ON OPTIMIZING THE COMMUNICATION OF MODEL PARALLELISM

Yonghao Zhuang [*1] **Hexu Zhao** [*2] **Lianmin Zheng** [3] **Zhuohan Li** [3]
Eric P. Xing [1 4 5] **Qirong Ho** [4 5] **Joseph E. Gonzalez** [3] **Ion Stoica** [3] **Hao Zhang** [3]

ABSTRACT

We study a novel and important communication pattern in model-parallel deep learning (DL) at scale, which we call cross-mesh resharding. This pattern emerges when the two paradigms of model parallelism – intra-operator and inter-operator parallelism – are combined to support large models on large clusters. In cross-mesh resharding, a sharded tensor needs to be sent from a source device mesh to a destination device mesh, on which the tensor may be distributed with the same or different layouts. We formalize this as a many-to-many multicast communication problem, and show that existing approaches either are sub-optimal or do not generalize to different network topologies or tensor layouts, which result from different model architectures and parallelism strategies. We then propose two contributions to address cross-mesh resharding: an efficient broadcast-based communication system, and an "overlapping-friendly" pipeline schedule. On microbenchmarks, our overall system outperforms existing ones by up to 10x across various tensor and mesh layouts. On end-to-end multi-node training of two models, GPT and U-Transformer, we improve throughput by 10% and 50%, respectively.

1 INTRODUCTION

Model-parallel distributed training and inference using GPU or TPU clusters have been key drivers for many recent advances (Brown et al., 2020; Shoeybi et al., 2019) in deep learning (DL). Concurrent model parallelism approaches can be roughly classified into two paradigms: intra-op parallelism that partitions individual layers or computational operators of the model (Xu et al., 2021; Lepikhin et al., 2020) and inter-op parallelism which partitions the computational graphs (Huang et al., 2019; Narayanan et al., 2021b).

Core to the performance of these model parallelism approaches is the need to communicate partial results between parallel devices, which are responsible for different parts of the model computations. Because there are many possible parallelism strategies, there is a correspondingly wide variety of communication patterns that results from said strategies. For example, in intra-op parallelism, communication is required to transform a tensor (Xu et al., 2021; Wang et al., 2019) (which is sharded over a group of devices called a *device mesh*) between two distributed tensor layouts (Figure 1b); whereas, in inter-op parallelism, communication is required to exchange a full tensor between a pair of devices (Figure 1c). Both intra-op and inter-op communication can be easily implemented using, respectively, existing collective and point-to-point (P2P) communication primitives, as previous research (Zheng et al., 2022; Xu et al., 2021) has revealed.

In practice however, neither intra-op nor inter-op parallelism alone suffices to scale out to models like GPT-3 (Brown et al., 2020; Narayanan et al., 2021b). Instead, they must be combined and lead to a new and composite communication pattern, called *cross-mesh resharding*, which requires (1) exchanging a tensor between two different device meshes, and simultaneously (2) changing its layout (see Figure 1d and §2.2). Since the communication happens from one group of devices to the other group, it cannot be implemented by directly using existing collective communication primitives. At the same time, because the tensor may have a different layout on the destination mesh, the communication also cannot be directly addressed by P2P communication primitives. Several previous works (Narayanan et al., 2021b; Zheng et al., 2022) have proposed specialized solutions to this problem, but only for transformer architectures under a fixed parallelism plan. As we will show later, none of them are optimal or general – for example, when they are applied to emerging models such as U-Transformer (Petit et al., 2021) – the backbone of diffusion models (Rombach et al., 2022) – they cause substantial slowdowns in communication.

In this paper, we ask: given an arbitrary model and its

*Yonghao and Hexu contributed equally. Part of this work was done when Hexu was intern at MBZUAI [1]Carnegie Mellon University [2]Tsinghua University [3]University of California, Berkeley [4]MBZUAI [5]Petuum Inc.. Correspondence to: Yonghao Zhuang <yhzhuang@cmu.edu>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

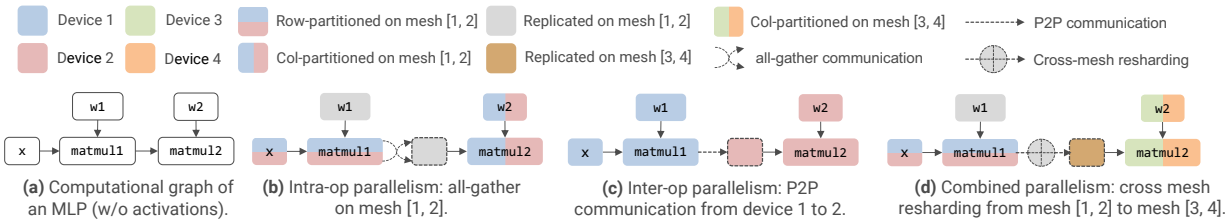


Figure 1. Model parallelism applies to an MLP, where each node represents an operator and its output tensor, and solid arrows indicate data flowing directions. The node with dotted boundaries in each of (b)-(d) shows the required input layout to `matmul2`. When combining two parallelisms in (d), cross-mesh resharding emerges for both exchanging tensors and converting their layouts between two meshes.

composite inter-op and intra-op parallelism plan, what is the best way to perform cross-mesh resharding? We formalize this as a many-to-many multicast communication problem between two device meshes, and reveal several unique challenges in the context of model-parallel DL:

Complex communication requirements. The message transferred in cross-mesh resharding is a distributed tensor sharded on two separate meshes. They may exhibit distinct layouts on the source and destination mesh (which consist of different, non-overlapping, sets of devices). For example, a slice of the tensor might be *replicated* over the source devices, but devices in the destination mesh may be expecting different parts of that same slice – in other words, the destination mesh requires the slice to be *partitioned*. Fulfilling such communication requirements while minimizing communication costs requires careful choice of communication primitives, sender and receiver devices, and communication routes.

Heterogeneous networking. A deep learning cluster contains heterogeneous networking hardware: NVLink, PICe, NIC, InfiniBand, etc.; this results in uneven bandwidth when devices communicate within and across meshes. Thus, in a model-parallel job, we need to carefully consider how to assign communication tasks, in order to make the best use of uneven bandwidth whilst balancing network traffic.

