $K_b : \mathbb{F}^m \to \mathbb{F}^m$ is realized by,

$$K_b\boldsymbol{x} = \begin{bmatrix} I_a & \\ & O_b\end{bmatrix}\begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_b\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_a \\ \mathbf{0}_b\end{bmatrix}, \tag{6}$$

and it is trivially self-adjoint, $K_b^* = K_b$, following a similar derivation as above.

2.1.3 Add

The *add* operator, $S_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$, performs in-place summation of $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$,

$$S_{a\to b}\boldsymbol{x} = \begin{bmatrix} I_a & \\ I_a & I_b\end{bmatrix}\begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_b\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_a + \boldsymbol{x}_b\end{bmatrix}. \tag{7}$$

The adjoint of an add is also an add, but in the reverse direction,

$$S_{a\to b}^*\boldsymbol{y} = \begin{bmatrix} I_a & I_b \\ & I_b\end{bmatrix}\begin{bmatrix}\boldsymbol{y}_a \\ \boldsymbol{y}_b\end{bmatrix} = \begin{bmatrix}\boldsymbol{y}_a + \boldsymbol{y}_b \\ \boldsymbol{y}_b\end{bmatrix} = S_{b\to a}\boldsymbol{y}. \tag{8}$$

The derivation follows as above.

3. The adjoint is strongly dependent on the inner product and is not always the matrix transpose, but it will be in this work.

2.1.4 Copy

The *copy* operator, which copies data from the subset $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$, has both in-place and out-of-place forms, made distinct only by the semantics of an implementation. An in-place copy is the composition of clear and add while an out-of-place copy is the composition of allocate and add. This may seem pedantic, as "`x = c;`" is more concise than "`x = x * 0; x = x + c;`", but the distinction becomes important when we subsequently define higher-level operations.

The in-place copy operator, $C_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$, takes input $\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_b\end{bmatrix} \in \mathbb{F}^m$, produces output $\hat{\boldsymbol{x}} = \begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^m$, and

$$C_{a\to b} = \begin{bmatrix} I_a & O_b \\ I_a & O_b\end{bmatrix} = \begin{bmatrix} I_a & O_b \\ I_a & I_b\end{bmatrix}\begin{bmatrix} I_a & O_b \\ O_a & O_b\end{bmatrix} = S_{a\to b}K_b. \tag{9}$$

From linear algebra, the adjoint of in-place copy is,

$$C_{a\to b}^* = (S_{a\to b}K_b)^* = K_b^*S_{a\to b}^* = K_bS_{b\to a}. \tag{10}$$

The out-of-place copy operator, $C_{a\to b} : \mathbb{F}^m \to \mathbb{F}^n$, takes input $\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^m$, produces output $\hat{\boldsymbol{x}} = \begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^n$, and

$$C_{a\to b} = \begin{bmatrix} I_a \\ I_a\end{bmatrix} = \begin{bmatrix} I_a & O_b \\ I_a & I_b\end{bmatrix}\begin{bmatrix} I_a \\ O_b\end{bmatrix} = S_{a\to b}A_b, \tag{11}$$

and its adjoint is $C_{a\to b}^* = D_bS_{b\to a}$.

Whether a given copy is in-place or out-of-place depends strongly on the application. Parallel codes for numerical simulation tend to prefer in-place operations, as memory allocation is expensive and best practices for HPC software prefer re-using existing space. In deep learning frameworks, out-of-place operations are common because they simplify and stabilise the computation graph. These distinctions are important when considering a concrete implementation of higher-level primitives, but have minimal theoretical impact.

2.1.5 Move

The *move* operator moves a realization $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$, and similar to copy, has in-place forms,

$$M_{a\to b} = K_aS_{a\to b}K_b, \tag{12}$$
$$M_{a\to b}^* = K_bS_{b\to a}K_a = M_{b\to a}, \tag{13}$$

and out-of-place forms,

$$M_{a\to b} = D_aS_{a\to b}A_b, \tag{14}$$
$$M_{a\to b}^* = D_bS_{b\to a}A_a = M_{b\to a}, \tag{15}$$

which we justify in Appendix A. Again, the choice of in-place or out-of-place forms impacts only some implementation decisions.

2.2 Classical Parallel Primitives

Using these primitive memory operations, we construct linear operators representing several standard parallel data movement primitives and their adjoints. To accommodate operations on distributed memory computers, we now consider the definition of the memory space to include all memories on a compute cluster. While we discuss operations as if the parallel workers are distinct compute nodes, this distinction is made explicit only by a communication library, such as

MPI, and our model is independent of the data movement back-end. In the ensuing discussion, we will generally assume that operations are out-of-place – communicating data results in a new memory allocation on the "receiving" worker. While this is generally not best practice in large-scale simulation, out-of-place operations better map to the AD models of popular deep learning frameworks. The one exception in this presentation is the halo exchange (Sec. 3.2.5), which we will assemble as an in-place operation, following from standard practice in large-scale simulation. Adapting internal mechanics of out-of-place operations to in-place operations has no bearing on the output, so, if such an implementation is preferred in performance environments, it is of minor consequence.

For completeness, we show allocations or deallocations, but in practice they may not be explicitly implemented. While we often express data movement using copy, if the primal realization is deallocated or otherwise not used any further after the copy, the copy may be expressed as a move instead. In practice, many operations we make explicit are needed only theoretically. For example, in the adjoint halo exchange we express clears on the exchange buffers for mathematical consistency, but these are handled implicitly when assigning data to the buffers.

2.2.1 Point-to-point Operations

The most basic distributed memory data movement operation, from which all others can be derived, is *send-receive*. In a concrete implementation, the send-receive pair requires two function calls (send and receive) by separate workers, but from a linear-algebraic perspective, the send-receive operator is simply a copy $C_{a\to b}$, where the subsets $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$ are on the two different workers. Consequently, in the adjoint-phase, we make note of two observations. First, the adjoint of the send-receive involves an add, which means may require an additional communication buffer, but also that if the adjoint realization has $\boldsymbol{y}_a$ initialized to $\mathbf{0}_a$, then the adjoint operation is also essentially a copy. Second, for correctness the adjoint realization $\boldsymbol{y}_b$ should be cleared. This clear does not need to be explicitly performed if $\boldsymbol{y}_b$ is not accessed again. If the forward data $\boldsymbol{x}_a$ is not used after the send-receive, then we can interpret the send-receive as a move operation, instead of a copy, and the first consideration becomes explicit. While the send-receive operation is not self-adjoint, a practical implementation of its adjoint does require a receive-send pair. In a concrete implementation of forward and adjoint distributed copies may use either blocking or non-blocking operations, as long as the implementation of each direction is reentrant.

2.2.2 Scatters and Gathers

The *scatter* primitive is essentially a sequence of send-receive pairs, where subsets of $\boldsymbol{x}$ are copied or moved to multiple other workers. Linear-algebraically, this is is a block-diagonal matrix with send-receive blocks on the appropriate diagonal. The adjoint derivation follows from the previous discussion. If the data movement operations are equivalent to move, then the adjoint operation becomes an instance of the *gather* primitive, which collects data from multiple workers into one subset on one worker, otherwise communication still follows the gather pattern, up to the summation and clear operations. Similarly, the gather can be a forward operation, and its adjoint is essentially a scatter.

2.2.3 Broadcasts

A critical parallel primitive, the *broadcast*, is identified and implemented in many distributed memory deep learning tools [6], [9], [10] because it is necessary to distribute network parameters to multiple workers. A broadcast, $B_{a\to\{k\}}$, is a linear operator from one realization on subset $\boldsymbol{x}_a$ to k copies of that realization on subsets $\boldsymbol{x}_0, \dots \boldsymbol{x}_{k-1}$ and is k copy operations. Assuming an out-of-place formulation, that is, only the input realization $\boldsymbol{x}_a$ is allocated,

$$\begin{aligned} B_{a\to\{k\}}\boldsymbol{x}_a &= D_a C_{a\to k-1}\cdots C_{a\to 1}C_{a\to 0}\boldsymbol{x}_a \\ &= D_a S_{a\to k-1}A_{k-1}\cdots S_{a\to 1}A_1 S_{a\to 0}A_0\boldsymbol{x}_a \\ &= D_a S_{a\to k-1}\cdots S_{a\to 1}S_{a\to 0}A_{\{k\}}\boldsymbol{x}_a \\ &= D_a \Sigma_{a\to\{k\}}A_{\{k\}}\boldsymbol{x}_a \\ &= \begin{bmatrix}\boldsymbol{x}_0 & \boldsymbol{x}_1 & \cdots \boldsymbol{x}_{k-1}\end{bmatrix}^T, \end{aligned} \tag{16}$$

where $\Sigma_{a\to\{k\}}$ is a broadcasting summation that is practically implemented as a standard broadcast and the final deallocation is a notational convenience which may not be explicitly performed. While this above "implementation" scales linearly with k, the canonical logarithmic broadcast implementation has an equivalent representation.

