A Collective Communication Library for Distributed Tensors

Francisco Heron de Carvalho-Junior
Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil
heron@dc.ufc.br

Francisco José Lins Magalhães
Independent Researcher
Fortaleza, Brazil
franzejr@gmail.com

Abstract

Tensors, represented as multidimensional arrays, are a fundamental abstraction in scientific computing and deep learning. This paper presents NxColl, a collective communication library for distributed Nx tensors in the Elixir programming language. NxColl generalizes the semantics of collective communication to support multidimensional tensors distributed across processes arranged in a Cartesian topology, while offering a declarative interface tailored to functional programming. A case study based on the CG kernel from the NAS Parallel Benchmarks (NPB) evaluates the proposed approach.

Keywords

Tensor, Collective Communication, High-Performance Computing, Functional Programming Language, Elixir, and Nx

1 Introduction

Tensors now constitute an important programming abstraction in scientific computing and have gained greater attention due to their importance in deep learning. They are implemented using programming-language support for multidimensional arrays or highly optimized tensor computing libraries.

When tensors in a program do not fit within the memory capacity of a single computational device, they must be distributed across multiple devices using distributed-memory parallel computing techniques, such as Message Passing Interface (MPI) libraries [5, 6], and, more recently, GPU-oriented libraries such as NVIDIA's NCCL [20] and PyTorch's DTensor [12], which provide collective communication operations. However, because they are inspired by MPI, such operations assume flat, one-dimensional input tensors. Although a tensor carries meaningful structure along each of its axes, the programmer is forced to flatten it before communication and reshape it at the destination, discarding that dimensional semantics exactly where multidimensional reasoning would be most valuable.

Nx (Numerical Elixir) is the standard tensor computing library for Elixir [19, 22], a functional programming language that has gained industry attention. It supports distributing tensor computations across multiple GPUs on a single host via the EXLA backend, but lacks support for distributing tensors across processes running on different nodes in a cluster, despite Elixir's ease of creating and distributing processes across different hosts and communicating via message passing. To fill this gap and contribute to Elixir's ecosystem, we introduce NxColl, a collective communication library for Nx tensors distributed across the nodes of a distributed-memory parallel computing platform, such as a cluster or an MPP.

NxColl treats process topology, sharding, and collective communication as first-class abstractions. It provides three communication primitives: scatter, gather, and exchange. They generalize classical collective operations for multidimensional tensors distributed across Cartesian process topologies, eliminating the need to linearize tensors before communication. By simplifying collective communication interfaces and extending their semantics to multidimensional arrays, NxColl provides a more declarative programming model that is naturally suited to functional programming languages.

To evaluate NxColl, CGx has been developed. It is an Elixir implementation of CG, a kernel of the NAS Parallel Benchmark (NPB) that relies on a complex pattern of collective operations (all-reduce and all-to-all-v) across a power-of-2 number of processes organized in a bidimensional grid, and that implements the conjugate gradient method. CGx employs Nx for tensor computing. The results of an experimental evaluation on a cloud-based cluster show the simplicity and efficacy of the NxColl interface and provide insights into its performance, although it is not yet an optimal implementation.

The paper comprises an additional five sections. Section 2 provides the background necessary to understand the relevance and contributions of this paper. Section 3 presents NxColl, focusing on the interface and formalizing the semantics of the collective communication operations it offers. Section 4 presents the proof-of-concept evaluation of NxColl using CGx. Finally, Section 5 presents related work, and Section 6 concludes the paper and outlines the next steps of the research with NxColl.

2 Background

Tensors originate in the field of *multilinear algebra* [7]. As abstractions in computer programs, they are best known for the data structures that implement them: multidimensional arrays. Scalars, vectors, and matrices are tensors of orders 0, 1, and 2, respectively.

Tensors are widely used in scientific computing and deep learning to represent multilinear relationships. For instance, neural network activations are typically four-dimensional tensors, with *batch*, *channels*, *height*, and *width* axes; transformer attention operates on three-dimensional tensors [25]; and scientific simulations manipulate fields on three- or higher-dimensional grids. The shape of a tensor carries structural meaning, and operations (reductions, broadcasts, contractions, etc) are expressed in terms of its axes.

Elixir is a functional programming language that runs on the Erlang Virtual Machine (BEAM) [1, 10, 22]. From Erlang, it inherits a concurrency model based on lightweight, share-nothing processes that interact exclusively through asynchronous message passing, as well as transparent distribution, so that processes running on different nodes of a cluster communicate using the same primitives as local ones. This message-passing foundation aligns naturally with the distributed-memory parallel model targeted in this work, making Elixir an attractive host for a collective communication

library. Furthermore, its metaprogramming facilities and higher-order abstractions allow generic numerical code to be written in a way that is decoupled from low-level representation details.

Numerical computing in Elixir is provided by Nx, a library that introduces multidimensional tensors as first-class data structures and provides an API with tensor operations [19]. These operations are backend- and compiler-agnostic, so that numerical definitions can be just-in-time compiled for CPUs, GPUs, and TPUs via backends such as EXLA (built on Google's XLA) [14] and Torchx [18] (built on LibTorch), or executed via a pure-Elixir reference backend [15].

Nx is primarily designed to operate on *dense* tensors, and does not offer first-class support for sparse ones, which must be encoded by hand. Moreover, as will be discussed in Section 4, the coordinate list (COO) format proves to be more efficient than the Compressed Sparse Row (CSR) format when implemented with the operations currently available in Nx, such as indexed_add for sparse matrix-vector products [17], as opposed to sparse tensors implemented manually in native execution languages with support for linear arrays, such as C and Fortran.

When tensor processing exceeds the resource limits of a single computational device, distributed-memory parallel processing becomes necessary, where the Message Passing Interface (MPI) is the de facto standard, providing point-to-point and collective primitives over groups of processes [5, 13]. Beyond linear rankings, MPI supports *process topologies*, logical arrangements reflecting the application's communication structure. The Cartesian topology is the most commonly used, organizing processes as an n-dimensional grid or torus with explicit neighbors along each axis [8].

Process topologies are relevant because they bridge an application's data layout with the physical hardware: a two-dimensional stencil maps cleanly onto a two-dimensional grid where each process exchanges data with its four neighbors, and MPI implementations can remap ranks to minimize network hops [9, 24]. This connects directly to *distributed tensors*, i.e., tensors whose dimensions are partitioned across processes, each holding a local partition, commonly referred to as a *shard*.

