

MCR-DL: Mix-and-Match Communication Runtime for Deep Learning

Quentin Anthony
The Ohio State University
Columbus, OH
anthony.301@osu.edu

Ammar Ahmad Awan
Microsoft Corporation
Redmond, WA
ammar.awan@microsoft.com

Jeff Rasley
Microsoft Corporation
Redmond, WA
jeff.rasley@microsoft.com

Yuxiong He
Microsoft Corporation
Redmond, WA
yuxhe@microsoft.com

Aamir Shafi
The Ohio State University
Columbus, OH
shafi.16@osu.edu

Mustafa Abduljabbar
The Ohio State University
Columbus, OH
abduljabbar.1@osu.edu

Hari Subramoni
The Ohio State University
Columbus, OH
subramoni.1@osu.edu

Dhabaleswar Panda
The Ohio State University
Columbus, OH
panda.2@osu.edu

Abstract—In recent years, the training requirements of many state-of-the-art Deep Learning (DL) models have scaled beyond the compute and memory capabilities of a single processor, and necessitated distribution among processors. Training such massive models necessitates advanced parallelism strategies [1], [2] to maintain efficiency. However, such distributed DL parallelism strategies require a varied mixture of collective and point-to-point communication operations across a broad range of message sizes and scales. Examples of models using advanced parallelism strategies include Deep Learning Recommendation Models (DLRM) [3] and Mixture-of-Experts (MoE) [4], [5]. Communication libraries’ performance varies wildly across different communication operations, scales, and message sizes. We propose MCR-DL: an extensible DL communication framework that supports all point-to-point and collective operations while enabling users to dynamically mix-and-match communication backends for a given operation without deadlocks. MCR-DL also comes packaged with a tuning suite for dynamically selecting the best communication backend for a given input tensor. We select DeepSpeed-MoE and DLRM as candidate DL models and demonstrate a 31% improvement in DS-MoE throughput on 256 V100 GPUs on the Lassen HPC system. Further, we achieve a 20% throughput improvement in a dense Megatron-DeepSpeed model and a 25% throughput improvement in DLRM on 32 A100 GPUs with the Theta-GPU HPC system.

Index Terms—Neural Networks, DNN, MPI, GPU

I. INTRODUCTION

Distributed DL has become the standard training method for many state-of-the-art vision [6], language [7], [8], and recommendation [9] DL models. As the largest models grow from hundreds of millions [10] to hundreds of billions of parameters [7], new parallelization schemes have arisen to efficiently train DL models across thousands of processors [11], [1], [12]. While previous data-parallel DL models could heavily rely on a few collective operations (namely Allreduce), the model-parallel schemes of new models (e.g. sharding, pipeline and model parallelism, tensor slicing, etc) require a mixture of different collective and point-to-point operations

[3], [13], [11]. These advanced parallelization schemes rely heavily upon communication backends such as the NVIDIA Collectives Communication Library NCCL [14] and CUDA-Aware MPI libraries [15], [16]. However, modern communication backends have wildly varied performance characteristics across operations, within operations, and across releases (See Section I-C for a concrete example).

A. Problem Statement

There are two primary drawbacks to existing distributed DL frameworks’ communication: a lack of completeness in support for all communication operations/backends, and a lack of support for mixed-backend communication. Since modern distributed DL frameworks such as Horovod and PyTorch’s Distributed module do not support all MPI or NCCL operations (e.g. vectored collectives such as Gather), DL researchers are required to either: **(Option 1)**: implement their desired collectives via Point-to-Point operations (if point-to-point operations are supported in the chosen framework), or **(Option 2)**: transfer tensors between the distributed DL framework and an external MPI Python wrapper such as mpi4py [17]. Option 1 sacrifices the performance enhancements present in NCCL and most CUDA-Aware libraries, while option 2 introduces significant program complexity. For the second drawback, a lack of mixed-backend communication forces the user to decide where to sacrifice performance, since no communication backend performs all operations optimally (see Section I-C for a concrete example). These drawbacks bottleneck programmer productivity (e.g. a DL scientist must first implement an *MPI_Igather* before the intended optimization) and performance (e.g. NCCL performs well for Allreduce and MPI performs well for Alltoall. Which backend does one choose?), respectively.

B. Proposed Solution

We believe that a single unified interface between a given DL framework and the desired communication backend(s)

This research is supported in part by NSF grants 1818253, 1854828, 1931537, 2007991, 2018627, 2112606, and XRAC grant NCR-130002.

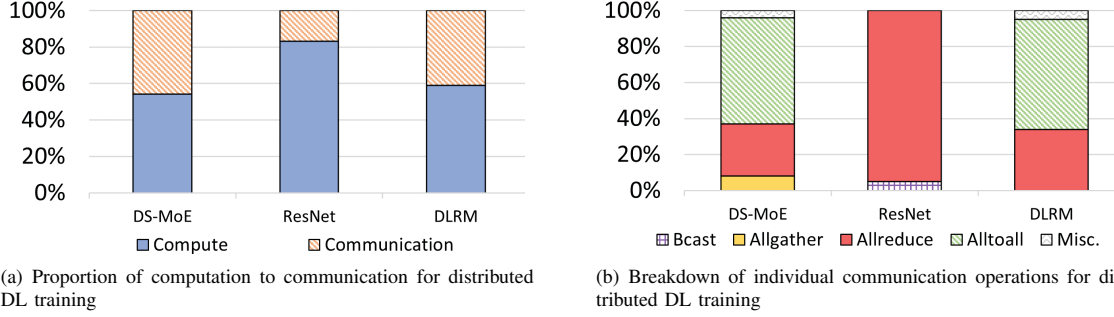


Fig. 1. Computation vs. Communication and breakdown of Communication operations breakdown for ResNet-50 (64 V100 GPUs on Lassen), DS-MoE (64 V100 GPUs on Lassen), and DLRM (32 A100 GPUs on Theta-GPU)

(MPI, NCCL, etc) will alleviate these performance and productivity bottlenecks, while introducing the possibility of mixed backend communication (e.g. MPI Alltoall and NCCL Allreduce).

In this paper, we introduce and evaluate a **Mix-and-Match Communication Runtime for Deep Learning (MCR-DL)**. Specifically, MCR-DL is a lightweight unified interface between the DL framework (PyTorch) and any combination of ABI-compatible¹ communication backends. MCR-DL users can dynamically switch between communication backends during distributed DL training. MCR-DL supports many existing communication backends (by implementing them as a high-level backend class), and provides an extensible design to enable new communication backends and performance optimizations.

C. Motivation

First, we profiled the computation and communication overhead of three representative DL models: DLRM and DeepSpeed-MoE (state-of-the-art hybrid-parallel DL models), and ResNet-50 (established data-parallel DL model). The overall computation vs. communication split as well as communication breakdown profiles are depicted in Figure 1. First, we note that data-parallelism is strongly compute-dominated, and its communication overhead is almost entirely made up of Allreduce. Therefore, data-parallel applications like ResNet-50 are able to achieve the best performance on existing monolithic distributed DL frameworks, and the choice of communication backend is simply determined by whichever library has the fastest CUDA-Aware Allreduce. We note that MCR-DL is still applicable to data-parallel frameworks with tuning (See Section V-F and Table II for details), but due to their much lower communication overhead, the benefits are marginal.