The adjoint of the out-of-place broadcast operator is

$$\begin{aligned} B^*_{a\to\{k\}}\boldsymbol{y}_{\{k\}} &= (D_a C_{a\to k-1}\cdots C_{a\to 1}C_{a\to 0})^*\boldsymbol{y}_{\{k\}} \\ &= C^*_{a\to 0}C^*_{a\to 1}\cdots C^*_{a\to k-1}D^*_a\boldsymbol{y}_{\{k\}} \\ &= D_0 S_{0\to a}D_1 S_{1\to a}\cdots D_{k-1}S_{k-1\to a}A_a\boldsymbol{y}_{\{k\}} \\ &= D_{\{k\}}S_{0\to a}S_{1\to a}\cdots S_{k-1\to a}A_a\boldsymbol{y}_{\{k\}} \\ &= D_{\{k\}}\Sigma_{\{k\}\to a}A_a\boldsymbol{y}_{\{k\}} = \boldsymbol{y}_a, \end{aligned} \tag{17}$$

where $\Sigma_{\{k\}\to a}$ is the adjoint of $\Sigma_{a\to\{k\}}$ and is a reducing summation that is practically implemented as a standard sum-reduction. For in-place formulations, the copy $C_{a\to 0}$ may be replaced with an identity and the allocations (and deallocations in the adjoint) are replaced with clears.

2.2.4 Reductions

The *sum-reduce* primitive, of equal importance with the broadcast, represents summation of k realizations in k subsets of the memory into $\boldsymbol{x}_a$. Again, assuming an out-of-place formulation,

$$\begin{aligned} R_{\{k\}\to a}\boldsymbol{x}_{\{k\}} &= D_{\{k\}}S_{k-1\to a}\cdots S_{1\to a}C_{0\to a}\boldsymbol{x}_{\{k\}} \\ &= D_{\{k\}}S_{k-1\to a}\cdots S_{1\to a}S_{0\to a}A_a\boldsymbol{x}_{\{k\}} \\ &= D_{\{k\}}\Sigma_{\{k\}\to a}A_a\boldsymbol{x}_{\{k\}} = \boldsymbol{x}_a, \end{aligned}$$

where, sensibly, the reducing sum should be implemented via sum-reduction and the final deallocation is again a notational convenience.

Just as the adjoint of the broadcast contains a sum reduction, the adjoint of a sum reduction contains a broadcasting summation,

$$\begin{aligned} R^*_{\{k\}\to a}\boldsymbol{y}_a &= D_a S_{a\to 0}S_{a\to 1}\cdots S_{a\to k-1}A_{\{k\}}\boldsymbol{y}_a \\ &= D_a\Sigma_{a\to\{k\}}A_{\{k\}}\boldsymbol{y}_a = \boldsymbol{y}_{\{k\}}, \end{aligned} \tag{18}$$

and again, an in-place implementation may replace the initial copy with an identity. Other reductions can be expressed similarly, up to linearization of the forward operation.

All-reduce primitives are useful in some frameworks, e.g., in [13] where it used in a distributed convolution, and in normalization layers. Conceptually, an all-reduce is simply the composition of a sum-reduction and a broadcast,

$$\mathcal{A}_{\{k\}\to\{k\}} = B_{a\to\{k\}} R_{\{k\}\to a}, \tag{19}$$

and is trivially self-adjoint,

$$\mathcal{A}^*_{\{k\}\to\{k\}} = R^*_{\{k\}\to a} B^*_{a\to\{k\}} = B_{a\to\{k\}} R_{\{k\}\to a} = \mathcal{A}_{\{k\}\to\{k\}}. \tag{20}$$

A practical implementation does not require explicit reduction and broadcast operations. However, this perspective illuminates a philosophical difference between our model and all-reduce oriented models. In an all-reduce-based distributed framework, the distributed weight updates must be manually all-reduced so that each worker can perform the same update. While correct from an engineering perspective, this means that the adjoint implementation may no longer be coherent with the forward implementation. Following our approach, if the original weight tensor is broadcast during the forward phase, the sum-reduction on the weight updates is induced naturally and the forward and adjoint functions remain coherent.

2.2.5 All-to-all

The *all-to-all* primitive is used to redistribute data across a distributed memory. This primitive, $T_{\{k\}\to\{l\}}$, is often called transpose, due to the movement patterns when the input data is 1D, or shuffle [13] to avoid confusion because it is not a true matrix transpose. In either case, from the linear algebraic perspective T is a block matrix where the copy blocks are placed to copy the correct data from the subset of memory $\{k\}$ to another subset $\{l\}$. From the perspective of any given parallel worker, copying off of that worker is a scatter and copying on to that worker is a gather. The data movement pattern of the adjoint is similar to the forward, however the summations must be respected. If moves are used instead of copies, the adjoint is another all-to-all operation, but the operation is not self-adjoint.

3 Parallel Primitives for Tensors

In the previous section we detailed a general linear algebraic approach to the data movement operations that form the foundation of many parallel codes, and the necessary adjoints for embedding them into an AD framework. In this section, we adapt and build from those primitives to identify five parallel primitives for distributed tensors that are used in our distributed neural network layer functions. In the previous discussion, we made no imposition on the structure of the input or output data – only that we were able to operate on subsets of that data. Here, we begin by describing the general structure of the data and how we represent its distribution on a parallel computer. Then, we detail the five primitives, Broadcast, SumReduce, AllSumReduce, Repartition, and HaloExchange themselves, as well as information on practical implementations, verification, and validation.

3.1 Tensor Partitions

In deep learning, we encounter a variety of tensors which may have arbitrary order (or number of dimensions). For example, an input tensor for a 3D classification problem will usually be of order 5 and have shape $n_b \times n_c \times n_2 \times n_1 \times n_0$, where the dimensions are the size of the batch, channel, and the three feature dimensions, respectively. The actual ordering the data in memory, especially respecting the fast dimension, is not relevant for this discussion, though is highly relevant in a practical implementation. In this work we will assume that input and output tensors will always have the batch dimension first, followed by channel, and then the D feature dimensions. Weight and bias tensors will depend on the operation they are a part of and their shapes will be noted as needed. Occasionally, we will need to inject fictitious, degenerate dimensions to simplify some operations. Regardless of any structure associated with underlying features, we will assume that the tensors themselves are stored by strided multi-dimensional arrays.

In a parallel environment, we will *partition*[4] these tensors along each of their dimensions. A partition P will have the same order (number of dimensions) of the tensor it partitions. For example, a partition of the above input tensor for 3D classification would have the structure

$$P = P_b \times P_c \times P_{D-1} \times \cdots \times P_0, \tag{21}$$

where the channel size n_c is partitioned into P_c parts, etc. The total number of workers required to store a tensor partitioned by P is the product of the number of workers in each dimension.

The choice of optimal partition is highly problem-, computational kernel-, and machine-dependent. P may partition more than one tensor. In general, for simplicity, we take $P_b = 1$, as $P_b > 1$ is data parallelism and is generally embarrassingly parallel. As with tensors, it will occasionally be convenient to promote a partition by adding a degenerate (size 1) dimension.

3.2 Primitives

Using the linear algebraic perspective of parallel data movement, we are able to build five data movement primitives for partitioned tensors. These primitives are higher-order analogs of the standard parallel primitives – their forward behavior is composed from the classical primitives and their adjoint behavior is naturally induced. Here, we define a high-level sketch of the primitives, along with examples describing their usage. Algorithms for these primitives are available in DistDL [20].

3.2.1 Broadcast

Assume a tensor $\boldsymbol{x}$ is partitioned by P_x and that P_y is a partition of the same order as P_x. The distributed-tensor broadcast $B_{\{P_x\}\to\{P_y\}}$ broadcasts $\boldsymbol{x}$ from P_x onto P_y if, for all dimensions k, $P_{x_k} = P_{y_k}$ or $P_{x_k} = 1$. Note, these requirements are a subset of the NumPy broadcasting rules [22]. Subtensors of $\boldsymbol{x}$ are copied P_{y_k} times in dimensions where

4. Partitions are sometimes referred to as a *shard* in related contexts, however as our approach is inspired by classical mesh partitioning, we will follow that terminology.

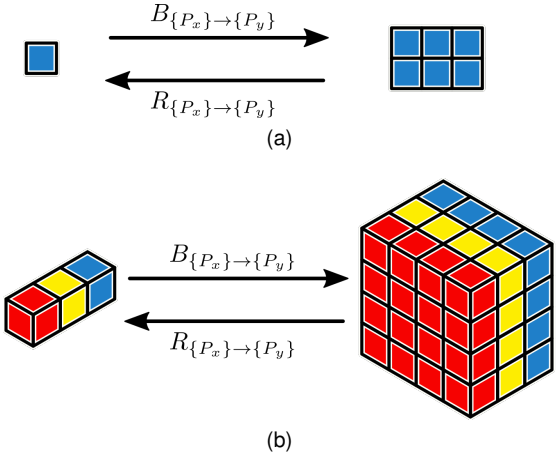

Fig. 1. Broadcast and SumReduce (a) for $P_x = 1 \times 1$ and $P_y = 2 \times 3$ and (b) for $P_x = 1 \times 1 \times 3$ and $P_y = 4 \times 4 \times 3$.

$P_{x_k} = 1$ and no operation occurs when $P_{x_k} = P_{y_k}$. As in [22], we allow implicit extension with dimension 1 partitions. This yields an output tensor $\boldsymbol{y}$ containing $\prod_{k\in K} P_{y_k}$ copies of the input tensor, where $K = \{k|P_{x_k} = 1\}$. For example, in Figure 1a, a tensor with $P_x = 1\times1$ is broadcast to $P_y = 2\times3$, resulting in 6 copies of the original tensor. In Figure 1b, a tensor with $P_x = 1 \times 1 \times 3$ is broadcast to $P_y = 4 \times 4 \times 3$, resulting in 16 copies of the original tensor. The adjoint, $B^*_{\{P_x\}\to\{P_y\}}$ follows from Section 2.2.3 and is a reduction back along the same dimensions.