Dependency with computation schedules. An entire model-parallel job contains many cross-mesh resharding tasks, which are triggered whenever one mesh's computation depends on another mesh's results. Such computational dependencies are fully defined by the model computation and parallelism plan. To improve the model's overall performance, we need to optimally schedule and overlap these computation and communication tasks.

To address these challenges, we propose new solutions to (1) optimize a single arbitrary cross-mesh resharding task, and (2) collectively optimize all cross-mesh resharding tasks in a model-parallel job. Our contributions are summarized as:

★ We formalize the cross-mesh resharding problem in model-parallel DL jobs, and reveal its characteristics.

★ We propose a general and principled solution to optimize for a single cross-mesh resharding task. Specifically, we first show that communication primitives in existing solutions are suboptimal, then we propose *broadcast* as an alternative and show it is provably optimal. Based on this, we develop new algorithms to schedule and load-balance multiple broadcast primitives within a single cross-mesh resharding task.

★ We propose a new pipelining schedule that overlaps multiple cross-mesh resharding communication tasks with the pipeline-parallel computation in a model-parallel job.

★ We implement proposed techniques as a communication library, AlpaComm , and show it performs up to 10x faster than existing solutions on various microbenchmarks.

★ We integrate the library with Alpa, a state-of-the-art model-parallel system. Our techniques improve the end-to-end multi-node training throughput of two models, GPT and U-Transformer, by 10% and 50%, respectively.

2 Background

In this section, we introduce the basic communication characteristics in model-parallel DL. We then describe a new communication pattern, called *cross-mesh resharding*, that has emerged in large model-parallel workloads.

2.1 Model Parallelism

When the model is large and cannot fit into the memory of a single compute device, model parallelism is an essential step to parallelize the model computation over multiple devices. Since DL computation is often defined as a computational graph of tensors and computational operators, model parallelism is equivalent with partitioning the computational graph and placing different parts on parallel devices. Depending on how the graph is partitioned, model parallelism methods can be categorized into two classes: *intra-operator parallelism* and *inter-operator parallelism*, with different communication requirements.

Intra-op parallelism refers to methods that shard or replicate the operators and its input and output tensors along some tensor axes, and assign different regions of the operator computation to parallel devices, as Figure 1(b)

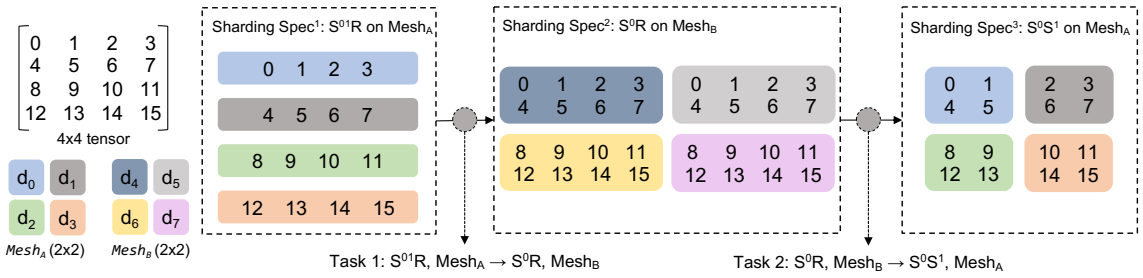


Figure 2. Two examples of cross-mesh resharding. Given a 4×4 matrix and two 2×2 meshes $Mesh_A$ and $Mesh_B$, each dotted box shows a sharding spec and its resulting tensor layout on each device of the mesh (note that $Mesh_A$ is used twice: in the first and third dotted boxes). Two cross-mesh resharding tasks are formed when communicating between Specs 1 and 2 and between Spec 2 and 3.

shows. In practice, intra-op parallelism is realized in an SPMD fashion (Lepikhin et al., 2020; Xu et al., 2021): the tensors and computation are *evenly* sharded and dispatched to all participating devices. When tensors are sharded along different tensor axes, they may be stored on multiple devices with different *distributed layouts*. When an operator is parallelized using a parallel algorithm, it might require its input tensors to follow a specific layout and produce output tensors with another layout. If an input tensor's layout disagrees with the required input layout of the operator, a *layout conversion* (also called *resharding* (Xu et al., 2021)) is needed. In intra-op parallelism, this layout conversion is easily facilitated by collective communication operations, such as *all-reduce*, *all-gather*, *all-to-all*, among all participant devices (see Figure 1(b)).

Inter-op parallelism partitions the graph (instead of individual operators or tensors) and places each resulting subgraph, called *a stage*, onto a device. If an operator's input tensor is generated by another operator on a different device, a point-to-point (P2P) communication is required to send the tensor from the source device to the destination device (see Figure 1(c)). Pipeline parallelism (Huang et al., 2019; Narayanan et al., 2021a) is a special case of inter-op parallelism where all devices are arranged into a pipeline and saturated by letting devices from different pipeline stages compute simultaneously in "assembly-line" fashion; P2P communications are used between stages. Due to data dependencies between partitions, some are idle while waiting its input or the last partition to finish, resulting in bubbles. Bubbles increase if partition is not perfectly even.

Combining Intra- and Inter-op parallelism. Neither intra-op parallelism nor inter-op parallelism alone suffices to train large models. Intra-op parallelism always has large communication volume, and cannot scale to multiple nodes, while inter-op parallelism suffers from bubbles. In practice, they must be combined to support large models like GPT-3 (Brown et al., 2020). This combined strategy is implemented in many model-parallel systems (Rasley et al., 2020; Zheng et al., 2022; Xu et al., 2021) by first partitioning the computational graph using inter-op parallelism then sharding each stage using intra-op parallelism, shown in Figure 1(d). Specifically, the graph is first partitioned into multiple stages. Each stage is assigned to a group of devices, referred to as a *device mesh*, sliced from the cluster. Operators and tensors of a stage are parallelized over that stage's assigned mesh following a chosen intra-op parallelism plan; collective communication happens only across devices within each mesh. At the boundary of any two adjacent stages, communication is required to exchange tensors between their meshes. Unlike inter-op parallelism, the tensor might have been sharded with different layouts on the source and destination meshes – in which case, communication involves not only transferring the tensor, but also performing tensor layout conversion between the source and destination groups of devices. We call this communication pattern *cross-mesh resharding*, which is the focus of this paper.