However, DLRM and DS-MoE have a significantly higher communication overhead at scale. Further, their communication operation requirements are heterogeneous. Therefore, there is a lot of room for mixing backends according to their strengths in order to improve training throughput.

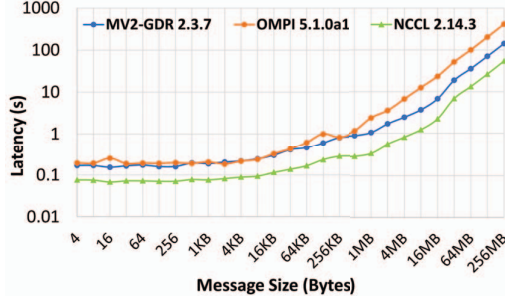
¹An Application Binary Interface (ABI), is the low-level interface between two program modules. An ABI determines such details as how functions are called and the size, layout, and alignment of datatypes. With ABI-compatibility, programs conform to the same set of runtime conventions.

Consider the case of DS-MoE. Given the communication breakdown in Figure 1(b) and the collective performance in Figure 2, which communication backend should be used? A myriad of application questions would need to be answered such as which collectives DS-MoE uses, their relative frequencies, and the range of message sizes for each collective. Any decision on a single communication backend will lose out on some collectives and at some message ranges. Specifically, since DS-MoE relies mostly on Allreduce and Alltoall, we could refer to Figure 2 and reduce communication overhead by applying MVAPICH2-GDR for Alltoall and NCCL for Allreduce. However, such a decision will need to be reevaluated at each subsequent release cycle of the communication backends. If the user is able to dynamically switch among communication backends, they could squeeze more performance out of their application while reducing the setup cost of changing communication backends if future communication backend releases change.

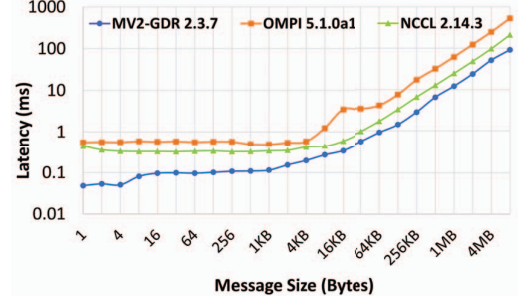
D. Contributions

Our contributions are as follows:

- C1) We proposed, designed, and evaluated MCR-DL: an extensible, scalable API for DL communication operations. MCR-DL supports all point-to-point and collective communication operations on PyTorch tensors, and all collective communication libraries (Section V-A)
- C2) We enabled deadlock-free mixed-backend DL communication via fine-grained synchronization techniques (Section V-D)
- C3) We fully implemented MCR-DL in a C++ backbone underneath a thin Python layer, and achieved a maximum of 5% overhead (compared to a pure micro-benchmark written in C) for small messages and a 1% overhead for large messages (down from 18% and 4% in PyTorch distributed, respectively) (Figure 7)
- C4) MCR-DL offers up to 31% throughput improvement (12% in scaling efficiency) in DeepSpeed-MoE and 25% throughput improvement (14% improvement in scaling efficiency) in DLRM by dynamically selecting the best-performing communication backend at each scale and message size (Figures 8 and 9)



(a) 64 GPUs (16 node 4 ppn) - iAllreduce



(b) 64 GPUs (16 nodes 4 ppn) - Alltoall

Fig. 2. Comparison of communication backends' collective performance on basic micro-benchmark with 64 V100 GPUs on Lassen

Studies	Features					
	Point-to-Point	Collectives	Vector Collectives	Non-Blocking Operations	Mixed-Backend Communication	Backend as a Class
Horovod	×	✓	×	NCCL Only	Experimental	×
PyTorch Distributed Module	✓	✓	×	NCCL Only	×	✓
LBANN	✓	✓	×	✓	×	×
mpi4py[17]	✓	✓	✓	✓	×	×
Proposed MCR-DL	✓	✓	✓	✓	✓	✓

TABLE I
FEATURES OFFERED BY MCR-DL COMPARED TO EXISTING FRAMEWORKS

- C5) Define and implement a tuning framework within MCR-DL that enables the best communication backend to be automatically selected for each communication operation (Section V-F)
- C6) Demonstrate the extensibility of MCR-DL by adding support for communication compression, logging, and tensor fusion (Section V-E)

II. RELATED WORK

A. DL Communication Framework Design

DeepSpeed [12] uses PyTorch's distributed module [18] to implement optimized DL communication at extreme scales. Recently, DeepSpeed has added support for Mixture-of-Experts (MoE) DL models [5], [4]. Horovod [19] is a data-parallel focused framework that experimentally supports mixed communications without deadlock-avoidance support. The Livermore Big Artificial Neural Network Toolkit (LBANN) is an HPC-centric distributed DL framework that supports multiple parallelism levels. The MPI for Python package [17] supplies Python bindings for the MPI standard. Our work competes with these works by seeking to unify communication calls into a single interface built atop PyTorch.

B. Mixing MPI with an External Framework

The work in [20] combined an MPI runtime with UPC in a deadlock-free architecture by unifying the runtimes. The resulting runtime shared resources between MPI and UPC to avoid data-dependencies. In recent releases, the MVAPICH2-GDR [16] CUDA-Aware MPI library has added support for NCCL collectives. However, this support is not optimized for non-blocking communication operations like those required by DLRM. Aluminum [21] is a DL-focused communication

library built on MPI and NCCL, but is focused on latency-bound communication operations. Our work is complementary to the above works, since we choose the best backend for each communication operation.

C. Scaling Mixture-of-Experts and DLRM Models

The work in [13] scaled a 600 billion parameter Mixture-of-Experts (MoE) model to 2048 TPU v3 processors. DeepSpeed has recently added support for MoE DL models [4] and scaled beyond a trillion parameters [5]. MoE models are gradually being applied to other domains such as vision [22]. DLRM [3] has scaled beyond a trillion parameters with 4D parallelism techniques [9]. We demonstrate that our work further improves the scaling behavior of these complex parallel DL models.

III. BACKGROUND

A. DL Training

Distributed DL can take several forms: data-parallelism, model-parallelism, and hybrid-parallelism. Data-parallelism places a full model replica on each processor, and splits the training data among processors. Model parallelism splits the model across processors, and propagates each data sample through each device. Hybrid-parallelism splits the model across sets of processors, and splits the training data among complete-model sets of processors. There are tradeoffs for each parallelism scheme: data-parallelism is the simplest and has low communication overhead but is restricted to models that fit in processor memory. Hybrid and model-parallelism can accommodate any model size, but can require complex communication with high overheads. All distributed DL schemes are increasingly deployed on HPC systems [23], [7].

B. Distributed DL Frameworks

Horovod is a distributed DL framework with a focus on distributed data-parallelism to train DNNs [19]. As such, Horovod primarily relies on Allreduce and Bcast collectives. Due to Horovod’s focus, they provide a simple API, quick installations, and powerful data-parallel optimizations and profiling tools. Horovod supports many major DL frameworks and communication backends, including MPI and NCCL [14].