Details of how to implement these operations are highly specific to the libraries, frameworks, and systems in use. The accompanying package DistDL [20] provides an example of how to construct the required communication patterns using MPI groups and communicators.

3.2.2 SumReduce

Assume a tensor $\boldsymbol{x}$ is partitioned by P_x and that P_y is a partition of the same order as P_x. The distributed-tensor sub-reduction $R_{\{P_x\}\to\{P_y\}}$ reduces $\boldsymbol{x}$ from P_x onto P_y if, for all dimensions k, $P_{x_k} = P_{y_k}$ or $P_{y_k} = 1$. These are essentially the reverse of the broadcast rules above. Subtensors of $\boldsymbol{x}$ are reduced from P_x along dimensions where $P_{y_k} = 1$ and no operation occurs for dimensions where $P_{x_k} = P_{y_k}$. This yields an output tensor $\boldsymbol{y}$ containing $\prod_{k\in K} P_{y_k}$ subtensors, for $K = \{k|P_{x_k} = P_{y_k}\}$. For example, Figure 1a, a tensor with $P_x = 2 \times 3$ is reduced to $P_y = 1 \times 1$, resulting in a single subtensor. In Figure 1b, a tensor with $P_x = 4 \times 4 \times 3$ is broadcast to $P_y = 1 \times 1 \times 3$, resulting in 3 subtensors. The adjoint, $R^*_{\{P_x\}\to\{P_y\}}$ follows from Sec. 2.2.4 and is a broadcast back along the same dimensions.

3.2.3 AllSumReduce

The distributed-tensor all-sum-reduction, $\mathcal{A}_{P_x}$, is conceptually related to the distributed-tensor broadcast and sum-reduction, however, we require only the input partition P_x and a list of the dimensions along which the sum-reduction should occur. These dimensions implicitly define an intermediate partition with the same structure as the output or input partition of $R_{\{P_x\}\to\{P_y\}}$ or $B_{\{P_x\}\to\{P_y\}}$, respectively. However, it can be implemented with a set of carefully selected classical all-reductions and does not require explicit use of reductions or broadcasts.

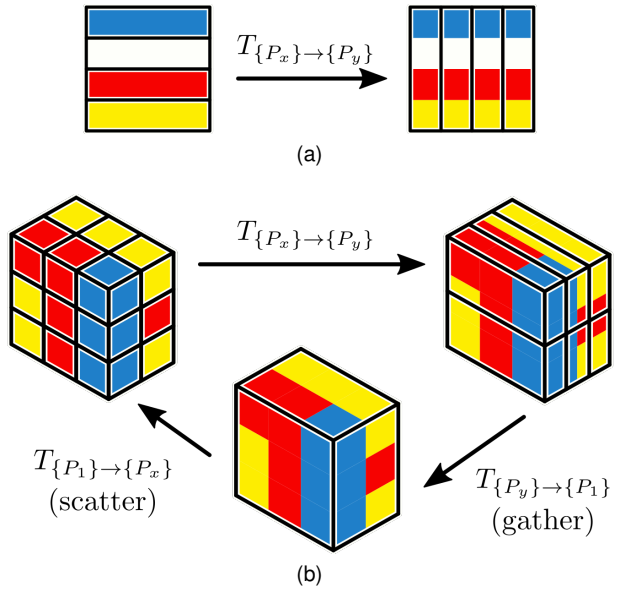

Fig. 2. Repartition operations (a) mimicking classical all-to-all and (b) for general repartitioning, gather, and scatter on distributed tensors.

3.2.4 Repartition

The repartition primitive $T_{\{P_x\}\to\{P_y\}}$ is the distributed-tensor equivalent of the classical all-to-all primitive and takes an input tensor $\boldsymbol{x}$ partitioned by P_x and remaps it to the workers in P_y. To see the relationship with all-to-all, consider the illustration in Figure 2a. Here, $\boldsymbol{x}$ on $P_x = 4\times1$ is equivalent to a row-major, flattened 1D array. The classical all-to-all operation repartitions that data to a $P_y = 1\times4$ partition. The same idea applies in higher-dimensions, as illustrated in Figure 2b, where an order 3 tensor on $P_x = 3 \times 2 \times 2$ is repartitioned onto $P_y = 1 \times 2 \times 3$.

In our development and application, we have found this primitive to be unexpectedly useful. In practice, there is no requirement that P_x and P_y have the same number of workers, so it can be used to intentionally idle some workers. Alternatively, if a tensor is not load balanced, it can be used to redistribute data within the same partition to create load balance. Some computational kernels are more efficient if tensors have specific structure, e.g., cache-blocking in fast dimensions. In these cases repartition can be used to better align the subtensors for the computational kernels. In the absence of parallel I/O, an entire tensor can be read from or written to disk by one worker, essentially on a $P_1 = 1\times\cdots\times1$ partition. Repartitioning P_1 to any other partition is the distributed-tensor scatter operation. Repartitioning from any other partition back to P_1 is the distributed-tensor gather operation, as illustrated in Figure 2b.

3.2.5 HaloExchange

In classical large-scale PDE-driven simulation, a decomposition of the relevant spatial domain allows for effective model parallelism: large variables are distributed to different workers according to the spatial decomposition. When a differential operator is sparse and the solver is explicit, physical interactions are local and minimal data, found near the domain boundaries, needs to be shared between adjacent workers. In neural networks, analogous situations arise for layers featuring small, sliding kernels, such as convolutional

layers and pooling layers. For each worker to correctly apply the computational kernel, this halo region must contain copies of the current data owned by neighboring workers. The exchange of this boundary data between workers is known as a *ghost exchange* or *halo exchange*. This is not a traditional primitive operation in classical distributed computing, rather it is considered a nuisance operation. The halo exchange can be conveniently written in our framework, which allows it to be extended to tensors of arbitrary order and naturally embedded into distributed neural networks.

Due to this irregular structure, we use the linear-algebraic framework to define an algorithm for generalized halo exchange for partitioned tensors, as well as its adjoint. In our algorithm, we neither make any assumptions on the order of the input or output tensors (only that they match) nor the structure of the kernel. As computational load on a given worker is driven by the volume of that worker's output subtensor, we assume that the output tensor is optimally load balanced and derive the necessary input tensor halo sizes in each dimension from there. We assume that the tensors are sensibly decomposed, relative to kernel size, so that halos require data from directly adjacent neighbor workers only. The portion of the distributed tensor that is owned by a worker is the bulk region and the halo exchange ensures that a worker has copies of the necessary portions of its neighbor's bulk regions in its halo region.

All halo regions, both left and right, from all dimensions of an order d tensor may have different thickness. The thicknesses are determined by the minimum and maximum global indices of the worker's output tensor and the size, stride, dilation, and padding parameters of the kernel. From a linear-algebraic perspective, the halo exchange is a sequence of send-receive operations. For efficiency, we assume that the halo exchange is in-place, so the input and output realizations are on the same memory subset. Following the linear-algebraic view, the halo exchange operator for one worker exchanging with its neighboring workers in dimension k is,

$$H_k = K_{\mathbf{T}} C_{\mathbf{U}} C_{\mathbf{E}} C_{\mathbf{P}} K_{\mathbf{S}}, \tag{22}$$

where the setup operator, $K_{\mathbf{S}}$, clears the exchange buffers; the pack operator, $C_{\mathbf{P}}$, copies from the bulk region to the send buffer; the exchange operator, $C_{\mathbf{E}}$, copies from the current worker's send buffer to the neighboring worker's receive buffer, and vice versa; the unpack operator, $C_{\mathbf{U}}$, copies from the receive buffer to the halo region; and the teardown operator, $K_{\mathbf{T}}$, clears on the exchange buffers. For order d tensors, the exchange is performed one dimension at a time, in a nested manner to ensure proper communication of data in corner cases [23]. Thus, the full exchange operator is,

$$H = H_{d-1} \cdots H_1 H_0, \tag{23}$$

with corresponding adjoints,

$$H_k^* = K_{\mathbf{S}}^* C_{\mathbf{P}}^* C_{\mathbf{E}}^* C_{\mathbf{U}}^* K_{\mathbf{T}}^*, \tag{24}$$

$$H^* = H_0^* H_1^* \cdots H_{d-1}^*. \tag{25}$$

In practice, clearing the exchange buffers is implicit. We illustrate the generalized, unbalanced forward and adjoint halo exchanges in Appendix B. This view justifies an observation that has been used in production PDE-constrained optimization codes for some time [24]: in the adjoint of halo exchange, there is an *add* operation into the bulk tensor. This is ultimately because the three copy operations, at the center of each part of the exchange, copy data from the bulk of one worker to the halo region of another and the ensuing adjoint phase must produce an add.

3.3 Implementation & Validation

In our distributed deep learning library, DistDL, we have provided implementations of many necessary primitives for PyTorch `autograd` [25]. In parallel environments, verification of correctness using numerical gradient validation is difficult. Fortunately, data movement operations are linear and we can exploit the fact that the forward operator is its own Jacobian, $F = \mathcal{F}$, and the definition of the adjoint to establish an equivalent test for correctness. We say that an implementation of F^* is coherent with F if the adjoint test is satisfied $\forall \boldsymbol{x} \in \mathbb{F}^m,\ \forall \boldsymbol{y} \in \mathbb{F}^n$,

$$\frac{|\langle F\boldsymbol{x}, \boldsymbol{y}\rangle_{\mathbb{F}^n} - \langle \boldsymbol{x}, F^*\boldsymbol{y}\rangle_{\mathbb{F}^m}|}{\max\left\{\|F\boldsymbol{x}\|_{\mathbb{F}^n}\|\boldsymbol{y}\|_{\mathbb{F}^n}, \|\boldsymbol{x}\|_{\mathbb{F}^m}\|F^*\boldsymbol{y}\|_{\mathbb{F}^m}\right\}} < \varepsilon, \tag{26}$$

where ε is related to machine-ε and realizations from $\mathbb{F}^m$ and $\mathbb{F}^n$ are, in practice, drawn randomly.