In practice, model-parallel systems may sub-optimally map the cross-mesh resharding communication to slower interconnects in a deep learning cluster, such as the Ethernet connection between different nodes/hosts, which has limited communication bandwidth. Such under-optimized mappings can significantly hurt overall training or inference performance (see §5). Next, we formally define cross-mesh resharding and discuss optimization opportunities.

2.2 Cross-mesh Resharding

A model-parallel job may involve many cross-mesh resharding communications. We define a *cross-mesh resharding task* as the communication needed to send a single tensor from a source mesh to a destination mesh while meeting the layout requirements. To analyze it, we first formalize device mesh and tensor layout.

Device mesh. We follow the definitions of mesh and tensor layout in GSPMD (Xu et al., 2021)[1]: a mesh is an n-dimensional array of identical processors. We express a 2D view of a mesh $Mesh_A$ as (m_1, m_2), where $m_1 \times m_2 =$

[1]Some model-parallel systems define them differently, which can be equivalently converted to definitions used in this paper.

$|Mesh_A|$. For example, a cluster of 2 nodes with 2 GPUs each can be represented as a $(2, 2)$ mesh $[[0, 1], [2, 3]]$, where each integer is a device index, or as a $(1, 4)$ mesh $[[0, 1, 2, 3]]$.

Sharding spec. With this definition, we can describe the layout of a sharded N-dimensional tensor D over $Mesh_A$ using a *sharding spec*, notated as an N-element string: $\mathcal{X} = X_0^{d_0} X_1^{d_1} \dots X_{N-1}^{d_{N-1}}, X_i \in \{S, R\}, d_i \in \{0, 1, 01\}, 0 \leq i \leq N-1$, where S and R stands for *sharded* or *replicated*, respectively. $X_i = S$ means the i-th dimension of D is sharded; $X_i = R$ means it is replicated. When a tensor dimension is sharded, a superscript d_i is added and represents which mesh dimensions the sharding are mapped to, e.g., $X_i^{d_i} = S^0$ means the i-th dimension of D is sharded along the first dimension of $Mesh_A$; $X_i^{d_i} = S^{01}$ means the i-th dim is sharded along both dimensions of $Mesh_A$; $X_i^{d_i} = R$ means that dimension is replicated. Figure 2 illustrates three sharding specs and resulting tensor layouts.

Data slice. Due to the sharding or replication, each device in $Mesh_A$ might hold a *data slice* or *replica* of D. For examples, in the 1st sharding spec of Figure 2, each device in $Mesh_A$ holds a unique 4×1 data slice of D; In the 2nd sharding spec, D is sharded into two unique 2×4 data slices on $Mesh_B$; devices 4 (blue) and 5 (red) hold a replica of the top unique slice (containing matrix elements 0 through 7).

Cross-mesh resharding. With these notations, we can see that a cross-mesh resharding is a *many-to-many multicast communication task* between two non-overlapping device meshes $Mesh_A$, $Mesh_B$ with $Mesh_A \cap Mesh_B = \varnothing$. The task aims to send D, sharded on the $Mesh_A$ under sharding spec $\mathcal{X}_A$, to $Mesh_B$, where sharding spec of D becomes $\mathcal{X}_B$. Figure 2 shows two examples of cross-mesh resharding.

It is obvious to see that, regardless of $\mathcal{X}_A$ and $\mathcal{X}_B$, the size of messages transferred between two meshes is lower bound by the size of D. Hence, we can break down a cross-mesh resharding into a set of *unit communication tasks*, and each corresponds to a *unique* data slice DS_i of D on the source $Mesh_A$, and is responsible to send DS_i from $Mesh_A$ to a subset of devices in $Mesh_B$. For instance, Task 1 in Figure 2 has four unit tasks. The first unit task sends slice $[0, 1, 2, 3]$ to device 4 and 5. Task 2 has two unit tasks; its first unit task sends the first 2×4 slice to device 0 and 1 in $Mesh_A$. In Appendix.B, we list all unit tasks of the two cross-mesh resharding in Figure 2.

Since multiple devices in the source mesh may hold a replica of DS_i (due to replication) and multiple devices in the destination mesh may require DS_i, there exist many possible communication routes. Each route may result in different total sizes of messages transferred across the network and traffic on network switches. Moreover, although we have used 2D tensor for explanations, a real model-parallel job works on N-dimensional arrays ($N \geq 3$). In §3, we develop general solutions to minimize the size of message transferred while achieve good load balance for a single cross mesh resharding task.

Globally, a model-parallel job has many cross-mesh resharding tasks for each tensor exchanged at the boundary of two pipeline stages. Each task is repeatedly performed at every forward and backward pass following the computational dependency of stages. This exposes overlapping opportunities to hide the communication time by scheduling multiple cross-mesh resharding tasks and computation tasks together, as discussed in §4.

3 Optimizations for a Single Cross-Mesh Resharding Task

In §2.2, we show that a cross-mesh resharding problem can be decomposed into many unit communication tasks and each unit task is responsible for one data slice. Our goal is to complete all tasks as fast as possible. To achieve that, we frame the original problem as a two-level optimization:

- **Optimizing individual unit communication tasks.** We analyze different methods and propose to use broadcast for this part and show that a broadcast-based strategy could give optimal performance.
- **Load balance and schedule for all unit tasks in a cross-mesh resharding.** We formulate the load balance and scheduling problem of multiple unit communication tasks and design two algorithms to solve this problem. Our algorithms can give optimal solutions in most cases.

Before going into details of the two levels, we first consider the typical ML cluster setting with the following properties:

- **Fast intra-node and slow inter-node communication.** For example, in a typical GPU cluster, GPUs within a node have fast NVLink interconnect while GPUs across nodes need to communicate via slower Ethernet or Infiniband connections.
- **Fully-connected topology between nodes.** We assume that the network bandwidth between a pair of nodes will not be affected by communication between other nodes. In addition, we assume any pair has the same bandwidth. This assumption holds for most clouds and recent datacenters (Andreyev, 2014).
- **Communication bottleneck at hosts.** The bandwidth of a device (e.g., GPU) is often much higher than its host. When multiple devices in a single host send data to another host, they will compete for

the communication bandwidth at the host's network interface. In this work, we assume each host has only one NIC, which is the common setup in public clouds.