PyTorch’s distributed module is a built-in communication API within the PyTorch [24] DL framework. PyTorch distributed supports most communication operations, and contains several optimizations for distributed training (e.g. mixed-precision, gradient bucketing, sharded optimizer states). While official PyTorch wheels come packaged with the NCCL backend, other backends require a PyTorch source installation.

DeepSpeed is a distributed DL framework built atop PyTorch’s distributed module. DeepSpeed’s focus is on efficient training of large-scale models that don’t fit into a single processor’s memory. A myriad of parallelism schemes and optimizer sharding techniques are included in DeepSpeed.

C. Communication Backends

MPI is a parallel programming standard that enables processes to communicate with each other. CUDA-aware MPI libraries such as SpectrumMPI [25], OpenMPI [15], and MVAPICH2 [16] provide optimized support for heterogeneous systems containing GPUs. GPU communication optimizations such as staging, CUDA Inter-Process Communication (IPC), and GPUDirect RDMA enable MPI libraries to provide superior performance across different combinations of GPU and interconnect [26].

NCCL implements optimized collective communication patterns for NVIDIA GPUs [14]. The various collective communication primitives found in NCCL are: Allgather, Allreduce, Reduce, ReduceScatter, Alltoall, Point-to-Point, and Broadcast. NCCL is not MPI-compliant, however, and does not provide support for many common MPI operations such as gather, scatter, and variable message-size collectives. Microsoft’s Synthesized Collective Communication Library (MSCCL) [27] creates custom collective algorithms for a given hardware topology. MSCCL supports both AMD and NVIDIA GPUs, and supports all major collective operations.

D. Mixture-of-Experts

Mixture-of-experts (MoE) is an ensemble machine learning technique where a collection of “expert” feed-forward networks (FFNs) are trained on subtasks of the problem. Only a few experts are applied to a given data sample. In recent years, the MoE technique has been applied to transformer DL models in an effort to increase the model size (and therefore accuracy) while lessening the computational burden. MoE models require less computation to train than equivalent standard (i.e. “dense”) models because each token only propagates through an expert subset of the full model. Incoming tokens are routed to existing expert FFNs via a gating function, and this routing as well as its subsequent

combination of FFN outputs require Alltoall operations. Such Alltoall operations scale with the number of devices, and quickly become a dominant communication overhead at large scales. The distributed DL frameworks DeepSpeed [5], [4] and Fairseq [28] have recently added support for MoE transformer models.

E. Deep Learning Recommendation Models

Deep Learning Recommendation Models (DLRMs) are a family of recommendation models that rely upon at least one deep neural network (DNN) [3], [9]. Such models are composed of sparse embedding tables and dense multilayer perceptrons (MLPs). Note that a MLP is a special case of an FFN where every layer is fully connected to the next layer in the network. While sparse categorical data must be processed via embedding lookups (and are memory-bound), dense continuous data is fed through the bottom MLPs (and are compute-bound). The MLPs are trained via data-parallelism, and hence depend on Allreduce. The embedding tables are split across processes, and must be shuffled with an Alltoall prior to being fed into the top MLP. Each batch’s Alltoall operation is overlapped with the previous top MLP’s forward pass from the previous batch, which necessitates non-blocking Alltoall.

IV. CHALLENGES

The key challenge addressed in this paper is: *Can we improve the interface between a DL framework and communication backends with a single unified framework built on top of PyTorch?* We seek to create an extensible framework that encapsulates all MPI and NCCL functionality. To answer this broad question, we solve the following concrete challenges:

- What are the key communication needs of modern distributed DL models and frameworks? Do existing distributed DL frameworks provide these needs?
- Can a unified framework improve rapid prototyping for DL parallelism schemes while enabling mixed-backend communications?
- What benefits can mixed-backend communications provide to improve DL training throughput?

V. DESIGN

MCR-DL is split into a C++ implementation layer underneath a thin Python wrapper. Each backend is implemented as an object of a class, and implements the MCR-DL API in accordance with each backend’s requirements.

A. MCR-DL API

MCR-DL implements all communication operations as depicted below in Listing 1.

```
1 def get_backends()
2 def init(list<str> backends)
3 def finalize(list<str> backends)
4 def synchronize(list<str> backends)
5 def get_size(str backend)
6 def get_rank(str backend)
7 def send(str backend, torch.Tensor t, int rank,
   bool async_op)
```

```

8 def recv(str backend, torch.Tensor t, int rank,
  bool async_op)
9 def all_to_all_single(str backend, torch.Tensor
  output, torch.Tensor input, bool async_op)
10 def all_to_all(str backend, list<torch.Tensor>
  output, list<torch.Tensor> input, bool async_op
  )
11 def all_reduce(str backend, torch.Tensor output,
  ReduceOp op, bool async_op)
12 def all_gather(str backend, torch.Tensor output,
  torch.Tensor input, bool async_op)
13 def gather(str backend, torch.Tensor output, int
  root, bool async_op)
14 def scatter(str backend, torch.Tensor output, int
  root, bool async_op)
15 def reduce(str backend, torch.Tensor output, int
  root, ReduceOp op, bool async_op)
16 def reduce_scatter(str backend, torch.Tensor output
  , int root, ReduceOp op, bool async_op)
17 def bcast(str backend, torch.Tensor output, int
  root, bool async_op)
18 def gatherv(str backend, torch.Tensor output, int
  root, list<int> rcounts, list<int> displs, bool
  async_op)
19 def scatterv(str backend, torch.Tensor output, int
  root, list<int> scounts, list<int> displs, bool
  async_op)
20 def all_to_allv(str backend, torch.Tensor output,
  torch.Tensor input, list<int> scounts, list<int>
  > rcounts, list<int> def sdispls, list<int>
  rdispls, bool async_op)
21 def all_gatherv(str backend, torch.Tensor output,
  int root, list<int> rcounts, list<int> displs,
  bool async_op)

```

Listing 1. High-level MCR-DL API

```

1 # Before MCR-DL
2 def allgather_host(self,
3     comm,
4     cupy_sign,
5     cupy_rbuf_sign,
6     cupy_scale,
7     cupy_rbuf_scale):
8
9     # 1. Convert cupy to numpy
10    numpy_rbuf_sign = np.zeros(
11        [comm.Get_size(),
12         cupy_sign.size],
13        dtype=cupy_sign.dtype)
14    numpy_rbuf_scale = np.zeros([comm.Get_size(),
15                                1],
16                                dtype=
17    cupy_scale.dtype)
18
19    numpy_sign = cupy.asnumpy(cupy_sign)
20    numpy_rbuf_sign = cupy.asnumpy(cupy_rbuf_sign)
21    numpy_scale = cupy.asnumpy(cupy_scale)
22    numpy_rbuf_scale = cupy.asnumpy(cupy_rbuf_scale
23    )
24    cupy.cuda.get_current_stream().synchronize()
25
26    # 2. Communicate numpy buffers
27    comm.Allgather(numpy_sign, numpy_rbuf_sign)
28    comm.Allgather(numpy_scale, numpy_rbuf_scale)
29    comm.Barrier()
30
31    # 3. Convert numpy back to cupy
32    cupy_sign = cupy.asarray(numpy_sign)
33    cupy_rbuf_sign = cupy.asarray(numpy_rbuf_sign)
34    cupy_scale = cupy.asarray(numpy_scale)
35    cupy_rbuf_scale = cupy.asarray(numpy_rbuf_scale
36    )
37    cupy.cuda.get_current_stream().synchronize()