Examples of distributed-memory parallel deep learning frameworks that treat distributed tensors as first-class objects with explicit sharding annotations along each axis are Mesh-TensorFlow [21], PyTorch's DTensor [12], and JAX's GSPMD [26]. The mapping is natural. When a tensor is sharded along a subset of its dimensions, those dimensions map onto a multidimensional process grid, with each grid axis corresponding to one tensor dimension. In this way, communication patterns become predictable, collectives can be restricted to subgroups along a single axis, and compilers can reason about data movement statically [21, 26].

However, still inspired by MPI, modern collective communication libraries, such as NVIDIA's NCCL [20] and DTensor, still operate over one-dimensional buffers, despite the multidimensional structure of tensors at the application level. As a result, programmers must flatten tensors into contiguous bytes before transmission and reshape them at the destination, thereby losing dimensional semantics at the communication layer.

We argue that a multidimensional collective communication interface that preserves tensor shape at the programming level should be useful for expressing collective operations directly in terms of a tensor's axes, mirroring how sharding and process topologies are already organized, rather than over opaque one-dimensional buffers. Such an interface relieves the programmer of writing repetitive and error-prone flattening and reshaping code, preserves the dimensional semantics of tensors at the communication layer, and improves expressiveness when manipulating tensors distributed across multidimensional process grids. This is precisely what we propose with NxColl, described in the next section.

3 The NxColl Communication Library

We propose NxColl, a library for collective communication over distributed tensors. It is primarily targeted at distributed-memory parallel computing platforms, such as clusters and MPPs [4, 23], but it could also be applied to multi-GPU hosts. For simplicity, in the rest of the paper, we refer to the target platforms as clusters, assuming they are built from a set of relatively independent computing nodes on which parallel processes run.

NxColl sits upon four concepts. The first two relate to distributed processes:

- *group*: a subset of the processes in an Nx program engaged in a parallel computation;
- *topology*: how processes belonging to a group are organized, assuming a Cartesian topology.

In turn, the last two concepts are related to distributed tensors, which are distributed across the processes by taking their topology into consideration:

- *sharding*: the partitioning of tensors into blocks to be distributed across the nodes of a cluster;
- *collective operations*: a set of functions for managing blocks of a distributed tensor across the nodes of a cluster.

In the following sections, we describe how NxColl addresses the concepts outlined above through an intuitive approach followed by a formal description when we deem it necessary. Concrete examples of use will be presented later, in the case study in Section 4. Also, tensor coordinates and slice boundaries follow the zero-based indexing convention adopted by Elixir and Nx. Consequently, slice intervals are represented as inclusive ranges a:b, where both a and b are zero-based coordinates.

3.1 Sharding

In distributed-memory parallel computing over tensors, it is assumed that a tensor is distributed across a set of processes mapped to the cluster's nodes. In this context, we say that tensors are partitioned into *shards*, each of which is mapped to a process.

In what follows, inspired by Elixir syntax, we use $\{a_i\ ^{i=1\ldots n}\}$ to denote the tuple $\{a_1, a_2, \ldots, a_n\}$. Let $\{n_i\ ^{i=1\ldots k}\}$ denote the shape of an arbitrary tensor T to be distributed across the nodes of a cluster, with $n_i > 1$. For that, sharding of T may be specified as a tuple $\{p_i\ ^{i=1\ldots k}\}$, where $p_i > 0 \wedge n_i \% p_i = 0$, i.e., n_i is a multiple of p_i. It represents a set of $\prod_{i=1}^{k} p_i$ shards $S_{l_1,l_2,\ldots,l_k}$, for $l_j \in \{0, \ldots, p_j - 1\}$, with shape $\{n_i/p_i\ ^{i=1\ldots k}\}$. Indeed, the elements of T are homogeneously partitioned across shards, so that the shard $S_{l_1,l_2,\ldots,l_k}$ represents the zero-based tensor slice $\left\{\left(l_j - 1\right) \times \frac{n_j}{p_j} + 1 : l_j \times \frac{n_j}{p_j}\ ^{j=1\ldots k}\right\}$ in the global view of T.

We also define *variant sharding* for non-homogeneous tensor partitioning. It will be represented as $\{\{q_{ij}\ ^{i=1\ldots n_j}\}\ ^{j=1\ldots k}\}$, for a

tensor with shape $\left\{\sum_{i=1}^{n_j} q_{ij}\ ^{j=1\ldots k}\right\}$. So, this is a partitioning of T where each shard $S_{l_1,l_2,\ldots,l_k}$ represents the following T's slice:

$$\left\{\left(\sum_{i=1}^{l_j-1} q_{ij}\right)+1 : \sum_{i=1}^{l_j} q_{ij}\ ^{j=1\ldots k}\right\}$$

.

For example, let T be a three-dimensional multivariate tensor with shape $\{100, 200, 50\}$ ($k = 3$, $n_1 = 100$, $n_2 = 200$, $n_3 = 50$). A valid sharding for such tensor would be $\{2, 4, 1\}$ ($p_1 = 2$, $p_2 = 4$, $p_3 = 1$), leading to 8 shards to be distributed across a set of processes. For example, shard $S_{1,2,0}$ represent the T's slice $\{50{:}99, 100{:}149, 0{:}49\}$. In turn, $\{\{30, 40, 30\}, \{60, 50, 90\}, \{25, 25\}\}$ is an example of variant sharding ($q_{11} = 30$, $q_{21} = 40$, $q_{31} = 30$, $q_{12} = 60$, $q_{22} = 50$, $q_{32} = 90$, $q_{13} = 25$, and $q_{23} = 75$), with 18 shards, where shard $S_{2,2,1}$, for example, representing the slice $\{70 : 99, 110 : 199, 25 : 49\}$.

3.2 Topology

Processes are organized in an n-d grid, i.e., a grid with d dimensions and n_i processes per dimension, for $i \in \{0, \ldots, d-1\}$. Such grids are often referred to as Cartesian topologies, and linear arrays, two-dimensional and three-dimensional meshes/tori, and hypercubes are the most commonly used in parallel programs.

Considering process topologies in the development of parallel programs is motivated by the interconnection properties of the cluster nodes, aiming to optimize communication performance, or by the communication pattern between their processes, with the objective of improving program structuring and promoting an efficient mapping of processes onto cluster nodes by respecting the topology of their interconnecting network, thereby minimizing communication and synchronization overheads.