- **Separate sending/receiving bandwidth (full duplex).** We assume each device has separate sending and receiving bandwidth. One device can receive data at full bandwidth while also sending at full bandwidth. This applies to both NVLink and inter-host networks.

3.1 Optimizing individual unit communication task

We use DS_i to be the data slice corresponding to the i-th unit communication task. DS_i has replicas in device set $N_i \subseteq Mesh_A$, and we want to send DS_i to devices in set $M_i \subseteq Mesh_B$. In this subsection, we propose several communication strategies to achieve this goal.

We first point out that a unit task cannot be directly handled by collective primitives. In a unit task, potential senders does not overlap with receivers, while collective primitives focus on cases where senders are also receivers.

From the properties of ML clusters described above, we can see that a good communication strategy will (1) minimize inter-host communication volume and (2) maximize the total sending and receiving bandwidth utilization of all participating devices.

In the following analysis, we assume we are sending a fixed-size object that takes duration t to send from a group of nodes to another group and ignore the time we spent on intra-node communication. For a communication strategy, we denote $T(A, B)$ to be the time to send the fixed-size object from one device to $A \times B$ devices with A nodes and B devices on each node. Next, we introduce several strategies, from the most naive to the most effective.

Send/recv. We first consider the case with only one host as the potential sender. The simplest strategy is to let the only sender send the data slice DS_i to each receiver device in M_i one by one. In this case, the total communication latency is $T^{sr}(A, B) = ABt$, proportional to the number of total devices, as in Figure 3a. If some receiver devices are co-located on the same node, this strategy introduces redundant inter-node communication, since the other devices can receive DS_i from peers on the same node that have already received the data slice. When there are multiple potential senders, we can distribute workloads to different senders and speed up the overall performance. We discuss such load balance in the next subsection.

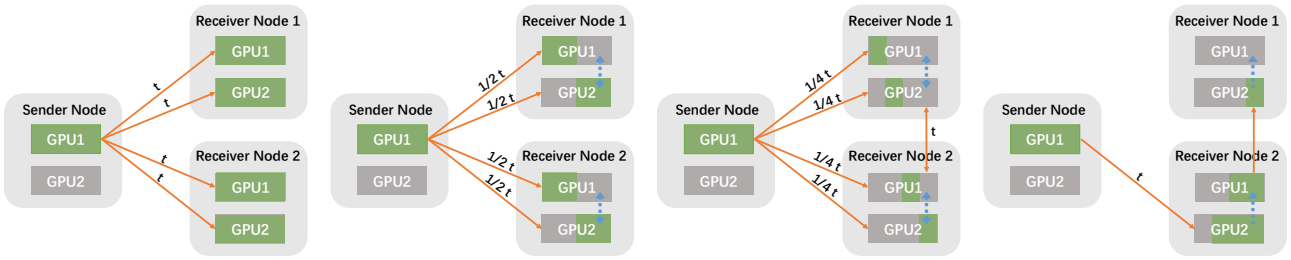
Send/recv with local allgather. One straight-forward improvement to the naive send/recv strategy is to let each receiver node only receive a copy of DS_i, and then use the fast intra-node connection to transfer the slices among all devices on this node. To achieve this, we split DS_i into B parts, and send each part to one of the B devices on a specific receiver node. Then the receiver devices on the same node assemble a full copy of DS_i via an *all-gather* collective communication primitive. Therefore, the overall latency for a sender is $T^{srla}(A, B) = At$, as in Figure 3b. This strategy improves the naive send/recv by reducing inter-host communication volume, so the latency is only proportional to the total number of hosts. Similar to send/recv, we can distribute the workload with multiple sender candidates.

Send/recv with global allgather. We further slice DS_i into more fine-grained chunks. More specifically, we can split DS_i over all $A \times B$ slices, and send one slice to each of the $A \times B$ devices. The latency of this step is t since in total the sender only sends one copy of DS_i. These devices then perform a global all-gather to collect all slices for each device, whose latency is also t with a typical ring all-gather algorithm (NVIDIA, 2018). Therefore, the ideal overall latency $T^{srga}(x, y) = 2t$, which does not grow with the number of nodes or devices, as in Figure 3b.

Broadcast based resharding. Both the previous two allgather-based strategies can be decomposed into two stages: the first stage where a sender sends the data slice DS_i to receiver devices, and the second where receiver devices exchange different partitions of the data slice. We observe that we can further optimize this by overlapping these two stages. More specifically, when a receiver starts to receive data from the sender, it can act as another sender to send data to the remaining receiver devices, as illustrated in Figure 3d. Eventually, this strategy can be viewed as a broadcast from the sender device to all receiver devices. Suppose we split the data slice DS_i into K partitions. Each device sends the partition to the next device after it received one full partition. The time to send each partition is t/K and the overall latency is $T^{bc}(A, B) = \frac{t}{K} \cdot (K + A) = t + \frac{At}{K}$, as in Figure 3d. When we pick K to be a relatively large value compared to A (e.g., $K \approx 100$ in our experiments), the overall latency will be around t, which is optimal among all proposed strategies and has already reached the upper bound. t is the communication upper bound because each receiver node has only one NIC, and needs at least one replica, which takes t to receive. The broadcast only involves one sender device from N_i and all receiver devices M_i. This is different from previous send/recv-based methods which have to utilize more senders. For the case where receivers have multiple NICs, a unit task can be optimized by dividing it into multiple tasks. We leave this part to our future work.

3.2 Load balance and schedule for multiple unit communication tasks

Each cross-mesh resharding task includes multiple unit communication tasks. These tasks might overlap on both sender and receiver devices and affect each other. Therefore,



(a) Send/recv. (b) Send/recv + local all-gather. (c) Send/recv + global all-gather. (d) Broadcast.

Figure 3. Communication strategies for an individual communication task. The orange arrows represent cross-host communication and the numbers on it represent the latency of the communication. The blue arrows represent intra-node communication.

to optimize the overall completion time for cross-mesh resharding, we treat this problem as a *load balancing and scheduling* problem. Specifically, we need to (1) balance the loads by evenly distributing the communication workloads across sender devices and inter-host communication links to avoid congestion and stragglers; (2) schedule the order of different tasks assigned to specific devices to minimize the wait due to unavailable sender/receiver.