```

```

35
36     return cupy_sign, cupy_rbuf_sign, cupy_scale,
37         cupy_rbuf_scale
38
39 # After MCR-DL
40 def allgather_host(self,
41     comm,
42     sign,
43     rbuf_sign,
44     scale,
45     rbuf_scale):
46
47     comm.all_gather_base(rbuf_sign, sign)
48     comm.all_gather_base(rbuf_sign, sign)
49
50     return sign, rbuf_sign, scale, rbuf_scale

```

Listing 2. Example of simplified prototyping with MCR-DL

There are a few key takeaways from this API listing:

- All operations take either a single **backend** string that matches an underlying backend class (e.g. "mv2-gdr", "nccl", etc) or a special backend flag "auto", which will dynamically choose the best message size for a given scale and message size if tuning tables are available. (Note: MCR-DL comes packaged with a tuning suite which first runs communication operation benchmarks for each backend, and uses this data to map each message size, scale, and operation to a given backend. This optimal backend choice is then used at runtime if "auto" is chosen)
- We conform to the PyTorch distributed module API conventions when possible to ease code refactoring to MCR-DL. An example of this is *all_to_all*, which shuffles lists of tensors rather than individual tensor elements. This is a common usecase in distributed PyTorch applications. Another example is *all_to_all_single*, which directly shuffles the tensor elements themselves on each rank.
- Vectored collectives (e.g. *gather*/*scatter*) and non-blocking collectives are supported for all backends.

B. Advanced Communication Support

Most distributed DL frameworks do not support the full underlying communication backend, only the operations that matter for DL parallelism (e.g. Allreduce). If a user needs a communication operation that is not currently supported by their distributed DL framework (e.g. advanced parallelism or data processing), they would need to sacrifice performance or productivity as mentioned in Section I-C.

MCR-DL is a thin layer atop each currently-supported backend, and fully implements each backend on PyTorch tensors (See Figure 3 for the software stack). The MCR-DL "Backend" class can be easily extended to new communication backends such as MSCCL [27], Gloo, oneAPI, etc.

C. Synchronization

One of the most important design considerations for a distributed framework is synchronization. We seek to add enough synchronization to rid the programmer of having to

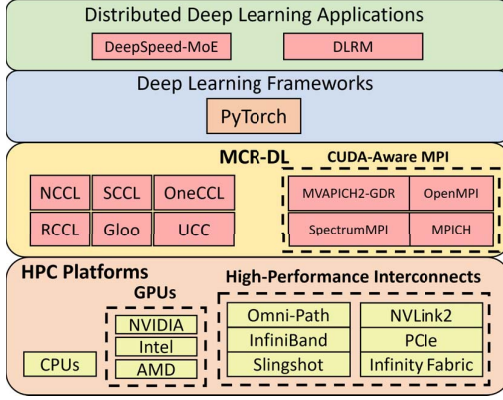


Fig. 3. The MCR-DL Software Stack. MCR-DL is a thin layer between a target DL framework and the HPC system, and supports any number of stream-aware communication backends along with CUDA-Aware MPI.

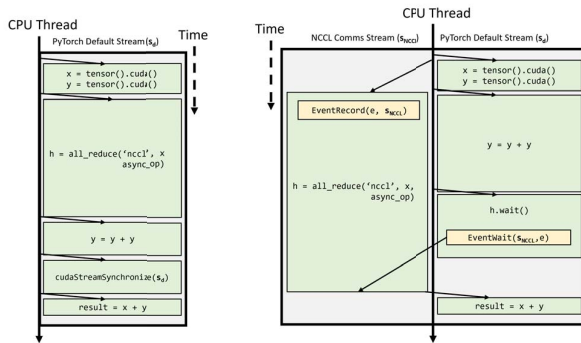
frequently debug deadlocks and data validation issues, while achieving enough overlap to maintain high performance at scale. With the right synchronization strategy, we are able to efficiently both overlap computation with communication, and overlap across communication backends without deadlocks or data validation issues.

```

1 import torch
2 import mcr_dl
3
4 def tensor():
5     return torch.rand(1,1)
6
7 x = tensor().cuda()
8 y = tensor().cuda()
9 mcr_dl.init('nccl')
10
11 h = mcr_dl.all_reduce('nccl', x, async_op=True)
12 y = y + y
13 h.wait('nccl')
14 result = x + y

```

Listing 3. Example of available overlap between communication and computation in a DL setting



(a) Naive synchronization (b) Synchronization in MCR-DL

Fig. 4. Synchronization diagrams of Listing 3 for the naive scheme and MCR-DL’s fine-grained CUDA event scheme

First consider a naive synchronization scheme where a) all communication operations are posted to the PyTorch default stream, and b) we synchronize operations with `cudaStreamSynchronize` on that stream. We demonstrate the behavior of

this scheme in Listing 3, a prototypical example of available communication/computation overlap faced in distributed DL. The resulting serial execution is depicted in Figure 4(a)². In MCR-DL, we exploit communication/computation overlap by creating a pool of **communication streams** for each backend. These streams are managed internally to MCR-DL. Communication operations posted to a backend’s stream(s) are synchronized with fine-grained CUDA events. For figure 4(b), this translates to: (1): An `all_reduce(x)` operation is posted to a NCCL communication stream in MCR-DL, and a distributed work handle is stored in `h`, (2): MCR-DL records a CUDA event `e` onto the communication stream and begins executing the `all_reduce(x)`, (3): the PyTorch default stream is able to progress with operations unrelated to `x`, (4): when a data-dependency on `x` is encountered, the user must call `wait()` on the work handle `h`, which MCR-DL uses internally to wait on the prior event `e`.

This scheme is similar to PyTorch’s distributed module, but there are a few key implementation details that enable greater performance: (1): The use of multiple streams enables concurrent small-message operations (concurrent large-message operations are bandwidth-bound and show no benefit), (2): Instead of having an overall communication stream, each backend contains its own stream for overlap across backends. This synchronization behavior is extended to multiple backends in MCR-DL, which we will now discuss.