4 Domain-decomposed Layer Functions

Using the distributed tensor primitives in Section 3 or direct application of the classical primitives in Section 2.2, we can define distributed algorithms for common neural network layers. We classify these algorithms, broadly, by the way mathematical operations and learnable parameters interact with input data: hyper-locally, locally, and globally. Hyper-local functions, e.g., activation functions and dropout functions, are characterized by their point-wise application and no data movement is required to apply them in parallel. Local functions require information from other nearby features (usually no further away than one neighboring worker) and are characterized by structured sparsity in the operator at the core of the function. Global functions may require data from any part of the distributed tensor and are characterized by dense linear operations.

4.1 Local Functions

4.1.1 Down- or Up-sampling Layers

Among this class of layers, *pooling* layers are the most straightforward to parallelize. Assume the input and output tensor $\boldsymbol{x}$ and $\boldsymbol{y}$ have feature-space dimension D and overall shape $n_b \times n_c \times m_{D-1} \times \cdots \times m_0$ and $n_b \times n_c \times n_{D-1} \times \cdots \times n_0$, where n_b, n_c, and m_k and n_k are the batch, channel, and feature dimensions, respectively. For both tensors, distributed over a partition P with shape $1 \times P_c \times P_{D-1} \times \cdots \times P_0$, the distributed pooling algorithm and the adjoint of its Jacobian are in Figure **??**.

The algorithm does not rely on linearity in the pooling operation, so any pooling operation is permitted, including average and max pooling. The halo exchange H is strongly dependent on the pooling kernel size, stride, dilation, and padding parameters. In practice, padding shims are required to account cases where halos are needed or extra input is provided (e.g., those in Appendix B).

Forward Pooling	Adjoint Pooling
1: Input: $\boldsymbol{x}$	1: Input: $\delta\boldsymbol{y}$
2: $\boldsymbol{x} \leftarrow H\boldsymbol{x}$	2: $\delta\boldsymbol{x} \leftarrow [\delta\text{POOL}]^*(\delta\boldsymbol{y})$
3: $\boldsymbol{y} \leftarrow \text{POOL}(\boldsymbol{x})$	3: $\delta\boldsymbol{x} \leftarrow H^*\delta\boldsymbol{x}$
4: Output: $\boldsymbol{y}$	4: Output: $\delta\boldsymbol{x}$

Fig. 3. Forward and adjoint distributed pooling algorithms. POOL is a pooling function and $[\delta\text{POOL}]^*$ is the adjoint of its Jacobian.

Forward Convolution	Adjoint Convolution
1: Input: $\boldsymbol{x}$	1: Input: $\delta\boldsymbol{y}$
2: $\boldsymbol{x} \leftarrow H\boldsymbol{x}$	2: $\delta\hat{\boldsymbol{y}} \leftarrow B_{\{P_y\}\to\{P_w\}}\delta\boldsymbol{y}$
3: $\hat{\boldsymbol{w}} \leftarrow B_{\{P_r\}\to\{P_w\}}\boldsymbol{w}$	3: $\delta\hat{\boldsymbol{w}}, \delta\hat{\boldsymbol{b}}, \delta\hat{\boldsymbol{x}} \leftarrow [\delta\text{CONV}]^*(\delta\hat{\boldsymbol{y}})$
4: $\hat{\boldsymbol{b}} \leftarrow B_{\{P_r\}\to\{P_w\}}\boldsymbol{b}$	4: $\delta\boldsymbol{x} \leftarrow R_{\{P_w\}\to\{P_x\}}\delta\hat{\boldsymbol{x}}$
5: $\hat{\boldsymbol{x}} \leftarrow B_{\{P_x\}\to\{P_w\}}\boldsymbol{x}$	5: $\delta\boldsymbol{b} \leftarrow R_{\{P_w\}\to\{P_r\}}\delta\hat{\boldsymbol{b}}$
6: $\hat{\boldsymbol{y}} \leftarrow \text{CONV}(\hat{\boldsymbol{w}}, \hat{\boldsymbol{b}}; \hat{\boldsymbol{x}})$	6: $\delta\boldsymbol{w} \leftarrow R_{\{P_w\}\to\{P_r\}}\delta\hat{\boldsymbol{w}}$
7: $\boldsymbol{y} \leftarrow R_{\{P_w\}\to\{P_y\}}\hat{\boldsymbol{y}}$	7: $\delta\boldsymbol{x} \leftarrow H^*\delta\boldsymbol{x}$
8: Output: $\boldsymbol{y}$	8: Output: $\delta\boldsymbol{x}$

Fig. 4. Forward and adjoint distributed convolution algorithms. CONV is a convolution function and $[\delta\text{CONV}]^*$ is the adjoint of its Jacobian.

Upsampling may be handled by a variety of interpolating layers or even so-called "transposed convolutions". In the case of the former, the algorithm is very similar to that of pooling. A transposed-convolution approach will be similar, however, due to learnable parameters, it shares algorithmic elements with the classical convolution.

4.1.2 Convolutional Layers

Convolutional layers are a frequent target for parallelization [11] and were targeted by [13] to improve strong parallel scalability. Ultimately, we seek weak scalability as we are interested in problems where the input tensors can have billions of features. Anticipating that these tensors will be decomposed over potentially thousands of workers, we avoid the explicit all-reduce operation often used in the gradient update. Instead, we formulate the layer so that the all-reduce appears implicitly: a broadcast in the forward implementation naturally induces a sum-reduce in the adjoint phase.

Assume a similar structure as for the pooling layer, except that the learnable weights $\boldsymbol{w}$ have shape $n_{co} \times n_{ci} \times k_{D-1} \times \cdots \times k_0$, where n_{ci} and n_{co} are the input and output channel sizes and k_i is the kernel size, and are distributed over partition P_r with shape $P_{co} \times P_{ci}$. To avoid multiple counting of the bias, assume that the learnable part of the bias is only present on one $P_{co} \times 1$ subpartition of P_r. For $P_x = 1 \times 1 \times P_{ci} \times P_{D-1} \times \cdots \times P_0$, $P_y = 1 \times P_{co} \times 1 \times P_{D-1} \times \cdots \times P_0$,[5] and a work partition $P_w = 1 \times P_{co} \times P_{ci} \times P_{D-1} \times \cdots P_0$, the generalized distributed convolution layer and the adjoint of its Jacobian are given in Figure 4.

If the input tensor is distributed over the feature-space exclusively, then the weights are not distributed over the channels and the algorithm can be significantly simplified by removing the broadcasts and reductions in steps 5 and 7 of the forward algorithm (and corresponding operations in steps 2 and 4 of the adjoint). If the tensors are distributed over the channels exclusively, we remove the need for the halo exchange and the broadcasts in steps 2, 3 and 4 of the forward (and corresponding operations in steps 5, 6, and 7 of the adjoint). However, this form of the algorithm is quite greedy with respect to need of workers and further research is required more efficient resource utilization.

5. The additional dimensions aid the broadcasting pattern but do not impact the result.

Forward Affine	Adjoint Affine
1: Input: $\boldsymbol{x}$	1: Input: $\delta\boldsymbol{y}$
2: $\hat{\boldsymbol{x}} \leftarrow B_{\{P_x\}\to\{P_w\}}\boldsymbol{x}$	2: $\delta\hat{\boldsymbol{y}} \leftarrow B_{\{P_y\}\to\{P_w\}}\delta\boldsymbol{y}$
3: $\hat{\boldsymbol{y}} \leftarrow \text{AFF}(\hat{\boldsymbol{w}}, \hat{\boldsymbol{b}}; \hat{\boldsymbol{x}})$	3: $\delta\hat{\boldsymbol{w}}, \delta\hat{\boldsymbol{b}}, \delta\hat{\boldsymbol{x}} \leftarrow [\delta\text{AFF}]^*(\delta\hat{\boldsymbol{y}})$
4: $\boldsymbol{y} \leftarrow R_{\{P_w\}\to\{P_y\}}\hat{\boldsymbol{y}}$	4: $\delta\boldsymbol{x} \leftarrow R_{\{P_w\}\to\{P_x\}}\delta\hat{\boldsymbol{x}}$
5: Output: $\boldsymbol{y}$	5: Output: $\delta\boldsymbol{x}$

Fig. 5. Forward and adjoint distributed affine algorithms. AFF is an affine function and $[\delta\text{AFF}]^*$ is the adjoint of its Jacobian.