Since hosts are the main bottleneck for communication, we perform load balancing and scheduling at host level, rather than individual devices. Suppose a unit communication task has replicas DS_i in the host set $n_i \subseteq Mesh_A$ and we want to send DS_i to the hosts in set $m_i \subseteq Mesh_B$. Our first goal is to pick a host $n_{i*} \in n_i$ to send the data to all the receiver hosts in m_i to balance the loads across different devices. Once we picked the sender host n_{i*} for all tasks, we also need to schedule the execution order of different tasks. Note that all the hosts participate in the i-th task need to send or receive data with in a shared period of time and different tasks' execution should not overlap. Suppose the duration of the i-th task to be T_i and denote the execution starting time to S_i, we can formulate the load balancing and scheduling problem as the following optimization problem:

$$\min_{S, n_*} \max_i S_i + T_i; \tag{1}$$

$$\text{s.t. } n_{i*} \in n_i; \tag{2}$$

$$(S_i, S_i + T_i) \cap (S_j, S_j + T_j) = \emptyset,$$

$$\forall n_{i*} = n_{j*} \text{ or } m_i \cap m_j \neq \emptyset. \tag{3}$$

The optimization objective (Eq.1) minimizes the completion time of the last unit communication task. The two constraints make sure the sender of each task has the correct data slice (Eq.2) and the tasks that share the same sender or receiver devices are not overlap with each other (Eq.3).With multiple NICs on a node, the formulation still applies by making n_i, n_{i*} and m_i represent sets of NICs.

Next, we introduce several load balancing and scheduling algorithms and analyze their properties.

Naive algorithm. The most straight-forward algorithm is to assign each task to be sent by the first (i.e., lowest-indexed) device in the sender's host. We schedule all the tasks following an arbitrary global order. We serve this algorithm as our baseline to compare with our later algorithms.

Load balance only. We start from the load balance across multiple sender hosts. In this case, the original problem is simplified as

$$\min_{n_*} \max_{k \in Mesh_A} \sum_{i: n_{i*} = k} T_i. \tag{4}$$

We note that this minimax optimization problem can be solved optimally by a classical greedy algorithm: we can sort all tasks in descending order of duration T_i. Then, iterate through the sorted tasks and assign each task to the sender host with the currently-lightest workload.

Depth First Search (DFS) with Pruning. All previous algorithm does not take the scheduling of different tasks into account. We first note that scheduling in the original optimization problem (i.e., deciding S in Eq.1-3) can be simplified to the following problem: For each host, assign an execution order to all of the send/receive tasks on that host. Then, the starting time of each task S_i can be set to the earliest time at which all preceding tasks have finished on the sender host n_{i*} and the receiver hosts in m_i.

For the complex space of sender assignments and task orders, we first propose a search based algorithm: We construct a depth-first search tree over the two types of decisions. When the algorithm reaches a leaf node with a complete sender assignment and task schedule, we compute the completion time in Eq.1. During the search process, we prune the branches whose execution time lower bound (measured by the sum of task durations on a single device as in Eq.4) exceed the current best completion time.

This algorithm has exponential computation complexity, so in practice, we impose a time budget and return the best solution found within the time budget. However, this algorithm fails to produce an efficient schedule within the time budget when there are > 20 unit communication tasks. This motivates the next algorithm.

Greedy Search with Randomization. We propose an iterative greedy algorithm that at each iteration, selects a set

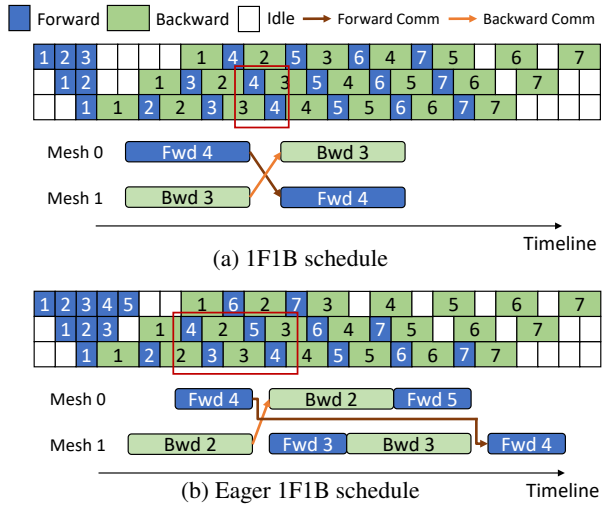


Figure 4. Timelines of 1F1B and eager-1F1B schedule. Each row is a pipeline stage. Numbers indicate micro-batch indices. Micro-batch 4's communication from stage 2 to stage 3 is magnified.

Table 1. Size of parameters, optimizer states, and activations per GPU for a GPT-3 layer in mixed precision training. The sequence length $S = 1024$, hidden size $H = 12288$, per-GPU micro-batch size $B = 2$, and tensor model parallel degree TMP $= 8$.

	expression	value
#parameter	$12H^2$/TMP	216M
#optimizer state parameters	$24H^2$/TMP	432M
#activation elements	BSH	24M
Memory of weights and optimizer	$168H^2$/TMP	2.95GB
Memory of activation	$2BSH$	48MB

of non-overlapping tasks which has the maximum number of devices. To find such a set, we use a randomized algorithm: First, we generate a random ordering over all tasks to be scheduled. Then, we iterate through each task with the generated order and select the tasks that do not overlap with previously selected ones. We repeat this process multiple times and select the largest candidate set for the iteration.

For cross-mesh resharding tasks, since we communicate tensors evenly sharded, many unit communication tasks are identical, in terms of the number of devices involved as well as the amount of transferred data and the tasks are distributed uniformly across different devices. Therefore, the greedy randomized algorithm can find good and even optimal solutions in short amount of time.

4 Overlapping-friendly Pipeline Schedule

The previous section addresses how to optimize a single cross-mesh resharding task. However, this is not enough for performing pipeline parallelism which requires scheduling multiple compute and communication tasks from multiple micro-batches and stages. To study the overhead of cross-mesh communication, we implement a hypothetical upper bound called "Signal Send/Recv". This method only communicates one byte for all cross-mesh resharding tasks, so it keeps the data dependency of the compute tasks but removes almost all communication costs. We find the training throughput can be 1.1x to 1.5x faster with "Signal Send/Recv" (§7) . This means there is still an opportunity to reduce communication costs.