D. Mixed-Backend Communications

Since MCR-DL is a thin layer atop communication backends, we can pass the desired backend for any given communication operation dynamically within a Python script. An example of this is depicted below in Listing 4.

```

1 import torch
2 import mcr_dl
3
4 def tensor():
5     return torch.rand(1,1)
6
7 x = tensor().cuda()
8 y = tensor().cuda()
9 z = tensor().cuda()
10 mcr_dl.init(['nccl', 'mpi'])
11
12 h1 = mcr_dl.all_reduce('nccl', x, async_op=True)
13 h2 = mcr_dl.all_reduce('mpi', y, async_op=True)
14 z = z + z
15 h1.wait()
16 h2.wait()
17 result = x + y + z

```

Listing 4. Example of explicit mixed-backend communications in MCR-DL. All inter-backend synchronization is performed internally. MCR-DL can dynamically choose the best backend to use at runtime if `'auto'` is passed as the backend (See Section V-F)

However, each communication backend conforms to its own synchronization scheme. NCCL and its derivatives are synchronized on the CUDA streams, while MPI is synchronized on a host thread. If we are to mix backends without deadlocks,

²The length of operation boxes in Figures is purely for synchronization discussion

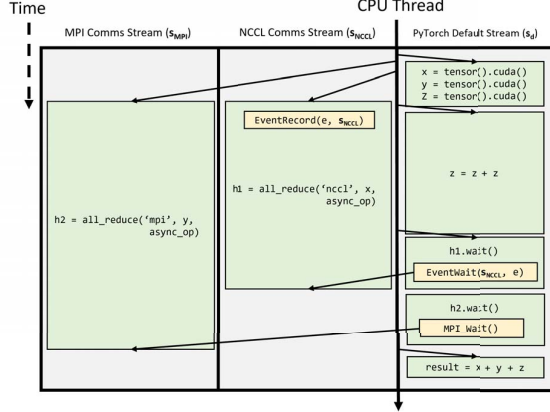


Fig. 5. In MCR-DL, communication backends can be explicitly chosen, or users can dynamically choose the best backend for a given operation with "auto"

we will need to loop over each implemented backend and synchronize with their respective thread/stream. For the mixture of CUDA-Aware MPI and non-blocking NCCL, for example, this entails a call to CUDA-event based synchronization for NCCL (as discussed in section V-C), followed by an `MPI_Wait` for MPI. Handling CUDA-aware MPI is a challenge since CUDA streams are not exposed by MPI to the application, this leads to two options (which MCR-DL provide at the initialization of an MPI backend): (1): Allow MPI to handle all streams, which sacrifices some MCR-DL overlap across backends, but preserves multiple CUDA stream logic (if it exists) within MPI. (2): Intercept calls to `cudaStreamCreate` and manage streams in MCR-DL, which exploits overlap across backends, but could potentially lead to deadlocks if multi-stream logic is used in MPI³. An example of streams managed by MCR-DL is depicted by Figure 5. For ease of synchronization, every work handle's `wait()` call waits on the PyTorch default stream (i.e. synchronization purely between communication streams is not supported). We note that stream-aware MPI like the implementation by MPICH [29] allows MCR-DL to fully overlap communication backends by self-managing streams.

While Figure 5 depicts the mixture of a stream-aware backend (NCCL) and a backend without streams exposed to the user (MPI), the combination of any number of stream-aware backends (NCCL, SCCL, etc) is supported in MCR-DL and synchronized with CUDA events. Further, the combination of ABI-compatible MPI backends is supported⁴. In our experiments, the initialization overhead for multiple communication libraries is negligible after being amortized over a few (< 10) DL training steps.

E. Communication Optimization Extensibility

In PyTorch's distributed module and Horovod, there are a number of communication optimizations (e.g. Tensor Fusion,

³In our experiments, we find that the best choice for this option is dependent on the MPI library

⁴In our experiments, we found that mixing at most one non-stream-aware backend is optimal for overlap

Padding, etc) built atop the communication layer to improve performance. Similarly, by encapsulating all communication operations into MCR-DL, these optimizations can be easily integrated into all communication operations and backends. One can utilize the rich Python ecosystem to insert optimizations into MCR-DL's Python layer as depicted in Figure 6. As examples, we have implemented lossy communication compression with zfp [30], Tensor Fusion (combining small tensors into a bandwidth-optimal large tensor), and communication logging (which is used to generate Figures 1 and 12). Further, future optimizations (e.g. persistent collectives) can be easily added with minimal changes among backends and operations. These optimizations can be applied to incoming messages with only a few lines of Python code before routing the operation to its respective C++ backend.

There are two parameters for Tensor Fusion: the maximum fusion buffer size B and the maximum time T to wait for that fusion buffer to fill with small tensors. MCR-DL introduces a small optimization for Tensor Fusion, where if the Fusion buffer does not reach B before T (and therefore does not saturate bandwidth), the communication is overlapped with other backends' Fusion buffers, if available. This Tensor Fusion optimization is used in all DL training results in Section VI.

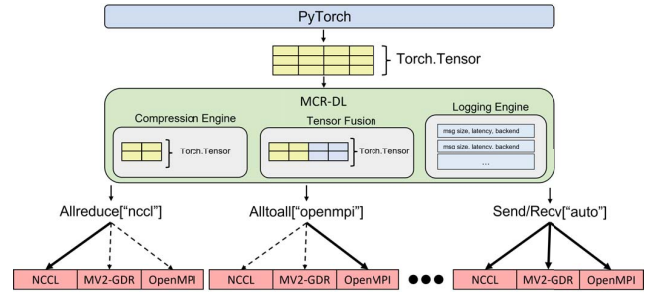


Fig. 6. In MCR-DL, communication backends can be explicitly chosen, or users can dynamically choose the best backend for a given operation with the "auto" backend option. Further, MCR-DL routes all communication operations through an optional set of optimizations, including Tensor Fusion (combining small tensors into a bandwidth-optimal large tensor), message compression, and logging.

F. Communication Tuning

Tuning is an established problem in distributed communication [31], [32], [33]. MCR-DL comes packaged with a tuning suite that seeks to map an input communication operation (with associated message size) to the best-performing backend (e.g. `all_reduce` $\rightarrow$ NCCL). This introduces additional complication since not only are distinct communication operations mixed-backend (e.g. `all_reduce` and `gather`), but MCR-DL allows a single operation to choose the best backend with the "auto" backend option (e.g. `mcr_dl.gather("auto")` routes `{small-message gather}` $\rightarrow$ MPI, and `{large-message gather}` $\rightarrow$ NCCL). This behavior is depicted in Figure 6, where solid lines depict the backend chosen for a given operation.

This tuning is implemented as a static tuning table. The tuning suite is composed of a set of micro-benchmark scripts

Message Size	Backend
256	MVAPICH2-GDR
512	MVAPICH2-GDR
1024	MVAPICH2-GDR
2048	MVAPICH2-GDR
4096	NCCL
8192	NCCL
16384	SCCL
32768	SCCL

TABLE II
EXAMPLE TUNING TABLE FOR THE ALL_GATHER COLLECTIVE OPERATION
AT A SINGLE WORLD SIZE GENERATED BY MCR-DL

that evaluate end-to-end time on a set of overlapped communication operations for each backend. By choosing the backend with the minimum end-to-end time for each input tensor size, MCR-DL generates a table like Table II for each world size (i.e. the number of GPUs) trained over. Every collective requires its own static tuning table. The size of each collective’s tuning table is dependent both on the number of specific message sizes we wish to tune for, as well as the number of scales (world size) we are tuning over. Specifically, a given table entry is first mapped by the world size, then by the message size. Therefore, the total number of tuning table entries is given by: $(\text{Num_Collectives} \times \text{Num_Scales} \times \text{Num_Message_Sizes})$. Since the performance of each communication backend depends heavily on the combination of inter-node fabric, intra-node fabric, and compute hardware used, tuning tables are not transferable across HPC systems. However, we find that general trends tend to hold across systems with a coarsely similar architecture (e.g. MVAPICH2-GDR consistently performs the best for small messages).