4.2 Global Functions

4.2.1 Dense Linear Layers

Dense layers are characterized by full-connection between input and output features, often through the *affine* function $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{W}$ is a dense $n_{fo} \times n_{fi}$ matrix and n_{fo} and n_{fi} are the number of output and input features. Optimal parallelism in such layers is found through a distributed generalized matrix-matrix multiplication, or GEMM, algorithm [26]. Optimal GEMM structure and performance is dependent on the computing environment, the size and rank of the tensors, and is an area of open research. We present an implementation based on the primitives above, recognizing that depending on the partitioning of workers, production distributed GEMM implementations may have different flavor. A distributed affine layer has similar setup as the distributed convolution, except that the weight tensor is $n_{fo} \times n_{fi}$, where n_{fo} and n_{fi} are the number of features in and out, and is distributed on $P_w = P_{fo} \times P_{fi}$ partition. The learnable bias, of size n_{fo}, is present only on one $P_{fo} \times 1$ subset of P_w, to avoid any issue with multiple-counting of the bias. For simplicity, we assume that the layer is fully-connected and that input and output tensors $\boldsymbol{x}$ and $\boldsymbol{y}$ have size $n_b \times n_{fi}$ and $n_b \times n_{fo}$, and are distributed on partitions P_x and P_y, with shape $1 \times P_{fi}$ and $1 \times P_{fo}$, respectively. The extension to arbitrary tensor dimensions is similar to the distributed convolution layer. The algorithm and the adjoint of its Jacobian are given in Figure 5.

Algorithms for channel-only distributed convolutions are quite similar to those for distributed dense linear operations, so future research in either should inform improvements to the other. For example, like the generalized convolution, this algorithm is greedy with respect to the require workers, so, e.g., Cannon's algorithm, should be considered for application to networks that traditionally use many dense layers, such as recurrent networks or transformers.

4.2.2 Normalization Layers

Normalization layers, which normalize feature statistics, are critical to neural network training to prevent catastrophic growth or decay in gradients. Strategies, such as batch normalization [27] and group normalization [28], which is useful when only small minibatches are feasible [29], are common. Normalization layers all have similar structure: feature means and variances are computed along some

dimensions of the tensor and all values in the tensor are normalized by the resulting statistics. In a distributed environment, the statistics are required on all workers, and thus must be computed by way of an all-reduction. For example, in distributed batch normalization, this is performed along locally along the channel dimension of each worker's subtensor and then globally via an AllSumReduce operation along the partition's channel dimension. The correct adjoint all-reduction is naturally induced during the training process.

4.3 Loss Functions

While they require global information over the entire output tensor, loss functions are computed point-wise, up to a sum-reduction. Losses in a distributed environment are thus nearly embarassingly parallel, but do require a distributed sum reduction to assemble the final loss. Some distributed deep learning models may compute this via an all-reduction so that all workers have the loss value, but if it is not manually short circuited, this *should* induce a redundant all-reduction in the adjoint pass. In our approach, to preserve mathematical consistency, we allow the required broadcast (second half of the all-reduction) to be induced naturally in the adjoint from the sum-reduction in the forward evaluation.

5 Domain-decomposed Neural Networks & Distributed Performance

We have implemented a number of the above layers [20] to demonstrate of the effectiveness of our model. These implementations explicitly rely on PyTorch's underlying implementation of the base layer function. Using our distributed convolution, pooling, and affine layers, as well as some repartitioning layers as glue, we have implemented distributed versions of the Lenet-5 convolutional neural network [30], the VGG [31] family of networks, the ResNet [32] family of networks, and U-nets [33]. While the first two are optimized around small inputs, and therefore benefit little from domain decomposition, we provide them as small examples [20] of the concepts in this paper. Here, the convolutional nature of feature-identification makes it simple to parallelize, while the dense classifier is more difficult. In the ensuing discussion and experiments, we focus on ResNets, which are inherently easier to domain decompose due to their convolutional nature and their intrinsic relationship with differential equations [34], and U-nets, which are notoriously difficult to apply efficiently to large, 3D problems [29]. DistDL's programming interface is similar to that of PyTorch and adapting sequential PyTorch networks to the domain decomposition model requires but a small number of changes. Thus, with some knowledge of the target cluster and available resources, it is relatively simple to modify existing training code to take advantage of model parallelism through domain decomposition.

5.1 Inputs

To preserve performance, loading and storage of distributed network input must also be performed in parallel. When this is not possible, a single worker can load the data and repartition (scatter) it to the other workers. In this scenario, care must be taken to avoid poor memory load balance.

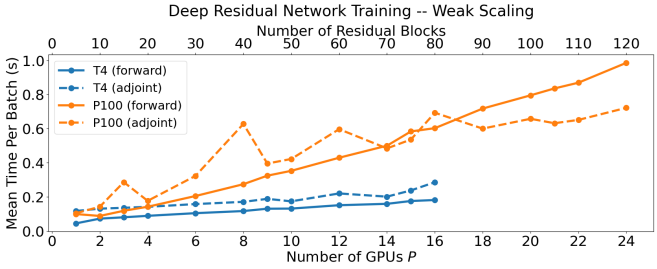

Fig. 6. Weak scaling study for deep residual network in DistDL, profiled on T4 and P100 GPUs with $1{,}400 \times 1{,}400$ inputs. Solid and dashed lines are mean time per batch for forward and adjoint passes.

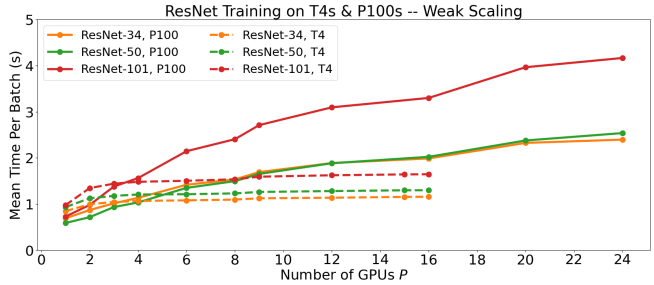

Fig. 7. Weak scalability of ResNet34, ResNet50, and ResNet101 on up to 24 NVIDIA P100 and up to 16 NVIDIA T4 GPUs. Initial problem sizes are given in Table 1.

Additionally, the ground truth classes, segmentations, etc., must also be distributed similarly. In our examples, we assume that the input data can be loaded in parallel using, e.g., MPI I/O or generated in parallel to mimic this behavior.

5.2 Residual Neural Networks

Residual neural networks (resnets) are composed of *blocks*, in which the input $\boldsymbol{x}$ to the block is added to the output of some operator $\mathcal{F}$ via a "skip connection," [32]

$$\boldsymbol{x}_{i+1} = \mathcal{F}[\boldsymbol{x}_i] + \boldsymbol{x}_i. \tag{27}$$

Skip connections regularize gradient calculation in very deep neural networks, but with this depth can come high memory consumption, especially for large inputs. Additionally, due to their relationship with differential equations [34] which are well-studied in distributed memory environments, resnets are an ideal candidate for demonstrating the capabilities of a generalized distributed deep learning framework.

Using DistDL, we have implemented distributed variants of all of the classes of ResNet [32], as well as a more

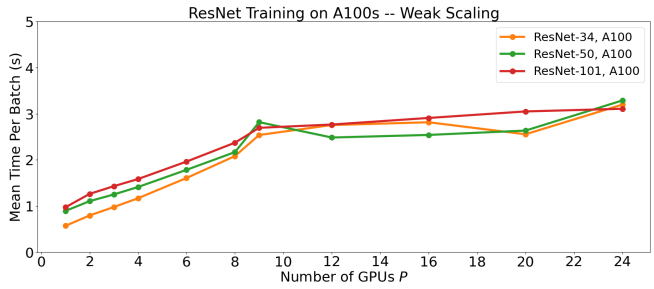

Fig. 8. Weak scalability of ResNet34, ResNet50, and ResNet101 on up to 24 NVIDIA A100 GPUs. Initial problem sizes are given in Table 1.

	ResNet34	ResNet50	ResNet101
P100	$2{,}200^2$	$1{,}300^2$	$1{,}100^2$
T4	$2{,}048^2$	$1{,}500^2$	$1{,}200^2$
A100	$3{,}900^2$	$3{,}500^2$	$3{,}000^2$

TABLE 1
Input size per-worker for each GPU (parallel task) and ResNet model.

general resnet capable of scaling to arbitrary input size and depth. Each block in the general network consists of two convolutional layers with 10 input and output channels, ReLU activation, and batch normalization. We conducted scalability studies on Virginia Tech's TinkerCliffs and Infer GPU clusters. Each of up to 3 A100 nodes on TinkerCliffs contains two 64-core AMD EPYC 7742 processors, 2TB of RAM, 8 NVIDIA A100 GPUs, and are connected using HDR Infiniband. Each of up to 16 T4 nodes on Infer contains two 16-core Intel Xeon Gold 6130 processors, 192GB of RAM, 1 NVIDIA T4 GPUs, and are connected using EDR Infiniband. Each of up to 24 P100 nodes on Infer contains two 12-core Intel Xeon E5-2680 processors, 512GB of RAM, 2 NVIDIA P100 GPUs, and are connected using ethernet.

Results of a weak scalability study on ResNet34, ResNet50, and ResNet101 are shown in Figure 7 for P100 and T4 GPUs and in Figure 8 for A100 GPUs. Per-GPU input sizes are in Table 1. These sizes were chosen to fill available GPU memory, yielding a maximum input image size of $12{,}000 \times 18{,}000$ for ResNet101. As this is a 2D study, partitions are 2D and as square as possible, i.e., for 24 GPUs the partition is 4×6 not 3×8. In the presence of high GPU memory bandwidth and fast interconnect (e.g., on the T4 cluster), we observe excellent scalability. For lower bandwidth and slower interconnect (e.g., the P100 cluster), we observe an expected performance degradation. In the A100 study, where the first 8 GPUs are on a single node, we observe an increase in run time until a 2$^{\text{nd}}$ compute node is required. From 1 to 8 GPUs, the cost increases due to increases in data movement within the node. For 9 workers, the maximum amount of communication is achieved – the middle worker has to communicate with 4 neighbors. We observe similar behavior in for the T4 study, but it is less pronounced as network effects manifest after only 2 GPUs.