Our key observation is that the existing synchronous pipeline schedules such as 1F1B (Narayanan et al., 2021b) and GPipe (Huang et al., 2019) are not overlapping-friendly. Fig. 4(a) shows the timeline of 1F1B schedules. The following three tasks of the same micro-batch are scheduled contiguously: the computation of i-th stage, the communication between i-th stage and $(i+1)$-th stage, the computation of $(i+1)$-th stage. Due to this strict data dependency, the communication cannot be overlapped.

To solve the problem, we develop a novel and more overlap-friendly schedule called eager-1F1B. The main idea is to run computations eagerly to create opportunities for overlapping. Fig. 4(b) shows the timeline of the new schedule.

The eager-1F1B schedule is based on 1F1B but shifts forward computation tasks to several time steps ahead. During the warm-up phase of the original 1F1B, stage i runs forward computations for the first $(\#stages - i + 1)$ batches. Stage i then stops for a while and enters the steady one-forward-one-backward phase after it is able to run the first backward computation. During the warm-up phase of the eager-1F1B schedule, stage i runs forward computations for more micro-batches. Stage i runs $(2\times(\#stages-i)+1)$ forward computations and then enters the steady phase.

When there is no communication cost, the latencies of these two schedules are the same. However, when the communication cost is not negligible, 1F1B cannot hide the cost due to the strict data dependency. On the contrary, the eager-1F1B schedule inserts computation tasks from other micro-batches between two dependent tasks, making it possible to hide the communication costs. One example of this overlapping is shown as the magnified plot in Fig. 4.

The downside of the eager-1F1B schedule is a slight memory usage increase because each GPU stores temporary activation tensors for more micro-batches. However, this increment is very small in common cases. We list the size of parameters and activations with a common parallel configuration in Table 1. The new schedule increases per-GPU memory usage by at most $\#stages \times size_{activation}$, which is a lot smaller than the parameter size.

The eager-1F1B schedule can overlap the communication of the forward part but does not help the backward part. To overlap backward communication, we divide a backward computation into two parts: computing gradients

of activations and gradients of weights. The first does not rely on the second, and cross-mesh communication only uses the output of the first. Hence, we delay the second part so that it overlaps with the communication. We call this backward weight delaying. Similarly, this technique slightly increases the peak memory usage, so we use a simple cost model to estimate the compute and communication time and delay the least to cover all communications.

We find that in practice, even without backward weight delaying, the performance of the eager-1F1B schedule is very close to the "Signal Send/Recv". This is because later pipeline stages store fewer activations and have less memory pressure. As a result, they use less rematerialization and are slightly faster, which provides time for communication.

5 Evaluation

The optimizations in this paper are implemented with about 900 lines of C++ and 2.5k lines of python. We implement a standalone cross-mesh resharding communication library AlpaComm for GPU based on NCCL (NVIDIA, 2018) and integrate it into Alpa (Zheng et al., 2022), a state-of-the-art distributed training framework. We implement the eager-1F1B schedule and overlap optimizations on top of Alpa, which provides APIs to customize pipeline schedules and cross-pipeline-stage communication. Implementing in platforms like PyTorch is possible, but we believe it needs more effort.

We first evaluate the communication library on a set of microbenchmarks of cross-mesh resharding cases. We then evaluate all optimizations on end-to-end training of large models with billions of parameters, including a GPT-3 like language model (Brown et al., 2020) and a U-Net Transformer (Petit et al., 2021). We also isolate each part in our optimizations and perform ablation studies of them.

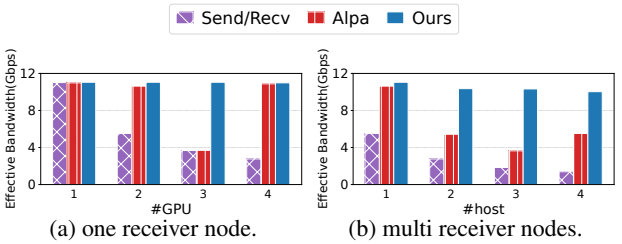
We run experiments on an AWS cluster where each node is a p3.8xlarge instance with 4 NVIDIA V100 (16GB) GPUs and 32 vCPUs. GPUs in a node are connected via NVLink and nodes are launched with 10Gbps cross-node bandwidth.

5.1 Microbenchmark

We first show the advantage of our broadcast-based resharding when sending the tensor from a single device to multiple devices. We then show the advantage of broadcast-based resharding with load balance when sending the tensor from multiple devices to multiple devices.

5.1.1 *Single device to multiple devices*

Experimental setup. In this setting, the sender mesh has only 1 GPU. We vary the number of GPUs in the receiver mesh. In the first group of benchmarks, the device mesh has



(a) one receiver node. (b) multi receiver nodes.

Figure 5. Single device to multiple devices microbenchmark result.

Table 2. Sharding specs of senders and receivers in multi-to-multi-device microbenchmark.

case	send spec	recv spec	send mesh shape	recv mesh shape
case1	S^0RR	S^0RR	(2,4)	(2,4)
case2	RRR	S^0RR	(2,4)	(2,4)
case3	RS^0R	S^0RR	(2,4)	(2,4)
case4	$RS^{01}R$	$S^{01}RR$	(2,4)	(2,4)
case5	S^1RR	S_0RR	(2,4)	(2,4)
case6	S^0RR	S^0RR	(2,4)	(3,4)
case7	S^1RR	RRR	(1,4)	(2,4)
case8	RRR	RRR	(2,3)	(3,2)
case9	RS^0R	RRS^0	(2,4)	(2,4)

1 node and the number of GPUs in this node varies from 1 to 4. In the second group of benchmarks, the number of GPUs per node is fixed at 2, but the number of nodes grows from 1 to 4. Both the sender and receiver use a fully replicated sharding spec. The message size keeps 1GB.

Baseline. We compare our work against two baselines: "Send/Recv" and "Alpa". "Send/Recv" only uses point-to-point send/recv communication primitives as in 3.1. "Alpa" is an all-gather based approach used in systems like Alpa and Megatron-LM, which can offload communications from slow cross-node connections to fast intra-node connections.