We assume that tensor-parallel programs distribute tensor shards across processes according to the process topology. So, let $\{a_i\ ^{i=1\ldots d}\}$, where $a_i > 1$, be the shape of an arbitrary process topology for $\prod_{i=1}^{d} a_i$ processes and $\{n_i\ ^{i=1\ldots k}\}$, where $k \geq d$, be the shape of a tensor T. A distribution mapping of T across such a process topology is defined by a sharding tuple $\{p_i\ ^{i=1\ldots d}\}$ and an associated index tuple $\{q_i\ ^{i=1\ldots d}\}$, where $q_i \in \{1 \ldots k\}$ and $i \neq j \implies q_i \neq q_j$. In fact, the q_i's are indices across tensor axes, and the p_i's specify how the tensor is sharded at the q_i axis. For a homogeneous distribution, $n_i \% p_i = 0$, i.e., n_i must be a multiple of p_i. A non-homogeneous distribution may be derived by applying variant sharding.

For example, let T be a tensor with a $\{400, 100, 400\}$ shape ($k = 3$, $n_1 = 400$, $n_2 = 100$, and $n_3 = 400$) to be distributed across a set of 16 processes in a $\{4, 4\}$ topology ($d = 2$, $a_1 = 4$, and $a_2 = 4$). A $\{16, 8\}$ sharding is valid to be applied ($p_1 = 16$, and $p_2 = 8$) over tensor axes specified by $\{1, 3\}$ ($q_1 = 1$ and $q_2 = 3$), leading to 128 shards with shape $\{25, 100, 50\}$. In such an example, $k_1 = 4$ and $k_3 = 2$. So, each process will receive 8 shards, forming a tensor block of shape $\{100, 100, 100\}$ by concatenating them.

The order in which the shards within a process are concatenated is defined by a selected application policy, applied separately to each sharding axis. For example, shards $S_{i,0,0}\ ^{i=0\ldots 15}$ may be distributed across the processes $P_{j,0}\ ^{i=0\ldots 3}$ in such a way that $P_{0,0}$ receives either $\{S_{0,0,0}, S_{1,0,0}, S_{2,0,0}, S_{3,0,0}\}$ (block/contiguous policy) or $\{S_{0,0,0}, S_{4,0,0}, S_{8,0,0}, S_{12,0,0}\}$ shards (round-robin policy).

The above definitions for mapping tensors onto process topologies using sharding may be analogously applied to variant sharding.

3.3 Groups

In a parallel computation over distributed tensors, a group of processes performs operations on tensor shards mapped to them. NxColl allows defining process groups organized in a Cartesian topology using two operations: `new_group` and `group_split`. Both must be applied to an existing group, through the `group` keyword parameter, forming a new group. When the computation starts, there is a single group of processes, following a linear topology, called the *world group*, a nomenclature inspired by MPI. It is the default value of the `group` parameter (:world).

Listing 1: The Group structure

```
@opaque group :: %Group{
          pids: [pid],
          topology: tuple(),
          coords: Tuple,
          coord: [integer],
          rank: integer
        }
```

NxColl defines a group as the opaque structure in Listing 1. The `pids` list includes the process identifier (pid) for each process in the group. In turn, `topology` is a tuple that specifies the topology's shape, i.e., the number of processes in each dimension, whose product equals the length of `pids`. The list `coords` specifies the coordinate of each process in `pids`, while the tuple `coord` specifies the coordinate of the current process, whose rank, i.e., its position in list `pids`, is defined by `rank`.

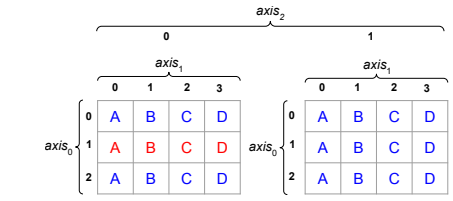
The `new_group` operation receives a mandatory argument with the pids of a set of processes to be included in the new group, and two keyword arguments (optional), named `group` and `topology`. While the first argument is the initial group, to which the pids must belong (default to `:world`), the latter specifies the topology in which they will be organized (default to $\{n\}$, where n is the number of pids, i.e., a linear topology). For example, the code below creates a group for processes in the list `pids`, belonging to the world group (default), organized in a $nprows \times npcols$ two-dimensional grid:

```
group_solve = Group.new_group(pids, topology: {nprows, npcols})
```

The `group_split` operation splits a group according to its topology. For that, it takes two arguments: a group and a list of integers representing dimensions. Processes in the initial group that belong to the same axes across the given dimensions are included in the same group, which is returned by the operation. For example, the following calls to group_split create `nprows` + `npcols` groups, each referenced by a process using the variables `group_row` and `group_col`. The first group includes all the processes in the initial group that are in the same row as the current process, while the last group includes all the processes in the current process's column.

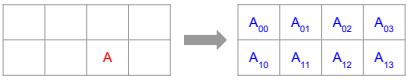
```
group_row = Group.group_split(group_solve, [0])
group_col = Group.group_split(group_solve, [1])
```

Future versions of NxColl may incorporate additional operations for creating groups.

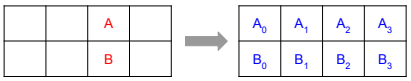


(a) `scatter(t, axis = [1], root = {1,0})` {3, 4, 2}

(b) `scatter(t) = broadcast(v)` {8}

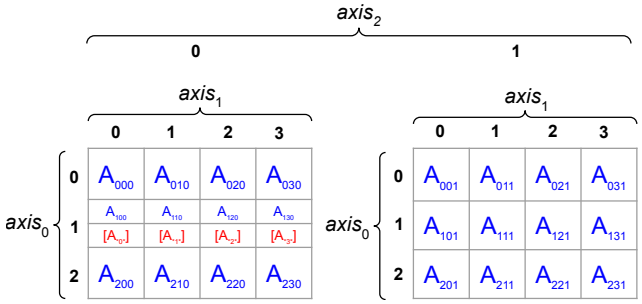


(c) `scatter(t, sharding = {2,4}, root = {1,2})` {2, 4}

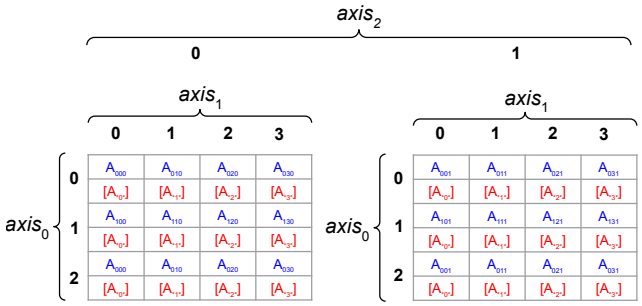


(d) `scatter(t, axis = [1], sharding = {4}, root = {2})` {2, 4}

Figure 1: Scatter examples (for different topologies)