VI. PERFORMANCE CHARACTERIZATION

1) Node Architecture

All experimental evaluations⁵ were carried out on the Lassen cluster at Lawrence Livermore National Laboratory and the ThetaGPU cluster at Argonne Leadership Computing Facility [34]. Lassen is composed of 792 nodes each consisting of four 16 GB NVIDIA V100 GPUs and two 44-core IBM Power 9 CPUs. Nodes are connected via Mellanox Infiniband

⁵The choice of cluster for a given application was purely made out of external factors such as available compute and ease of software compatibility

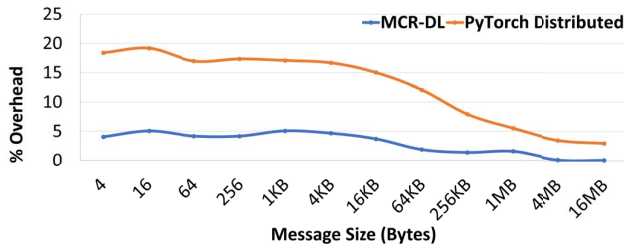


Fig. 7. Overhead over OMB for MCR-DL and PyTorch Distributed for a fixed backend on ThetaGPU (32 A100 GPUs). MCR-DL reduces overhead by ensuring top-level Python logic is minimal.

EDR in a fat-tree topology. ThetaGPU is composed of 24 NVIDIA DGX A100 nodes, each containing two AMD Rome CPUs and eight 40 GB NVIDIA A100 GPUs.

2) Communication Backends

We used a mixture of MVAPICH2-GDR 2.3.7 [16], OpenMPI v5.1.0 [15] (built with UCX v1.13.1), the latest MSCCL [27], and NCCL 2.14.3-1 [14] for all DL experiments. All backends and frameworks were built with CUDA 11.4.152 on ThetaGPU and CUDA 11.4.100 on Lassen.

3) Software Libraries

All micro-benchmark evaluations were carried out with OSU Micro-Benchmarks (OMB) 6.1. For our DL evaluations, we used source-built PyTorch v1.12.1 and DeepSpeed v0.7.4.

4) DL Training Settings

For both DS-MoE and DLRM, we had to replace all dependencies on PyTorch’s distributed module with MCR-DL calls. Since MCR-DL conforms to the PyTorch API wherever possible, this step is a straightforward search-and-replace.

We trained a 4B parameter DS-MoE model (350M+PR-MoE-32/64) on the Pile [35]. For more details on this model and on DS-MoE, see [5].

For DLRM, we trained 100 synthetic data batches of size 8k with bottom and top MLPs of size (512-512-64) and (1024-1024-1024-1), respectively. The embedding table size used is $1e6 \times (\text{num_ranks})$.

The dense Megatron-DeepSpeed model contained 6.7B parameters with a model-parallelism degree of 2 and ZeRO stage 2. It was also trained on the Pile [35].

A. Micro-Benchmarks

Before proceeding to application-level performance evaluations, we first created simple collective and point-to-point benchmarks to ensure MCR-DL doesn’t introduce significant performance overhead when compared to micro-benchmarks implemented at the C-level, as investigated earlier with OMB. As demonstrated in Figure 7, MCR-DL introduces an overhead of around 5% for small MPI_Alltoall operations (under 4kB). However, this overhead quickly reduces to 1% in the MB message range, which is the message range expected for most DL training applications [36]. PyTorch’s distributed module built atop MVAPICH2-GDR, however, has a high overhead (18%) for small messages, and converges to a higher overhead (4%) in the MB message range. MCR-DL doesn’t introduce significant overhead for communication operations.

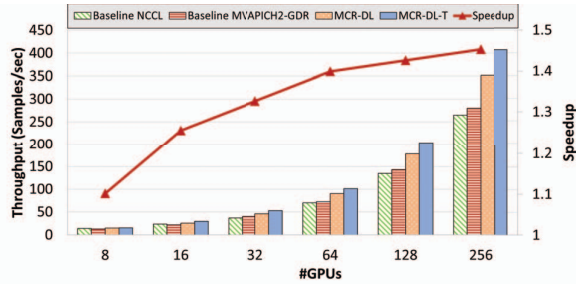
In order to spare users the OMB evaluations like Figure 2, we created a tuning suite to generate a static tuning table for later use in applications. The tuning suite first runs basic collective and point-to-point evaluations over a range of message sizes, scales, and backends. Then, the tuning scripts create a tuning table which maps a given message size and number of processes to a given communication backend. The tables for Lassen and ThetaGPU are used in subsequent DL evaluations. This tuning table is used whenever the “auto” backend is passed to a collective as described in Section V. The difference between static-backend mixing and tuned

mixing is depicted in all DL training figures as MCR-DL and MCR-DL-T, respectively.

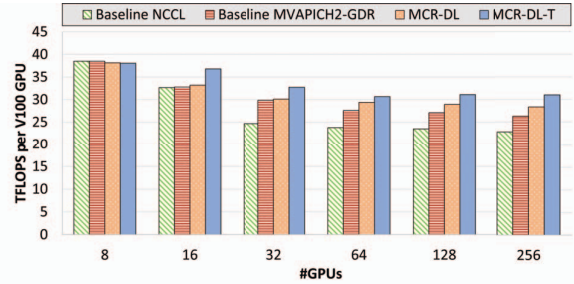
B. DL Training

With the setup described above in VI-1 through VI-4, we carried out DL training evaluations with MCR-DL on the Lassen HPC system. Baseline experiments were carried out with PyTorch’s distributed module built against a single communication backend (e.g. “Baseline SCCL” is PyTorch distributed built with the SCCL backend). Neither tensor fusion nor compression from Section V-E were used in evaluations⁶. Further, to compare coarse-grained mix-and-match (i.e. one backend per collective such as NCCL *Allreduce* and MPI *Alltoall*) against fine-grained mix-and-match (i.e. one

backend per (collective, message size) pair such as NCCL *Allreduce* for 1MB messages and MPI *Allreduce* for 512KB messages. These two settings of MCR-DL are depicted in Figures 8-10 as **MCR-DL** and **MCR-DL-T**, respectively. First, we run pre-training throughput experiments DS-MoE for pure NCCL, pure MVAPICH2-GDR and mixed backends. Results are depicted in 8(a). At smaller scales, NCCL performs better than MVAPICH2-GDR because Alltoall is not yet a dominant factor in communication time. We see a crossover threshold from Allreduce-bound to Alltoall-bound communication at around 32 GPUs, beyond which MVAPICH2-GDR’s improved Alltoall starts to show benefits. The performance difference between pure NCCL and pure MVAPICH2-GDR is still small, however, because NCCL’s Allreduce collective is more performant than MVAPICH2-GDR’s at this message range.