Results. As Fig.5 shows, when the receiver mesh has only one node, the latency of “Send/Recv" grows linearly with the number of GPUs in the mesh, while ours AlpaComm and Alpa only have less than 1% increase. The result is aligned with our analysis for broadcast and allgather based method(Alpa) in 3.1. However, when the number of nodes in the receiver mesh is greater than 1, the intra-mesh all-gather in Alpa has to use the inter-node connections as well, and hence the overhead of all-gather is no longer negligible. On the contrary, AlpaComm still shows almost no performance degradation. The sudden performance drop of Alpa when #gpu is 3 or #node is 3 is because Alpa cannot handle uneven partition, while ours AlpaComm efficiently handles tiling, padding, and pipelining with the broadcast-based approach.

5.1.2 *Multiple devices to multiple devices*

Experimental setup. Next, we move to more complicated cases where both the sender mesh and receiver mesh have multiple nodes. Table 2 lists the configurations of all cases. We use a tensor of shape (1024, 1024, 512) and pad it in case 6. We pick several representative sharding specs from

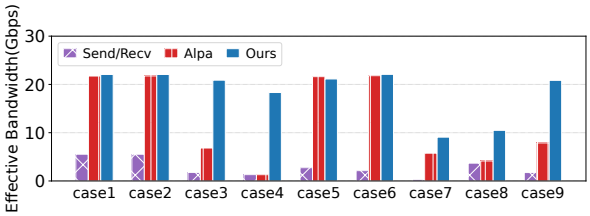


Figure 6. Multi-to-multi-device microbenchmark result.

Table 3. Models in end-to-end evaluation. Parallel config is a tuple of (data parallel degree, operator parallel degree, pipeline parallel degree).

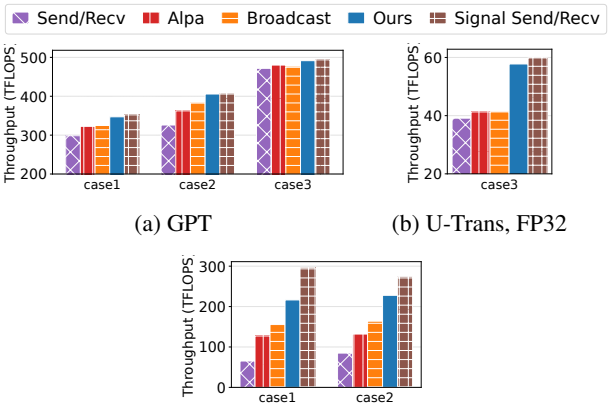
Model	Batch Size	#params	Precision	Parallel Config
GPT case1	1024	1.3B	FP16	(2, 2, 2)
GPT case1	1024	2.6B	FP16	(2, 2, 2)
GPT case2	1024	2.6B	FP16	(4, 1, 2)
U-Trans case1	2048	1B	FP16	(auto, auto, 2)
U-Trans case2	2048	2.1B	FP16	(auto, auto, 2)
U-Trans case3	2048	2.1B	FP32	(auto, auto, 2)

common deep learning workloads. For example, sharding spec S^0RR is the spec of activation tensors in transformer models when using combined data and operator parallelism; $S^{01}RR$ is the spec when using pure data parallelism; RRR is the spec when using pure operator parallelism. We use the same set of baselines as in §5.1.1. The baselines do load balancing with a greedy approach which picks the sender with the lowest load for the next data slice, as in 3.2.

Results. In case 1, 2, 5, and 6, AlpaComm and Alpa perform similarly. Both benefit from offloading communications to fast NVLinks. In case 7 and 8, Alpa's all-gather crosses nodes and is slow, but AlpaComm pipelines it with the inter-mesh communication, so it is up to 2.5x faster than Alpa. In case 3, 4, and 9, AlpaComm is 3x to 10x faster than Alpa respectively, because it reorders communications to utilize bandwidth of both two nodes of sender. In contrast, with Alpa's order, two sender nodes always share the same receiver, making one of them idle. This problem is more significant when the number of tiles is large, so AlpaComm shows an even better speedup in case 4 which schedules 64 unit tasks.

5.2 End-to-End Performance

Experiment setup. We evaluate our work on two models of which the configuration is listed in Table 3. GPT is a homogeneous transformer-based model built by stacking transformer layers with the same structure. At each pipeline stage, it sends the output activation tensor of the last transformer layer in the stage. The tensor is partitioned on data-parallelized devices and replicated on operator-parallelized devices. U-Transformer (Petit et al., 2021) is a U-shaped convolution neural network with long skip connections. It inserts attention layers after convolution layers. In the U-Net model, the i-th layer sends its output



(a) GPT (b) U-Trans, FP32

(c) U-Trans, FP16

Figure 7. End-to-end evaluation result.

activation to not only the next layer, but also the i-th layer from last. For the GPT model, we test two parallel configurations which combine data, operator and pipeline parallelism. For the U-Transformer model, we manually partition it into two stages and use Alpa to generate the intra-operator parallel strategy of each stage. We balance pipeline stages with respect to FLOPs.

Baselines. We compare our work against three baselines. The first two "Send/Recv" and "Alpa" are the same as that in §5.1.1. The third is "Broadcast", which uses broadcast-based resharding without overlapping introduced in §4. Therefore, "Broadcast" only optimizes a single resharding task, which mimics streaming optimization introduced in related work such as CoCoNet (Jangda et al., 2022b). We also add the performance of "Signal Send/Recv" as the upper bound, which is discussed in §4.

Results. Fig. 7 shows the evaluation results. The y-axis is the aggregated throughput of all GPUs in the cluster. We use TFLOPs as the throughput metric following the existing literature (Narayanan et al., 2021b).

On GPT models, both AlpaComm and Alpa are very close to the upper bound "Signal Send/Recv", because they can offload communication to high-bandwidth NVLink. AlpaComm shows a 1.1x speedup because of overlapping. On the U-Transformer model, the U-shaped skip connection makes cross-mesh communication a bottleneck. Our eager 1F1B schedule can effectively overlap the communication and achieve a 1.5x speedup compared to Alpa. AlpaComm reaches more than 75% of the upper bound in all cases.

The communication exhibits a sublinear increase in relation to the growth of models, whereas the computation linearly or even superlinearly increases. This disparity yields a relatively smaller proportion of communication, and thus enables better overlapping and brings AlpaComm closer to Signal Send/Recv. A similar effect can be observed when transitioning from FP16 to FP32 in U-Transformer.