(a) `gather(t, axis = [1], root = {1,0})`



(b) `gather(t, axis = [1], root = :all)`

Figure 2: Gather examples

3.4 Collective Operations

NxColl offers three fundamental collective communication operations over tensors: `gather`, `scatter`, and `exchange`. From them, other common, more specialized operations may be defined, such as `broadcast`, `reduce`, `allreduce`, `alltoall`, and so on, offered by existing libraries, such as MPI and PyTorch's DTensor. They may receive a group through the keyword argument `group`. In fact, they generalize the basic collective operations described in the literature from a simple linear topology to general Cartesian topologies. i.e., across multiple dimensions, thereby improving expressiveness for handling higher-order tensors and avoiding the need to linearize them to apply collective operations. The semantics of the operations and their implementation are described in the following sections.

3.4.1 *Scatter, with Support to Broadcast.*

With scatter, a tensor distributed across a slice of the process topology is copied onto each complementary slice by fixing a set of coordinates (*root*).

Formally, let $\{a_i \ ^{i=1\ldots d}\}$, where $a_i > 1$, be the shape of an arbitrary process topology. Also, consider $\{t_{s_i} \ ^{i=1\ldots k}\}$, where $k < d$, $t_{s_i} \in \{1, \ldots, a_{s_i}\}$ and $\{s_i \ ^{i=1\ldots k}\} \subset \{1 \ldots d\}$, a specification of fixed coordinates defining the slice of the process topology along which an arbitrary tensor T is distributed. T will be copied to each slice of the process topology belonging to the set

$$\prod_{i=1}^{d} \begin{cases} \{1, \cdots, a_i\} & \text{if } i \in \{s_j \ ^{j=1\ldots k}\} \quad \textit{fixed index} \\ \{1{:}a_i\} & \textit{otherwise} \quad \textit{free index} \end{cases}$$

, where $\prod$ represents the Cartesian product of tuples.

Figure 1(a) illustrates scatter for a $3 \times 4 \times 2$ topology ($d = 3$, $a_1 = 3$, $a_2 = 4$, and $a_3 = 2$), where the input tensor is distributed across the processes at the $\{1, 1{:}4, 0\}$ coordinates ($s_1 = 1$, $s_2 = 3$, $t_1 = 1$, and $t_3 = 0$). The scatter operation copies the input tensor to each $\{i, 1{:}4, k\}$ $^{i=1\ldots3 \ k=1\ldots2}$ face (in blue). For that, `axis` must be set to `[1]`, i.e., $axis_1$ as the free axis. In turn, `root` must be set to $\{1, 0\}$, which are the coordinates for the fixed axis ($axis_0$ and $axis_2$).

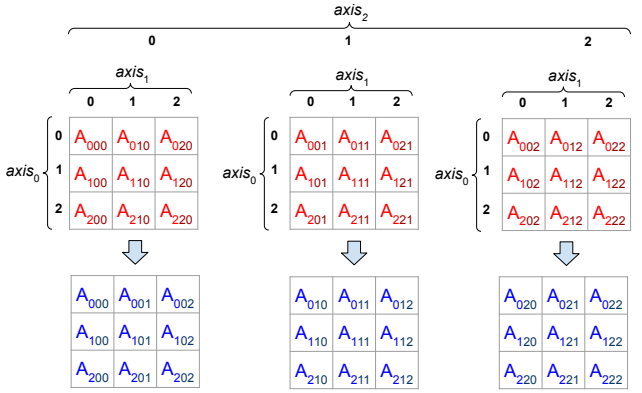
In turn, Figure 1(b) specifies how a simple linear broadcast, like implemented by MPI and PyTorch's DTensor, may be specified using `scatter`. In fact, it is the default case, being only necessary to rely on the default arguments for the keyword parameters ([] for `axis`, i.e., the empty list, and {0} for `axis`) and assume a linear process topology, i.e., $\{n\}$, where n is the number of processes.

With sharding, it is possible to distribute slices of the input tensor across processes along the corresponding axis, rather than copying entire slices. This is illustrated in figures 1(c) and 1(d). In the former, the `sharding` parameter is configured to perform a two-dimensional sharding ($\{2, 4\}$), so that each slice $\mathtt{A}_{i,j}$ of the input tensor `A`, located at the process at the coordinate $\{1, 2\}$ is mapped to the process at the $\{i, j\}$ coordinate, for $i \in \{1, 2\}$ and $j \in \{1, 2, 3, 4\}$. In turn, Figure 1(d) exemplifies sharding with a tensor distributed across a linear slice in the process topology, specified by setting `axis` to `[1]` and `root` to $\{2\}$. In such a case, `sharding` may be configured to shard the input tensor slices in 4 slices, across a single dimension ($\{4\}$).

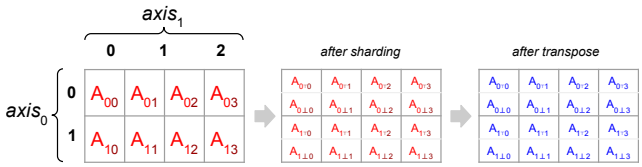
3.4.2 *Gather, with Support to Reduce, All-Reduce, and All-Gather.*

Gather is the dual operation of scatter. So, it can be interpreted as the reverse of the communication order of scatter.

Figure 2(a) illustrates the gather operation, based on the example of Figure 1(a) to illustrate the duality with scatter. The parameters `axis` and `root` specify the slice of process topology where the



(a) **exchange**(t, **transpose** = {0,2,1})



(b) **exchange**(t, **sharding** = {2})

Figure 3: Exchange examples

output tensor will be placed. The input tensor is distributed across all processes, so that it has a slice at each one of them (A_{ijk}). The target processes collect a set of tensor slices from the other processes and concatenate them, thereby redistributing the input tensor from all processes to the target processes.

Figure 2(b) illustrate how gather can be used to define allgather operations by setting `root` to `:all`. The output tensor is copied to all the corresponding process slices, replicating it. In fact, it corresponds to a gather followed by a broadcast.