We also investigated scalability with respect to depth of the network for resnets with the above block structure, depth $d = 5 \times P$, where P is the number of GPUs, and input of size $1{,}400 \times 1{,}400$. Results of the performance study are shown in figure 6, where the y-axis is the mean time to train 100 batches of batch size 1. As above, we observe better performance on the T4 cluster, likely for similar reasons. As the network deepens, per-halo communication volume is quite small, and the total number of communications on the network is high, reducing performance. We anticipate that the performance of very deep distributed networks can be improved by leveraging recent results in layer-parallel training [35], and by integrating multiple-halos to maximize the computation per communication.

5.3 3D U-nets

U-Nets were introduced by [33] for application to 2D image segmentation. These networks are also applicable to 3D or volumetric segmentation, yet the curse of dimensionality manifests itself through enormous memory requirements [29]. We've implemented a 3D version of classical depth 5 U-Net of [33] using DistDL. The network has fundamentally the same architecture as the original, except that we incorporate batch normalization after each convolutional block and replacing the transposed-convolutions in the up-cycle with linear interpolation followed by 1D convolutions. Inputs to the network are synthetic 3D phantoms that can be expressed parametrically. In this way, we could arbitrarily scale the inputs, and thus the computation and memory requirements, with available parallel workers. Practical problems of interest in our research have at least $1{,}024^3$ inputs, with potential long-range influence across the entire domain, so our objective was to scale to at least that size.

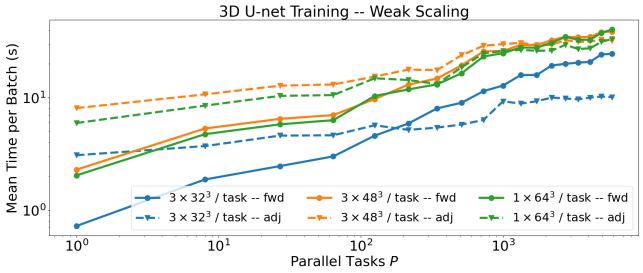

Fig. 9. Weak scaling study for 3D U-net in DistDL. Solid and dashed lines are mean time per batch for forward and adjoint passes.

We performed a weak scaling study on this network using up to 183 nodes of Virginia Tech's TinkerCliffs CPU cluster. Each node contains two 64-core AMD EPYC 7702 processors, 256GB of RAM, and are connected using HDR Infiniband. To optimally utilize the EPYC CPUs, we assigned each MPI task 4 OpenMP threads, meaning one node could support up to 32 workers. This also provided each worker with 8GB of RAM, similar to what would be available for an older GPU. For each of $P = p^3$ workers, for $p = 1, 2, \ldots, 18$, we performed a weak scaling study on three input configurations: $n = 3 \times 32^3$, $n = 3 \times 48^3$, and $n = 1 \times 64^3$ inputs per worker, where the first number is the batch size and the second is the 3D feature size. The largest inputs per worker represent the approximate upper bound before some task exceeded memory limitations. At the largest scale, this is an input of size $1{,}152^3$ distributed over $5{,}832$ parallel tasks and $23{,}328$ cores.

Results of the performance study are shown in Figure 9, where the y-axis is the mean time to train 10 batches at the given size. For a more representative sample, we discard the time for the initial batch as there is some overhead in the first pass that is not present for subsequent batches. We observe excellent scalability in the adjoint phase, especially for very large problem sizes – increasing the workload does not commensurately increase the execution time. In the forward phase, scaling is less ideal, but further investigation is required to fully understand this behavior. Considering the above results for resnet scaling in GPU environments, we anticipate similar performance on GPU clusters, provided they have sufficient memory.

6 Discussion & Outlook

We have presented a linear-algebraic framework for expressing data movement in distributed deep learning. With this approach, we have constructed a set of parallel primitives for high-order tensors (and their adjoints) which can be used to implement common neural network functions. Using DistDL, a concrete realization of the principles developed in this paper, we have empirically demonstrated the simplicity and effectiveness of this data movement model on large-scale neural networks on both CPU and hybrid CPU-GPU clusters, using MPI to perform the underlying communication. We believe that this effort provides a path toward democratization of HPC technology in deep learning, the same way that the broad availability of PyTorch, Tensorflow, and cloud computing technologies have democratized classical ML.

Future development of these algorithms will be highly dependent on the target distributed computing architecture and the target DNN structure. For example, modern GPUs and custom ML accelerator chips have access to Remote Direct Memory Access [36] (RDMA) technology, affording high-bandwith direct communication between accelerators. Our data movement model simply abstracts classical data movement concepts and can easily absorb these, or alternative interfaces to GPU-to-GPU communication technologies, whether or not they are accessible through MPI [37]. Future developments will improve both the forward and adjoint performance of the distributed network layers by overlapping work in the data movement and computational phases, however, this may be challenging to express as cleanly as the current implementation while maintaining re-entrance. We anticipate that increasing the number of layers that can be computed without a halo-exchange, aggressive idling of workers in narrow network components, integration of pipelining, and improvements to dense layers will provide additional improvements to parallel efficiency. Optimal deployment of these new directions will require further study on optimal network architecture design for distributed environments. With many technologies at play, care must be taken to design networks so that computation volumes, data movement surfaces, and hardware properties are properly optimized to the needs of the problem and the available hardware. The initial studies presented here are promising, yet significant improvements to efficiency can still be made.

Our approach to parallelism in deep learning is readily applicable to the largest problems in classical deep learning, such as natural language processing, volumetric segmentation, and video processing, as well as emergent extreme-scale problems in scientific machine learning [2], such those posed through physics-informed neural networks [38], [39] and extreme-scale physics-driven inversion [14], [40]. In these later contexts, following our inspiration from classical HPC data movement, we believe our model will help bridge the gap between parallelism in deep learning and the standard HPC software tools available for scientific simulation.

Acknowledgments

RJH is supported by the US Department of Energy Office of Science under award DE-SC0022041. TJG was supported by the Luther and Alice Hamlett Undergraduate Research Support program. The authors especially thank Justin Krometis and Matthew Brown from Virginia Tech's Advanced Research Computing (ARC) for system support and coordination of large-scale tests on ARC clusters. They also thank Ananiya Admassu, Mason Beahr, and Sarah Kauffman, a Virginia Tech computational modeling and data analytics Capstone team, for their project studying feasibility of 3D distributed U-nets.

References

[1] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[2] N. Baker *et al.*, "Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence," USDOE Office of Science (SC), Washington, D.C. (United States), Tech. Rep., Feb. 2019.

[3] A. Karpatne *et al.*, "Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 10, pp. 2318–2331, Oct. 2017.

[4] T. Kurth *et al.*, "Exascale Deep Learning for Climate Analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Dallas, TX, USA: IEEE, Nov. 2018, pp. 649–660.

[5] L. Yang *et al.*, "Highly-scalable, Physics-Informed GANs for Learning Solutions of Stochastic PDEs," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov. 2019, pp. 1–11.

[6] N. Ketkar, "Introduction to PyTorch," in *Deep Learning with Python: A Hands-on Introduction*, N. Ketkar, Ed. Berkeley, CA: Apress, 2017, pp. 195–208.

[7] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[8] Microsoft, "Msr-fiddle/pipedream, https://github.com/msr-fiddle/pipedream," Mar. 2020.

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv:1909.08053 [cs]*, Oct. 2019.

[10] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," *arXiv:1802.05799 [cs, stat]*, Feb. 2018.

[11] N. Shazeer, "Mesh-TensorFlow: Model Parallelism for Supercomputers (TF Dev Summit '19)," 2019.

[12] N. Shazeer *et al.*, "Mesh-TensorFlow: Deep Learning for Supercomputers," *arXiv:1811.02084 [cs, stat]*, Nov. 2018.

[13] N. Dryden, N. Maruyama, T. Benson, T. Moon, M. Snir, and B. Van Essen, "Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 210–220.

[14] A. Richardson, "Seismic Full-Waveform Inversion Using Deep Learning Tools and Techniques," *arXiv:1801.07232 [physics]*, Jan. 2018.

[15] U. Naumann, *The Art of Differentiating Computer Programs*. SIAM, 2012.

[16] R. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[17] C. Bischof, L. Roh, and A. Mauer, "ADIC : An extensible automatic differentiation tool for ANSI-C." *Software-Pract. Exper.*, vol. 27, no. ANL/MCS-P626-1196, Dec. 1997.

[18] L. Hascoët and V. Pascual, "The Tapenade automatic differentiation tool: Principles, model, and specification," *ACM Transactions on Mathematical Software*, vol. 39, no. 3, pp. 20:1–20:43, 2013.

[19] L. Clarke, I. Glendinning, and R. Hempel, "The MPI Message Passing Interface Standard," in *Programming Environments for Massively Parallel Distributed Systems*, ser. Monte Verità, K. M. Decker and R. M. Rehmann, Eds. Basel: Birkhäuser, 1994, pp. 213–218.