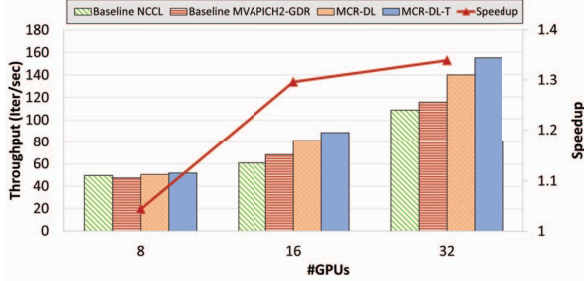


(a) DS-MoE Throughput

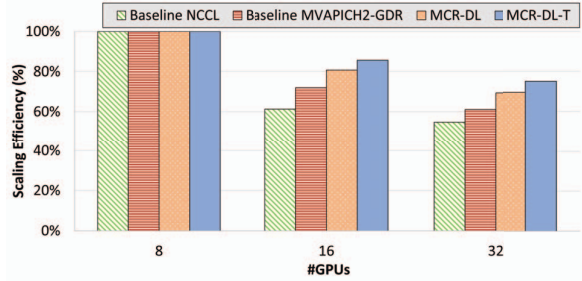


(b) DS-MoE Scaling Efficiency

Fig. 8. Throughput and scaling efficiency improvements for DS-MoE with pure MVAPICH2-GDR, pure NCCL, and mixed-backends with MCR-DL on Lassen

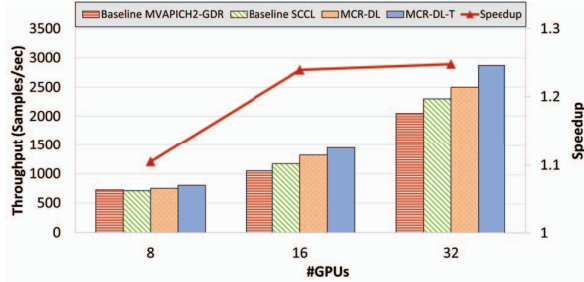


(a) DLRM Throughput

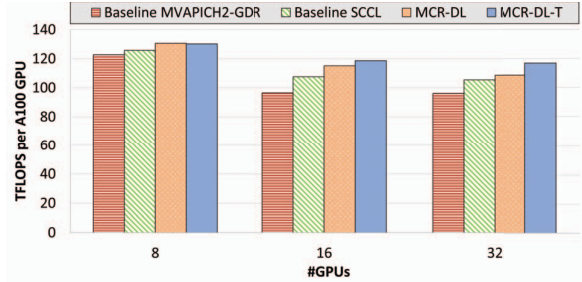


(b) DLRM Scaling Efficiency

Fig. 9. Throughput and scaling efficiency improvements for DLRM with pure MVAPICH2-GDR, pure NCCL, and mixed-backends with MCR-DL on ThetaGPU



(a) Megatron-DeepSpeed Dense Model Throughput



(b) Megatron-DeepSpeed Dense Model Scaling Efficiency

Fig. 10. Throughput and scaling efficiency improvements for dense Megatron-DeepSpeed with pure MVAPICH2-GDR, pure SCCL, and mixed-backends with MCR-DL on ThetaGPU

MCR-DL is able to exploit MVAPICH2-GDR’s improved Alltoall and NCCL’s improved Allreduce to perform best at all scales without deadlocks. At 256 GPUs, we see a 31% improvement over pure MVAPICH2-GDR and a 35% improvement over pure NCCL. Scaling efficiency 8(b) is also greatly improved with MCR-DL, maintaining a 81% efficiency at 256 V100 GPUs.

Second, we have evaluated pure NCCL, pure MVAPICH2-GDR and mixed backends on the ThetaGPU HPC system for DLRM. Results are depicted in Figure 9(a). NCCL again beats MVAPICH2-GDR at small scales due to its improved Allreduce. At higher scales, MVAPICH2-GDR again starts to perform better due to Alltoall’s scaling, and MCR-DL is able to use each backend’s strengths to improve performance, achieving a 25% improvement over pure MVAPICH2-GDR and a 30% improvement over pure NCCL. Scaling efficiency is less that of DS-MoE, but still improved by MCR-DL, maintaining a 75% efficiency at 32 A100 GPUs.

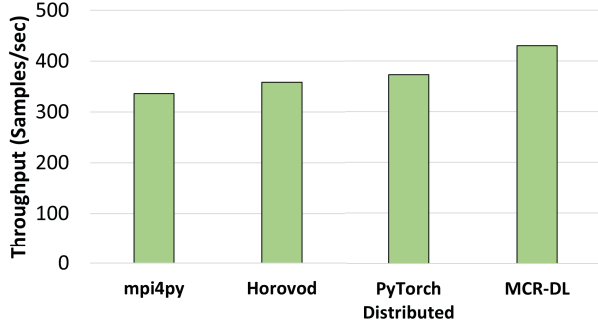


Fig. 11. Comparison of MCR-DL against competing PyTorch-compatible frameworks on a Mixture-of-Experts transformer using 256 Lassen V100 GPUs.

In order to directly compare the performance of MCR-DL with all PyTorch-compatible⁷ competing frameworks in Table I-B, we swapped all communication operations in Megatron-DeepSpeed with each respective framework’s implementation. The results on 256 Lassen V100 GPUs is depicted in Figure 11. In order to compare each framework’s best performance, MCR-DL, Horovod, and PyTorch-distributed were run with tensor fusion enabled, which leads to the performance gap between mpi4py and both Horovod and PyTorch-distributed. MCR-DL performs the best due to its mixed-backend optimizations coupled with tensor fusion.

For completeness, we have also trained a dense Megatron-DeepSpeed model on the ThetaGPU cluster with a mixture of MSCCL [27] and MVAPICH2-GDR [16]. As a secondary result, we have taken the compute vs. communication breakdown for DS-MoE and DLRM when using MCR-DL at 256 Lassen V100 GPUs and 32 ThetaGPU A100 GPUs, respectively. MCR-DL is an important component in reducing the computation bottleneck at scale, demonstrating a 9% reduction in communication time for DS-MoE and a 7% reduction in communication time for DLRM.

⁷LBANN does not provide any MoE implementation, and is not compatible with any mainstream DL frameworks such as PyTorch

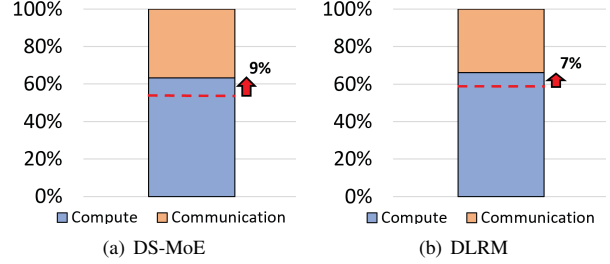


Fig. 12. Communication overhead reduction with MCR-DL at 256 Lassen V100 GPUs (DS-MoE) and 32 ThetaGPU A100 GPUs (DLRM).

VII. DISCUSSION

The throughput and scaling efficiency improvements in Figures 8, 9, and 10 demonstrate that a mixed-backend DL communication framework can significantly improve the performance of emerging DL models by reducing the communication bottleneck. Further, it was confirmed that a C++ backbone underneath a thin Python layer ensures low-overhead communication operations, which enables the exploration of small-message latency-bound operations for emerging models.

These results are in agreement with the original observation that modern communication backends vary widely in performance characteristics across operations, within operations, and across releases. By mix-and-matching backends for a given operation (and within an operation), significant communication performance improvements were achieved. Further, since our communication operations are implemented in low-latency C++ code underneath a thin Python interface, we have maintained low overhead while ensuring compatibility with Python-based DL frameworks.