It is also possible to specify a reduction function for the target processes to apply to the received slices via the `reduce` parameter. For example, `reduce = &Nx.add/2` may be applied in the examples of Figure 2 to sum the received tensor slices. This is how the reduce and allreduce operations are implemented.

Finally, using the `sharding` parameter, sharding can be applied to reorder the input tensor slices in the output tensor.

3.4.3 *Exchange, a Generalization of All-to-All-v.*

The exchange operation is rather complex. The simplest scenario is a square n-dimensional process topology over which the input tensor is distributed. It performs a transposition of the tensor's slices in the process topology according to a permutation of its dimensions, specified by the `transpose` parameter. This is exemplified in Figure 3(a), for a $\{3, 3, 3\}$ process topology over the second and third dimensions (i.e., `transpose` = $\{0, 2, 1\}$).

For non-square process topologies, sharding may be applied to slice the tensor within processes, obtaining a square view of the tensor slices across the process topology. For instance, in Figure 3(b), a $\{2, 1\}$ sharding is applied to the input tensor blocks in the first dimension to define a $\{4, 4\}$ square view of the tensor slices across the $\{2, 4\}$ grid of processes. For example, in the process at the $\{0, 0\}$ coordinate, the tensor shard A_{ij} is sliced in two blocks: $A_{i\top j}$ and $A_{i\perp j}$. With such an adjustment, transposition may be performed.

With sharding, exchange transposition is performed according to the shape in which the resulting slices of the distributed tensor are organized. So, it can be applied to more dimensions than the process topology. For instance, in the example of Figure 3(b), for a three-dimensional input tensor, a sharding of $\{2, 1, 2\}$ could be applied, so that a three-dimensional transposition could be specified through `transpose`, e.g., $\{2, 0, 1\}$, reorganizing the blocks across the process topology.

Let us specify the semantics of exchange in formal terms. Let $\{a_i \ ^{i=1\ldots d}\}$, with $a_i > 1$, denote the shape of an arbitrary process topology over which an n-dimensional multivariate tensor T is distributed, where $n \geq d$. T is distributed across the process topology through the dimensions in the list $E = [e_i \ ^{i=1\ldots d} \mid e_i \in [1, \ldots, n] \wedge \forall e_k, e_j . k \neq j \implies e_k \neq e_j]$.

Also, consider a sharding of $\{s_i \ ^{i=1\ldots n}\}$ over the tensor blocks, such that $m = s_1 \times r_1 = s_2 \times r_2 = \cdots == s_n \times r_n$, where

$$r_i = \begin{cases} a_i, & i \in E \\ 1, & otherwise \end{cases}$$

. Such a sharding configuration defines an m-dimensional square topology of slices of tensor blocks across the process topology. Now, let $\{q_i \ ^{i=1\ldots m}\}$ a permutation of $\{1, \ldots, m\}$. In such a transposition, the tensor block slice $T_{i_1 i_2 \ldots i_n}$ and $T_{q_{i_1} q_{i_2} \ldots q_{i_n}}$ will exchange their position along the process topology. If they are placed in different processes, communication between such processes is performed.

3.5 Implementation Status

In this initial version, NxColl remains a proof-of-concept prototype. It is not yet concerned with providing the most efficient possible implementations of the collective operations. Indeed, scatter, gather, and exchange are implemented on top of Elixir's basic send and recv primitives, with performance implications. In addition to the lack of support for RDMA communication, as with MPI libraries, all communication processing goes through the BEAM machine, which was not originally developed with HPC concerns in mind. In addition, it is not possible to use it in the context of defn functions, where the backend compiler may generate more optimized code[1]

All of these issues may be addressed in future versions of NxColl in order to make it ready for production use in the Elixir ecosystem. Our initial focus is validating the interface and the semantics of its operations, which are the main contribution of this paper.

4 Case Study

The NAS Parallel Benchmarks (NPB) is a benchmark suite from the 1990s for evaluating the performance of parallel computers in CFD applications [2]. It comprises five kernels (EP, IS, CG, MG, and FT) and three simulated applications (SP, BT, and LU), implemented in C or FORTRAN. It has also gained widespread use for

[1] In the initial exploratory tests, the performance of the sequential version of CGx (without NxColl), with the CG loop encapsulated using numerical functions (defn) [16] compiled with the EXLA CPU backend, was slightly worse than the version with conventional functions (def), despite we have followed coding recommendations for enabling aggressive optimizations. The EXLA CPU backend probably already performs aggressive tensor optimizations by default, for def functions.

$x = [1, 1, \ldots, 1]^T$
(*starting timing here*)
DO *it* = 1, niter
  run the conjgrad loop (right) to solve $Az = x$
  $\zeta = \lambda + 1/(x^T z)$
  Print *it*, $||r||$, and ζ
  $x = z/||z||$
ENDDO
(*stop timing here*)

(a) The main loop

$r = x$
$\rho = r^T r$
DO i = 1, 25
  $q = Ap$
  $\alpha = \rho/(p^T q)$
  $z = z + \alpha p$
  $\rho_0 = \rho$
  $r = r - \alpha q$
  $\rho = r^T r$
  $\beta = \rho/\rho_0$
  $p = r + \beta p$
ENDDO
$||r|| = ||x - Az||$

(b) The conjgrad loop

Figure 4: The NPB/CG loop [2]

assessing the performance of programming languages in scientific computing, particularly their parallelism support, leading to new official and unofficial implementations in other languages over the years, along with new kernels (UA, BT-IO, DC, DT, ED, HC, VP, and MB) and larger instance classes (E and F) to meet the requirements of emerging parallel computing platforms and programming interfaces.

CG is an NPB kernel that implements the conjugate gradient method to solve the system $Az = x$. It has been designed to exploit irregular memory access and communication in benchmarking parallel computers and languages. It is an iterative code that performs linear algebra operations (scalar-vector, vector-vector, and matrix-vector products) over a set of input and auxiliary vectors (x, z, p, q, and r) and a sparse matrix (a) in Compressed Sparse Row (CSR) format, as depicted in Figure 4. In the parallel version, a 2D block distribution is applied to distribute a across a grid formed by a power-of-two number of processes in a 2D topology. Vectors are replicated across the rows of this process grid, so that they are homogeneously distributed across processes in the same row.