[20] R. J. Hewett, T. Grady, and J. Merizian, "DistDL: Distributed Deep Learning for PyTorch," Sep. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5360401

[21] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, Sep. 2011.
[22] B. Numpy, "Broadcasting — NumPy v1.18 Manual," https://numpy.org/doc/1.18/user/basics.broadcasting.html, 2020.
[23] W. Gropp, "Lecture 25: Strategies for Parallelism and Halo Exchange," 2016.
[24] X. Zhang, X.-Y. Huang, and N. Pan, "Development of the Upgraded Tangent Linear and Adjoint of the Weather Research and Forecasting (WRF) Model," *Journal of Atmospheric and Oceanic Technology*, vol. 30, no. 6, pp. 1180–1188, Feb. 2013.
[25] A. Paszke *et al.*, "Automatic differentiation in PyTorch," Oct. 2017.
[26] G. Bosilca *et al.*, "Tensor contraction on distributed hybrid architectures using a task-based runtime system," p. 10, 2018.
[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 448–456.
[28] Y. Wu and K. He, "Group normalization," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 3–19.
[29] T. LaBonte, C. Martinez, and S. A. Roberts, "We Know Where We Don't Know: 3D Bayesian CNNs for Credible Geometric Uncertainty," *arXiv:1910.10793 [cs, eess]*, Apr. 2020.
[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.
[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385
[33] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, ser. Lecture Notes in Computer Science, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Springer International Publishing, 2015, pp. 234–241.
[34] L. Ruthotto and E. Haber, "Deep Neural Networks Motivated by Partial Differential Equations," *Journal of Mathematical Imaging and Vision*, vol. 62, no. 3, pp. 352–364, Apr. 2020.
[35] S. Günther, L. Ruthotto, J. B. Schroder, E. C. Cyr, and N. R. Gauger, "Layer-parallel training of deep residual neural networks," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 1, pp. 1–23, 2020.
[36] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, Jun. 2004.
[37] NVIDIA, "MPI Solutions for GPUs," https://developer.nvidia.com/mpi-solutions-gpus, Apr. 2017.
[38] Y. Yang and P. Perdikaris, "Adversarial uncertainty quantification in physics-informed neural networks," *Journal of Computational Physics*, vol. 394, pp. 136–152, Oct. 2019.
[39] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, Feb. 2019.
[40] L. Yang *et al.*, "Highly-scalable, Physics-Informed GANs for Learning Solutions of Stochastic PDEs," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov. 2019, pp. 1–11.

Russell J. Hewett received the B.S. *in honors* degree in computer science from the Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, in 2005, and the Ph.D. degree in computer science, with a certificate in computational science & engineering from the University of Illinois, Urbana-Champaign (UIUC), in 2011.

Since 2018, he is Assistant Professor in the Department of Mathematics at Virginia Tech, where is also affiliate faculty in Computational Modeling and Data Analytics. Before joining Virginia Tech, from 2014-2018 he was research scientist and R&D project manager at Total SA's research office in Houston, TX. From 2011-2014, he was Postdoctoral Associate in the Department of Mathematics at MIT, and by courtesy, with Earth Resources Laboratory. He has made contributions to computational aspects of inverse problems in solar physics and geophysics, has been a member of the board of directors for the SunPy project since 2014 and has released open source software for seismic inversion and parallel deep learning. His research interests lie at the intersection of high-performance computing, deep learning, and physics-driven inverse problems.

In 2021, he received the Early Career Research Program award from the Department of Energy's Office of Science. From 2008-2010 he was a NASA Graduate Student Research Program (GRSP) fellow.

Thomas J. Grady II received the B.S. degree in applied mathematics and the B.S. degree in computational modeling and data analytics from Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, in 2020.

Since 2021 he is a Ph.D. student in computational science and engineering at the Georgia Institute of Technology (Georgia Tech) working on large-scale distributed machine learning (ML) for modeling and inverse problems of partial differential equations (PDEs) as part of the Seismic Laboratory for Imaging and Modeling. Before entering Georgia Tech, from 2019-2021, he was a co-founder of the medical technology startup Radian Health, where he worked as a machine learning and product engineer. His contributions to Radian Health helped them to achieve and improve a working prototype of their technology during and after the Roanoke Business Accelerator Program. His research interests lie in the intersection of large scale ML, numerical methods for PDEs, inverse problems, and high performance computing; and how the combination of these fields can bring about something greater than the sum of its parts.

Jacob Merizian Jacob Merizian received the B.S. degree in computer science and the B.S. degree in applied discrete mathematics from from Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, in 2021. He performed undergraduate research on algorithms in high performance computing and on brain-computer-interfaces in neuroscience. He was a leader in the Virginia Tech Programming Team and represented the school at regional programming competitions.

He has worked at Microsoft, where he worked on real-time visualizations for Azure Kubernetes clusters. Since 2019, he is co-founder of Radian Health, a medical technology start-up for increasing patient engagement in physical therapy with natural language processing and computer vision tools. His research interests are in the interpretability of artificial and biological intelligence.

APPENDIX A
DERIVATIONS

A.1 Construction of move

The in-place move operator, $M_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$, takes input $\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_a \\ \boldsymbol{x}_b\end{bmatrix} \in \mathbb{F}^m$, produces output $\hat{\boldsymbol{x}} = \begin{bmatrix}\boldsymbol{0}_a \\ \boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^m$, and

$$\begin{aligned} M_{a\to b} &= \begin{bmatrix} O_a & O_b \\ I_a & O_b \end{bmatrix} \\ &= \begin{bmatrix} O_a & O_b \\ O_a & I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ I_a & I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ O_a & O_b \end{bmatrix} = K_a S_{a\to b} K_b. \end{aligned}$$

The adjoint is,

$$M^*_{a\to b} = (K_a S_{a\to b} K_b)^* = K_b^* S^*_{a\to b} K_a^* = K_b S_{b\to a} K_a.$$

The out-of-place move operator, $M_{a\to b} : \mathbb{F}^m \to \mathbb{F}^{m'}$, takes input $\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^m$, produces output $\hat{\boldsymbol{x}} = \begin{bmatrix}\boldsymbol{x}_a\end{bmatrix} \in \mathbb{F}^{m'}$, where $\mathbb{F}^{m'}$ is a different memory subset of the same size, and

$$M_{a\to b} = \begin{bmatrix} O_a \\ I_a \end{bmatrix} = \begin{bmatrix} O_a I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ I_a & I_b. \end{bmatrix} \begin{bmatrix} I_a \\ O_a \end{bmatrix} = D_a S_{a\to b} A_b,$$

Here, the adjoint is,

$$M^*_{a\to b} = (D_a S_{a\to b} A_b)^* = A_b^* S^*_{a\to b} D_a^* = D_b S_{b\to a} A_a.$$

APPENDIX B
HALO EXCHANGE

B.1 Irregularly structured halo regions

The subsequent examples show the impact of different kernel parameters and tensor partitions on the halo regions for some different kernels and input sizes. In each case, the driver for the computational load balance is the output distribution. Consequently, absent any padding, assuming the input comes from another layer with the same property, the input is also balanced. While we show 1D examples for simplicity of presentation, the same patterns emerge in multidimensional cases, with more complex interactions between the halo regions.

In the following figures, bulk regions are illustrated in solid black lines and halo regions are given in dashed lines. The numbers, arrows, and braces illustrate the access pattern. Directional arrows indicate the input influence on output, numbers in the input tensor are indices, and numbers in the output tensor are the input index at the root of the kernel for that output index. We have selected these examples for their representative behavior and the kernel parameters are commonly used in many DNNs.

B.2 "Normal" convolution

Assume a centered convolution kernel with size $k = 5$, input tensor size $n = 11$, partition size $P = 3$, and assume a zero-padding of width 2 is implicitly added to the input boundaries. In Figure B10a we illustrate that this situation yields the "normal", uniform halo sizes that would, for example, appear in a finite difference simulation.

B.3 Unbalanced convolution

Assume a centered convolution kernel with size $k = 5$, input tensor size $n = 11$, partition size $P = 3$, and assume a no padding is added to the input boundaries. Then the output length is $m = 7$. In Figure B10b we illustrate that this situation yields unbalanced halo sizes, where the first and last workers have large, one-sided halos and the middle worker has small, balanced halos.

B.4 Simple unbalanced pooling

Assume a right-looking pooling kernel with size $k = 2$, stride $s = 2$, input tensor size $n = 10$, partition size $P = 3$, and no padding or dilation. In Figure B10c we illustrate that this situation yields both unbalanced halos and unnecessary data in the input tensor, which manifests as a negative-sized halo region. For the first worker, there is no halo. For the second worker, only the right-side has a halo, with size 1. The last worker does not have any halo, but to produce the *same* output as the sequential case for this input, the first entry of the input tensor must be removed when the input is provided to the local pooling operator.

B.5 Complex unbalanced pooling

Assume a right-looking pooling kernel with size $k = 2$, stride $s = 2$, input tensor size $n = 20$, partition size $P = 6$, and no padding or dilation. In Figure B10d we illustrate that this situation yields many workers with unbalanced halos. For the first and second workers, there are no halos. The third worker has a right halo but no left halo. The 4$^{\text{th}}$ worker has 1 extra input on the left and a halo of length 2 on the right. The 5$^{\text{th}}$ worker has 2 extra inputs on the left and a halo of length 1 on the right. The final worker has no halos, but one extra input on the left. In cases with extra input data, those entries of the input tensor actually has to be removed when the input is provided to the local pooling operator.