The performance improvements inherent in mixing communication backends are consistent with the findings of previous NCCL and MPI studies [37] and studies exploring the mixture of MPI with external runtimes in [20].

VIII. CONCLUSION

State-of-the-art deep learning (DL) models are pushing the boundaries of existing fields while pioneering entirely new areas of study. However, such DL models are often impossible or impractical to train on single processors or small-scale workstations. Further work in novel parallelism schemes and optimizations will require a robust and extensible interface between DL frameworks and communication backends. In this paper, we present and evaluate MCR-DL: a Mix-and-Match Communication Runtime for DL. MCR-DL supports all communication operations and backends, and enables mixed-backend communication to ensure the most performant backend is being used for a given communication operation. The proposed design is demonstrated on state-of-the-art DL models such as DLRM [3] and Mixture-of-Experts (MoE) [4], [5]. We report up to a 31% improvement in DeepSpeed-MoE throughput on 256 V100 GPUs on the Lassen HPC system and a 25% improvement in DLRM on 32 A100 GPUs on the Theta-GPU HPC system. We believe that MCR-DL will pave the way for designing and implementing future DL communication enhancements and distributed DL frameworks.

REFERENCES

- [1] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, “Zero-offload: Democratizing billion-scale model training,” 2021.
- [2] M. Shoenybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” *CoRR*, vol. abs/1909.08053, 2019. [Online]. Available: <http://arxiv.org/abs/1909.08053>
- [3] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, “Deep learning recommendation model for personalization and recommendation systems,” *arXiv preprint arXiv:1906.00091*, 2019.
- [4] Y. J. Kim, A. A. Awan, A. Muzio, A. F. C. Salinas, L. Lu, A. Hendy, S. Rajbhandari, Y. He, and H. H. Awadalla, “Scalable and efficient moe training for multitask multilingual models,” *arXiv preprint arXiv:2109.10465*, 2021.
- [5] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, “DeepSpeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale,” *arXiv preprint arXiv:2201.05596*, 2022.
- [6] A. Jain, A. A. Awan, A. M. Aljuhani, J. M. Hashmi, Q. G. Anthony, H. Subramoni, D. K. Panda, R. Machiraju, and A. Parwani, “Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’20. IEEE Press, 2020.
- [7] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoenybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, “Using deepspeed and megatron to train megatron-turing nlG 530b, a large-scale generative language model,” 2022.
- [8] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, “Opt: Open pre-trained transformer language models,” 2022. [Online]. Available: <https://arxiv.org/abs/2205.01068>
- [9] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park *et al.*, “Software-hardware co-design for fast and scalable training of deep learning recommendation models,” *arXiv preprint arXiv:2104.05158*, 2021.
- [10] 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>
- [11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “Zero: Memory optimizations toward training trillion parameter models,” 2020.
- [12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, *DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters*. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: <https://doi.org/10.1145/3394486.3406703>
- [13] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, “Gshard: Scaling giant models with conditional computation and automatic sharding,” 2020.
- [14] NVIDIA, “NVIDIA Collective Communications Library (NCCL),” 2016, Accessed: March 7, 2023. [Online]. Available: <https://developer.nvidia.com/nccl>
- [15] The Open MPI Development Team, “Open MPI : Open Source High Performance Computing,” <http://www.open-mpi.org>, 2004, [Online; accessed March 7, 2023].
- [16] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, <https://mvapich.cse.ohio-state.edu/>, 2001, [Online; accessed March 7, 2023].
- [17] L. Dalcin and Y.-L. L. Fang, “mpi4py: Status update after 12 years of development,” *Computing in Science Engineering*, vol. 23, no. 4, pp. 47–54, 2021.
- [18] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, “Pytorch distributed: Experiences on accelerating data parallel training,” *CoRR*, vol. abs/2006.15704, 2020. [Online]. Available: <https://arxiv.org/abs/2006.15704>
- [19] A. Sergeev and M. Del Balso, “Horovod: Fast and Easy Distributed Deep Learning in TensorFlow,” *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: <http://arxiv.org/abs/1802.05799>
- [20] J. Jose, M. Luo, S. Sur, and D. K. Panda, “Unifying upc and mpi runtimes: Experience with mvapich,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS ’10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: <https://doi.org/10.1145/2020373.2020378>
- [21] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, “Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems,” in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 1–13.
- [22] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. S. Pinto, D. Keyzers, and N. Houlsby, “Scaling vision with sparse mixture of experts,” *CoRR*, vol. abs/2106.05974, 2021. [Online]. Available: <https://arxiv.org/abs/2106.05974>
- [23] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. V. *et al* B. A. Hechtman.
- [24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic Differentiation in PyTorch,” 2017.
- [25] IBM, “IBM Spectrum MPI: Accelerating high-performance application parallelization,” <https://www.ibm.com/us-en/marketplace/spectrum-mpi>, 2018, Accessed: March 7, 2023.
- [26] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, “Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences,” in *International Workshop on OpenPOWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*, 2018.
- [27] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, “Synthesizing optimal collective algorithms,” *CoRR*, vol. abs/2008.08708, 2020. [Online]. Available: <https://arxiv.org/abs/2008.08708>
- [28] M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. L. *et al* S. Chen, H. Akin, M. Baines, L. Martin, X. Zhou, P. S. Koura, B. O’Horo, J. Wang, L. Zettlemoyer, M. T. Diab, Z. Kozareva, and V. Stoyanov.
- [29] H. Zhou, K. Raffanetti, Y. Guo, and R. Thakur, “MPIX stream: An explicit solution to hybrid MPIx programming,” in *EuroMPI/USA’22: 29th European MPI Users’ Group Meeting*. ACM, sep 2022. [Online]. Available: <https://doi.org/10.1145/2F3555819.3555820>
- [30] P. Lindstrom, “Fixed-rate compressed floating-point arrays,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.
- [31] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, “Mpi collective algorithm selection and quadtree encoding,” *Parallel Computing*, vol. 33, no. 9, pp. 613–623, 2007.
- [32] A. Faraj, X. Yuan, and D. Lowenthal, “Star-mpi: self tuned adaptive routines for mpi collective operations,” in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 199–208.
- [33] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, “Predicting mpi collective communication performance using machine learning,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 259–269.
- [34] Argonne National Laboratory, “Theta/ThetaGPU Machine Overview,” <https://www.alcf.anl.gov/support-center/theta-thetagpu-overview>, 2021, Accessed: March 7, 2023.
- [35] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, “The pile: An 800gb dataset of diverse text for language modeling,” *CoRR*, vol. abs/2101.00027, 2021. [Online]. Available: <https://arxiv.org/abs/2101.00027>
- [36] A. A. Awan, A. Jain, C.-H. Chu, H. Subramoni, and D. Panda, “Communication Profiling and Characterization of Deep Learning Workloads on Clusters with High-Performance Interconnects,” in *Hot Interconnects 26 (HotI ’19)*, August 2019.
- [37] N. Senthil Kumar, “Designing optimized mpi+nccl hybrid collective communication routines for dense many-gpu clusters,” Master’s thesis, 2021. [Online]. Available: http://rave.ohiolink.edu/etdc/view?acc_num=osu1619132252608831