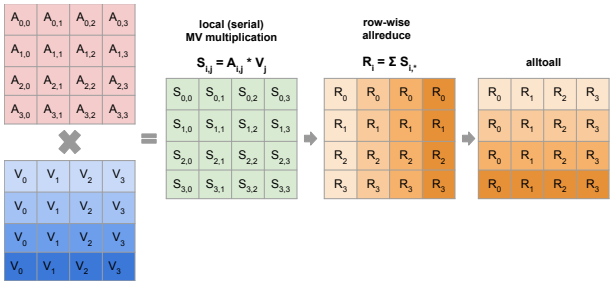
To evaluate NxColl, we have developed CGx, an Elixir implementation of CG using Nx. In contrast to the original FORTRAN implementation, CGx uses the COOrdinate List Format (COO) to represent the input sparse matrix a, because, as pointed out in Section 2, sparse matrix-vector multiplication (spMV) may be implemented more efficiently in Nx with matrices in the COO format, instead of CSR. In fact, Elixir's abstraction mechanisms enabled the implementation of the CG loop independently of how a is stored.

In the parallel version of CG, communication between processes occurs through vector-vector and matrix-vector operations, highlighted in blue and red, respectively, in Figure 4. Most of the computation time is spent on matrix-vector multiplication.

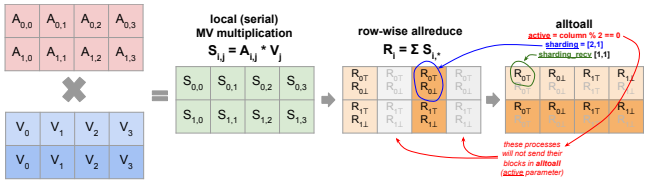
Listing 2: The vv_multiply function

```
defmodule VVMulParallel do
  def vv_multiply(x, y, opts \\ []) do
    group_row = opts[:group_row] || nil
    r = Nx.dot(x, y)
    if group_row do
      Collective.allreduce(r, &Nx.add/2, group: group_row)
    else
      r
    end
  end
end
```

Listings 2 and 3 present `vv_multiply` and `mv_multiply`, the CGx functions that implement vector-vector and matrix-vector multiplication, respectively. Both operate over groups of processes



(a) 2^n processes, where n is even (e.g., 16 processes, 4×4 topology).



(b) 2^n processes, where n is odd (e.g., 8 processes, 2×4 topology).

Figure 5: Parallel matrix-vector multiplication ($R = A \times V$)

in the same row of the grid (`group_row` variable) by applying allreduce over them, while `mv_multiply` also applies an alltoall over the overall group of processes (group_solve variable).

Listing 3: The mv_multiply function

```
defmodule MVMulParallel do
  def mv_multiply(a, x, t0, opts \\ []) do
    group_row = opts[:group_row]
    group_solve = opts[:group_solve]

    r = MVMulSerial.mv_multiply(a, x, t0)
          |> Collective.allreduce(&Nx.add/2, group: group_row)

    {rows, cols} = group_solve.topology
    {_row, col} = group_solve.coord

    if rows == cols do
      Collective.alltoall(r, group: group_solve)
    else
      Collective.alltoall(r, group: group_solve,
                             active: rem(col, 2) == 0,
                             sharding: [2, 1],
                             sharding_recv: [1, 1])
    end
  end
end
```

The way message-passing communication underlying all-reduce and all-to-all patterns is implemented significantly affects the performance of both operations, especially `mv_multiply`. So, using Figure 5, let us examine the collective interaction pattern between processes in matrix-vector multiplication. There are two scenarios, in figures 5(a) and 5(b), respectively: when the number of processes is a perfect square (2^n, where n is even) and when it is not (n is odd). So, let r and c be the number of rows and columns of the process topology, respectively. In the former case, $c = r$, while $c = 2 * r$ in the latter.

One can observe in Figure 5(a) that all-to-all is necessary because, after the local matrix-vector multiplication followed by row-wise reduction (all-reduce), the resulting vector (R) is replicated across the columns rather than the rows of the process topology, distributed

Table 1: Experimental results

Total number of cores: / Nodes and cores per node:			1 1×1	2 2×1	4 4×1	8 4×2	16 4×4
EXECUTION TIME (s)							
Elixir (CGx)	sparse (C)	COO	646.5	459.3	248.5	364.5	163.2
	dense (A)	-	38.8	22.3	12.1	10.7	6.9
FORTRAN	sparse (C)	CSR	85.8	40.9	22.0	12.6	7.8
		COO	186.6	76.6	39.8	18.8	11.1
	dense (A)	-	70.9	35.9	18.3	9.5	5.0
SPEEDUP							
Elixir (CGx)	sparse (C)	COO	1.0	1.4	2.6	1.8	4.0
	dense (A)	-	1.0	1.7	3.2	3.6	5.7
FORTRAN	sparse (C)	CSR	1.0	2.4	4.7	9.9	16.8
		COO	1.0	2.1	3.9	6.8	10.9
	dense (A)	-	1.0	2.0	3.9	7.5	14.1
EFFICIENCY (%)							
Elixir (CGx)	sparse (C)	COO	100%	70%	65%	22%	25%
	dense (A)	-	100%	87%	80%	46%	35%
FORTRAN	sparse (C)	CSR	100%	122%	117%	124%	105%
		COO	100%	105%	97%	85%	68%
	dense (A)	-	100%	99%	97%	94%	88%

across processes in the same column rather than across rows. So, all-to-all redistributes blocks of the resulting vector across processes in the grid to satisfy the row-wise vector distribution requirement. However, for the special case where the number of processes is not a perfect square, depicted by Figure 5(b), it is also necessary to shard R before all-to-all to align vector blocks for performing a $4x4$ transposition. So, in Figure 5, $R_{i,\top}$ and $R_{i,\perp}$ are the upper and lower halves of the R_i block of R. Also, only processes in even columns will send their blocks to avoid duplicating vector blocks in the columns after the all-to-all.

In MPI, where collective operations operate on linear *buffers*, such a complex all-to-all communication pattern is implemented using `MPI_Alltoallv`, the variant of `MPI_Alltoall` that allows control over whether and how many data items each process contributes in the collective operation. However, `MPI_Alltoall` cannot be applied even to the simplest perfect-square scenario in Figure 5(a), because the collective operations of MPI are applied to linear buffers, making it necessary to linearize a multidimensional tensor to use it in a `MPI_Alltoall` invocation. In fact, originally, CG implements both all-reduce and all-to-all-v collective communication patterns through a sequence of asynchronous point-to-point communication (`MPI_Isend` and `MPI_Irecv`) rather than using `MPI_Allreduce` and `MPI_Alltoallv`. In contrast, as shown in listings 2 and 3, CGx employs the allreduce and alltoall operations of NxColl directly, generalized for multidimensional tensors through the gather and exchange primitive operations, as explained in Section 3, leading to a more natural solution. Indeed, to implement the nuances of the more complex all-to-all pattern in Figure 5(b), the keyword parameters `active`, `sharding`, and `sharding_recv` are applied in the alltoall invocation.