B.6 Generalized tensor halo exchange

Here we illustrate the generalized, unbalanced halo exchange on an order-2 tensor, partitioned by a $P = 2 \times 2$ partition. While the algorithm works for tensors of arbitrary order with arbitrary partitions, an order-2 tensor is sufficient to illustrate the concept. In Figure B11a, we have partitioned the tensor into four unequal, but load-balanced subtensors. The colors will be maintained throughout subsequent figures to help illustrate data ownership. The differences in size are exaggerated for clarity. As seen in Figure B12a, where gray regions are the halo region, workers 0 and 2 require no data from workers 1 and 3, but share width 3 data with them, workers 0 and 1 require width 2 data from workers 2 and 3, workers 2 and 3 require width 4 data from workers 0 and 1, and there are interior halos only. We have chosen the vertical dimension to perform the first exchange.

Figure B12 illustrates the sequence of copy operations in the forward halo exchange algorithm. After two steps (Figures B12b and B12c), the final exchanged result is in Figure B12d. The exchange pattern is nested to minimize communication volume, as for larger, higher-rank tensors these volumes grow quickly. The gray arrows in the second exchange phase indicate that no data needs to be shared. We

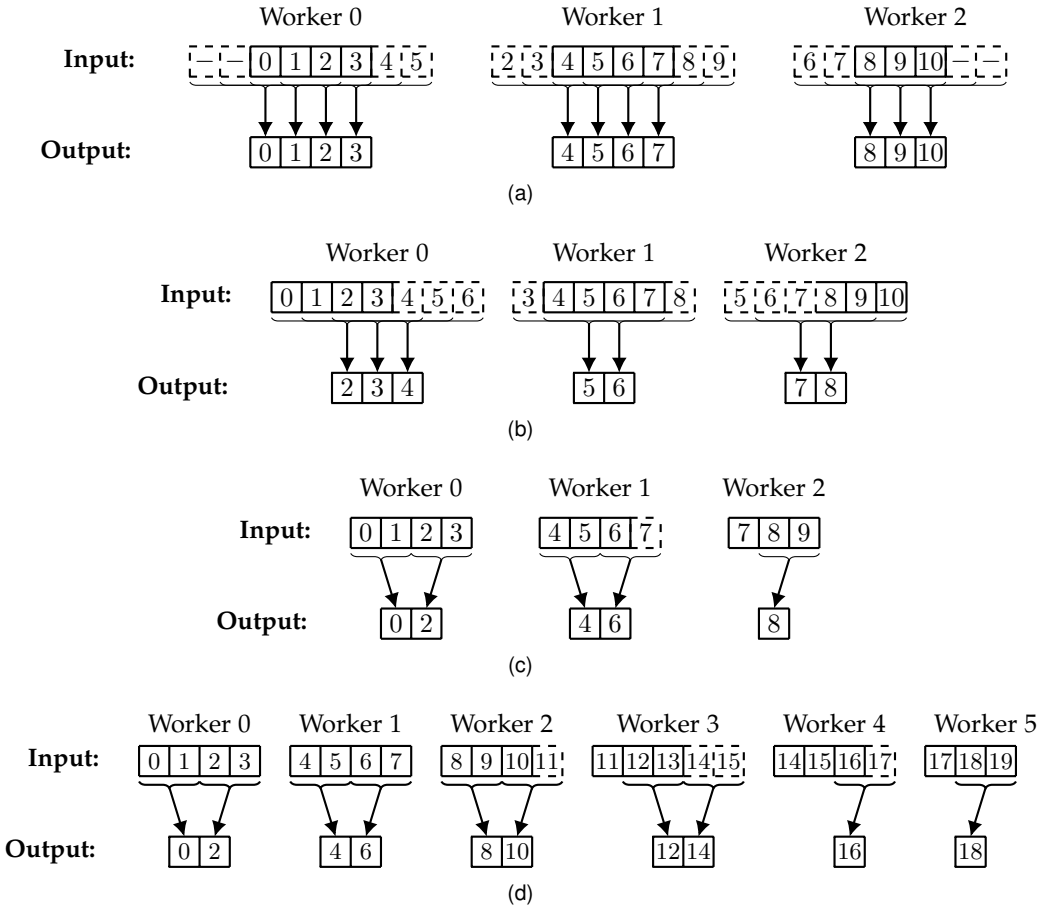

Fig. B10. Examples of possible halo patterns, such as (a) uniform halos induced by a $k = 5$ centered kernel and width 2 padding; (b) non-uniform halos induced by a $k = 5$ centered kernel and no padding; (c) unbalanced induced by a $k = 2$ right-looking kernel, with stride 2; and (d) unbalanced induced by a $k = 2$ right-looking kernel, with stride 2.

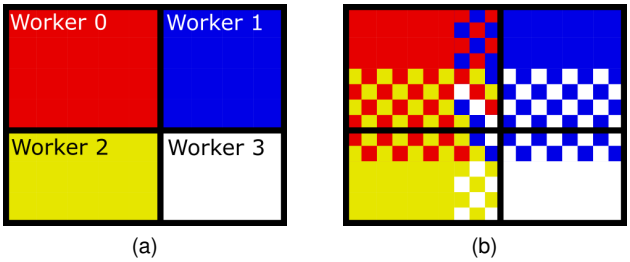

Fig. B11. Data (a) before forward halo exchange and (b) after adjoint halo exchange for $P = 2 \times 2$ partition of a order-2 tensor.

have omitted the action on the send and receive buffers, for clarity.

Figure B13 illustrates the sequence of add-clear operations in the adjoint halo exchange algorithm. Figure B13a shows the starting state, where each rank has input data starting in its halo regions. After two steps (Figures B13b and B13c), the final exchanged result is in Figure B13d. The checkerboard patterns indicate summation. The gray arrows in the adjoint of the second exchange phase indicate that no data needs to be shared.

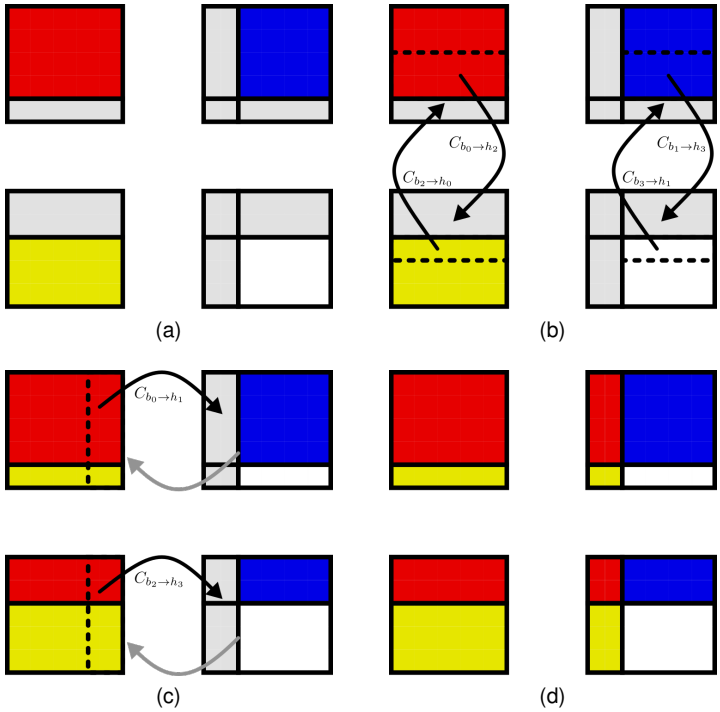

Fig. B12. Forward unbalanced halo exchange for $P = 2 \times 2$ partition of a order-2 tensor, from (a) setup, (b) and (c) directional exchanges, to (d) final result.

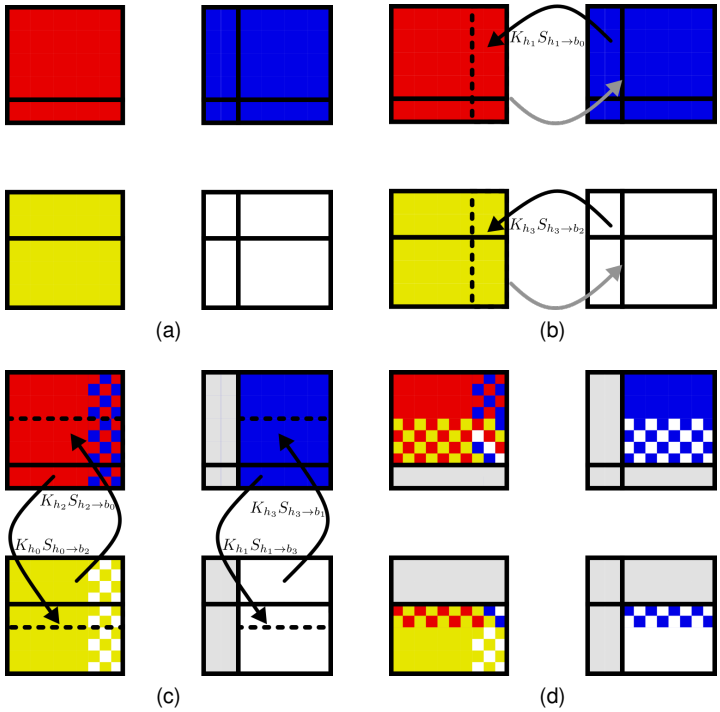

Fig. B13. Adjoint unbalanced halo exchange for $P = 2 \times 2$ partition of a order-2 tensor, from (a) setup, (b) and (c) directional exchanges, to (d) final result.