4.1 Performance Evaluation

We conducted an experimental proof-of-concept evaluation of NxColl using CGx, with the objective of demonstrating the efficacy and correctness of the collective communication operations over tensors and gathering insights into the performance of their implementation, despite its trivial implementation.

4.1.1 *Testbed Environment.*

The experiments have been performed on a cloud-based cluster comprising four nodes, instantiated using CloudClusters.jl [3]. Each node is a c3-standard-8 virtual machine instance on the us-central1-a zone of Google Cloud[2], offering four physical cores of an Intel(R) Xeon(R) Platinum 8481C CPU @ 2.70GHz (8 vCPUs), 32GB of available memory, and 23 Gbps of network bandwidth.

Regarding the software environment, the operating system on each node was Debian 12.2.0-14+deb12u1 (bookworm). Version 0.11 of both Nx and EXLA over Elixir 1.19.5-otp-8 has been employed, and MPICH 5.0.1 is the MPI implementation used in the experiments. Finally, GNU FORTRAN 12.2.0 has been used to compile FORTRAN code.

4.1.2 *Methodology.*

The experiment evaluates the performance of two versions of CGx, which assume the matrix A is sparse and dense, respectively, across 5 execution environment configurations, referred as 1×1, 2×1, 4×1, 4×2, and 4×4, where N×P denotes N nodes with P cores per node. So, N×P represents the number of processing elements of each configuration (1, 2, 4, 8, and 16, respectively), each corresponding to a single processor core. Each process was pinned to one CPU core using taskset -c to ensure a fair comparison with the sequential MPI processes.

The problem classes used in the experiments are limited by the memory available on the cluster nodes. Indeed, the largest problem class supported by the cluster was C for the sparse versions and A for the dense versions, as used in the experiment.

As a baseline for evaluating CGx, we use three versions of the FORTRAN implementation of CG: two assuming A is sparse in CSR and COO formats, respectively, and one assuming A is dense. Although the original version uses CSR, due to its better performance in native languages with multidimensional array support, such as FORTRAN, COO is much more efficient than CSR when implemented in Elixir via the currently available tensor operations of Nx. Thus, the COO-based FORTRAN version has been implemented for a fairer comparison between Elixir and FORTRAN sparse implementations of CG, yet another decision to minimize evaluation bias.

Table 1 summarizes the experimental configurations and their corresponding results. Each experiment was executed twice, and the average execution time of the two runs was used as the performance metric. The variation between runs was low, ranging from 0.01% to 7.55%, with a mean of 1.6% and a standard deviation of 0.85%. This low variability is expected because the experiments were conducted on a dedicated cluster, and the CG benchmark performs many identical iterations (15 for class A and 75 for class C), producing highly stable execution times. Consequently, the benchmark's inherent

[2]https://gcloud-compute.com/c3-standard-8.html

execution stability makes two runs per configuration sufficient to obtain representative performance measurements.

4.1.3 Results and Discussion.

Regarding the computational performance of CGx, it is highly competitive for dense matrices, even surpassing FORTRAN for scenarios with fewer processing elements (1, 2, and 4). This can be explained by the execution time dominated by the Nx.dot vector-vector and matrix-vector operations, which are implemented at a low level using state-of-the-art linear algebra algorithms by the XLA compiler underlying the EXLA backend, competing against manual implementations in the FORTRAN version.

However, CGx shows very poor computational performance on sparse matrices, as expected, since Nx does not provide specific support for sparse tensor computations. Indeed, the operations it supports do not enable efficient implementation of the CSR format, forcing us to use the COO format, whose sequential performance is worse for CG mainly due to worse cache behavior, as confirmed in Table 1, using measures for the CSR and COO FORTRAN versions.

In turn, regarding communication performance, which is the most relevant to be analyzed in this paper, the low speedup and efficiency metrics for CGx, especially with 8 or more processing elements, confirm the poor performance we expected for the current implementation of NxColl, as pointed out in Section 3.5. The worst overhead for unsquared process counts (2 and 8), which require additional sharding and communication steps, and the tendency for Nx's speedup and efficiency to worsen relative to FORTRAN as the number of processing elements increases reinforce this observation.

Despite these observations, the competitive performance of CGx up to 4 processing elements, mainly for dense matrix computation, where Nx competes with state-of-the-art tensor libraries, is a promising result, motivating us to advance research on how to develop an efficient state-of-the-art implementation of NxColl.

5 Related Works

NxColl lies at the intersection of two research areas: collective communication in message-passing interfaces and distributed (sharded) tensors in deep learning frameworks.

MPI is the reference model for collective communication, complementing point-to-point primitives with broadcast, scatter, gather, all-gather, reduce, scan, all-reduce, and all-to-all operations over process groups and communicators [5, 6]. To capture communication structure, MPI introduced *process topologies*, particularly Cartesian grids and tori, enabling runtimes to map ranks onto network topologies and reduce communication costs [8, 24]. Several works exploit this information to optimize collectives. Hoefler and Snir proposed generic topology-mapping strategies for large-scale architectures [9], while Karonis et al. introduced a multilevel topology-aware communication scheme for MPICH-G2 that minimizes traffic over slow network links [11]. NVIDIA's NCCL provides GPU-optimized collectives [20], and PyTorch's DTensor exposes an MPI-like collective interface over tensors [12].

All these libraries operate on linear memory buffers, requiring multidimensional tensors to be flattened before communication and reshaped afterward. Moreover, their notion of topology is primarily *physical*, reflecting the interconnection network rather than tensor organization. In contrast, NxColl treats Cartesian process topologies as first-class abstractions aligned with tensor axes, enabling shape-preserving collective operations while remaining compatible with topology-aware communication strategies [9, 11].

A second line of work treats distributed tensors as first-class objects with per-axis sharding. Mesh-TensorFlow distributes tensors over a multidimensional device mesh [21], GSPMD propagates sharding annotations through compiler analysis [26], and PyTorch's DTensor brings the same model to eager execution [12]. In all three systems, changes in tensor sharding rely on MPI-style collectives whose communication costs are critical to performance.

Zhang et al.'s HiDup reduces these costs through two complementary techniques [27]: a duplex execution model that overlaps communication and computation, and a dynamic-programming search for overlap-aware sharding strategies. Unlike NxColl, however, HiDup intentionally shards only one tensor dimension at a time because GPU clusters are typically not organized as multidimensional meshes. In contrast, NxColl natively models Cartesian process topologies and multidimensional tensor sharding.

6 Conclusions and Lines for Further Works

In this paper, we proposed and evaluated a new collective communication interface for distributed tensors to advance distributed-memory parallel computing in tensor computing libraries. It is implemented in the NxColl library for Nx-based Elixir programs and represents our first contribution toward enabling parallel computing on clusters and MPPs in the Nx ecosystem.

Unlike existing MPI-inspired tensor libraries, NxColl abstracts away communication buffers through just three primitive operations (scatter, gather, and exchange), which generalize classical collectives to multidimensional tensors distributed over Cartesian process topologies via sharding. This eliminates the need to linearize tensors for communication, resulting in a more declarative interface better suited to functional programming.

The efficacy and abstraction capabilities of NxColl were demonstrated through CGx, an Elixir implementation of the NPB CG kernel. Although the current prototype was not designed for high communication performance, the evaluation produced promising results and valuable guidance for future low-level optimizations.

The evaluation also quantified the current limitations of Nx for sparse tensors while showing that its dense-tensor performance is highly competitive with Fortran. These findings motivate future research on extending Nx with efficient sparse-tensor support.

As immediate future work, we plan to develop efficient implementations of collective communication primitives and explore low-level communication mechanisms such as RDMA. Another important direction is the integration of NxColl with defn numerical functions, allowing collective operations to be incorporated into compiled computation graphs and optimized by the underlying compiler infrastructure. Together, these advances are expected to significantly improve both the performance and the usability of NxColl as a communication layer for distributed tensor applications.

Artifact Availability

The NxColl and CGx projects are hosted in their respective repositories under the ElixirHPCLab GitHub organization [3].

References

[1] J. Armstrong. 2007. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 6–1–6–26. doi:10.1145/1238844.1238850

[2] D. H. Bailey and et al. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[3] F. H. de Carvalho Junior and J. M. de Alencar. 2024. Cloud-based parallel computing across multiple clusters in Julia. In *XXVIII Brazilian Symposium on Programming Languages* (Curitiba, Brazil). SBC, Porto Alegre, RS, Brasil, 44–52. doi:10.5753/sblp.2024.3470

[4] J. Dongarra. 2004. Trends in High Performance Computing. *The Computer Journal* 47, 4 (2004), 399–403.

[5] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. 1995. *An Introduction to the MPI Standard.* Technical Report CS-95-274. University of Tennessee. http://www.netlib.org/tennessee/ut-cs-95-274.ps.

[6] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. 1996. A Message Passing Standard for MPP and Workstation. *Communications of ACM* 39, 7 (1996), 84–90.

[7] P. Grinfeld. 2013. *Introduction to Tensor Analysis and the Calculus of Moving Surfaces.* Springer. doi:10.1007/978-1-4614-7867-6

[8] Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, and Jesper Larsson Träff. 2011. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience* 23, 4 (2011), 293–310. doi:10.1002/cpe.1643

[9] Torsten Hoefler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–84. doi:10.1145/1995896.1995909

[10] S. Jurić. 2024. *Elixir in Action* (3 ed.). Manning Publications, Shelter Island, NY. https://www.oreilly.com/library/view/elixir-in-action/9781633438514/

[11] Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William Gropp, Ewing Lusk, and Sébastien Lacour. 2002. A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids. *CoRR* cs.DC/0206038 (2002). arXiv:cs/0206038 [cs.DC] https://arxiv.org/abs/cs/0206038

[12] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2025. TorchTitan: One-stop PyTorch Native Solution for Production Ready LLM Pre-training. In *The Thirteenth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=SFN6Wm7YBI

[13] MPI Forum. 2026. MPI Forum. https://www.mpi-forum.org/. Official website of the Message Passing Interface (MPI) Forum, accessed 2026-06-02.

[14] Numerical Elixir (Nx). 2026. EXLA. https://hex.pm/packages/exla. Hex package for EXLA (Elixir bindings for XLA), accessed 2026-06-02.

[15] Numerical Elixir (Nx). 2026. Nx.BinaryBackend. https://nx.hexdocs.pm/Nx.BinaryBackend.html. HexDocs documentation for Nx.BinaryBackend, accessed 2026-06-02.

[16] Numerical Elixir (Nx). 2026. Nx.Defn. https://nx.hexdocs.pm/Nx.Defn.html. HexDocs documentation for Nx.Defn, accessed 2026-06-02.

[17] Numerical Elixir (Nx). 2026. Nx.indexed_add/4. https://nx.hexdocs.pm/0.11.0/Nx.html. HexDocs documentation for Nx.indexed_add/4 (Nx v0.11.0), accessed 2026-06-02.

[18] Numerical Elixir (Nx). 2026. Torchx. https://hex.pm/packages/torchx. Hex package for Torchx (LibTorch bindings and backend for Nx), accessed 2026-06-02.

[19] Numerical Elixir (Nx) Contributors. 2026. Nx: Multi-dimensional Arrays (Tensors) and Numerical Definitions for Elixir. https://github.com/elixir-nx/nx. GitHub repository, accessed 2026-06-01.

[20] NVIDIA Corporation. 2024. NCCL: NVIDIA Collective Communications Library. https://github.com/NVIDIA/nccl.

[21] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*. 10435–10444.

[22] The Elixir Team. 2026. Elixir Programming Language. https://elixir-lang.org. Official website, accessed 2026-06-01.

[23] TOP500 Project. 2026. TOP500. https://top500.org/. Official website of the TOP500 project for ranking and statistics of the world's most powerful supercomputers, accessed 2026-06-02.

[24] Jesper Larsson Träff. 2002. Implementing the MPI Process Topology Mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE Computer Society Press, 1–14.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008.

[26] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv preprint arXiv:2105.04663* (2021).

[27] Shiwei Zhang, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. 2022. Accelerating Large-Scale Distributed Neural Network Training with SPMD Parallelism. In *Proceedings of the 13th ACM Symposium on Cloud Computing (SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 403–418. doi:10.1145/3542929.3563487

[3] https://github.com/ElixirHPCLab