Figure 8. Ablation study for load balance algorithm.

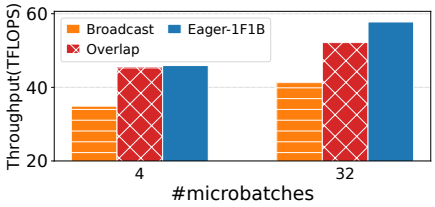


Figure 9. Ablation study for overlap-friendly schedule.

5.3 Ablation Study

In this section, we study the effectiveness of our load-balance algorithm and overlap-friendly schedule.

5.3.1 Load balance

Experiment setup. We use microbenchmarks for the ablation study of load balance. The setup shares §5.1.2's.

Baselines. We compare our load balance algorithm with two baselines. "Naive algorithm" just chooses the sender with lowest index as in §3.2 and sends to all receivers for each tile, and sends tiles by an arbitrary global order. "Load balance only" balances the size sent by each node and uses order generated by the greedy algorithm in §3.2. AlpaComm is the ensemble of DFS with pruning and the Greedy Search with Randomization. We run both algorithms and choose the better result as the schedule to launch broadcasts.

Results. There is only point-to-point communication in case 1 and 8, so all methods perform similarly. In case 2, "Naive algorithm" sends all tensors from the first node, leading to congestion. In case 8, there is only one broadcast to be launched and thus we have no opportunity to optimize order. In other cases, both "Naive algorithm" and "Load balance only" meet congestion, while AlpaComm can find the optimal order to avoid any send or receive node being idle.

5.3.2 Overlap

Experiment setup. We report the performance of our schedule and other variants on U-Transformer model. We use two batch sizes and keep the same microbatch size.

Baselines. We compare our work with two baselines. "Broadcast" uses broadcast-based resharding and load balance. "Overlap" overlaps pipeline communication and computation on top of Broadcast, but does not use the eager-1F1B schedule. Our work "Eager-1F1B" uses the eager-1F1B schedule on top of "Overlap".

Results. For a very small number of microbatches, the pipeline has no steady period, so "Overlap" is only 3% slower than AlpaComm . For a more typical case with a larger number of microbatches, "Overlap" is 30% faster than "Broadcast", and "Eager-1F1B" adds another 15%.

6 Related Work

Distributed machine learning The training of neural networks can be distributed with various parallel methods. Data parallelism distributes the training by partitioning the input data. Horovod (Sergeev & Del Balso, 2018) and PyTorchDDP (Li et al., 2020) are two widely adopted data-parallel systems. ZeRO (Rajbhandari et al., 2020; 2021) reduces the memory usage of data parallelism for training large models. On the other hand, model parallelism distributes the training by partitioning the model, which can be performed at either the inter-operator level or intra-operator level. Mesh-TensorFlow (Shazeer et al., 2018), GSPMD (Xu et al., 2021), and GPipe (Huang et al., 2019) provide annotation APIs for users to manually specify a parallel plan. Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021b; Korthikanti et al., 2022) and TeraPipe (Li et al., 2021) design specialized partition strategies for transformer models. Besides manually designing the parallel methods, Alpa (Zheng et al., 2022), Unity (Unger et al., 2022), FlexFlow (Jia et al., 2018), and AutoSync (Zhang et al., 2020) automatically searches for the best plan. The optimizations introduced in this paper can be easily integrated into the above systems to address the communication bottleneck in distributed training.

Collective communication Most communication in distributed training can be implemented by collective communication primitives. Efficient algorithms and implementations of these primitives are a long-standing research topic (Chan et al., 2007; NVIDIA, 2018). Earlier work solves the problem for regular topology (Scott, 1991; Barnett et al., 1994; Patarasuk & Yuan, 2009). Recently, more algorithms are proposed to exploit the hierarchy, heterogeneousness and dynamics (Cho et al., 2019; Wang et al., 2020; Luo et al., 2020; Zhuang et al., 2021). The optimal algorithm of these primitives on a specific topology can be synthesized using constraint solver, type-directed or syntax-guided approaches (Cai et al., 2021; Shah et al., 2021; Xie et al., 2022; Rink et al., 2021). On the other hand, domain-specific languages and compilers are built to allow manual optimization (Jangda et al., 2022a; Cowan et al., 2022). The collective communication works well for the single program, multiple data (SPMD) pattern. However, when doing model parallelism with multiple device meshes, new patterns such as multiple program multiple data (MPMD) and cross-mesh communication appear (Barham et al., 2022). Existing work does not address the new

patterns while our paper first comprehensively studies them.

Overlapping of computation and communication Overlapping is an effective technique to hide the latency of communication. Prior work has studied scheduling and partition methods to achieve better overlapping (Hashemi et al., 2019; Jayarajan et al., 2019; Jiang et al., 2020; Jangda et al., 2022a). They either focus on data parallelism or low-level communication primitives, while this paper proposes a new high-level pipeline schedule for model parallelism.

7 Conclusion

In this paper, we introduce communication optimizations to accelerate the cross-mesh resharding, which is a new and important communication pattern in distributed training of large-scale neural networks. We optimize the problem at three granularity including a unit communication task, the load balance among all unit tasks in a cross-mesh resharding task and overlapping all cross-mesh resharding tasks with computations. We port a state-of-the-art distributed training framework, Alpa, on top of our work. Our work speeds up training of large models by up to 1.5x and achieves a performance very close to the hypothetical upper bound.

References

Andreyev, A. Introducing data center fabric, the next-generation facebook data center network, 2014.

Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

Barnett, M., Shuler, L., van De Geijn, R., Gupta, S., Payne, D. G., and Watts, J. Interprocessor collective communication library (intercom). In *Proceedings of IEEE Scalable High Performance Computing Conference*, pp. 357–364. IEEE, 1994.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 62–75, 2021.

Chan, E., Heimlich, M., Purkayastha, A., and Van De Geijn, R. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

Cho, M., Finkler, U., Kung, D., and Hunter, H. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems*, 1:241–251, 2019.

Cowan, M., Maleki, S., Musuvathi, M., Saarikivi, O., and Xiong, Y. Gc3: An optimizing compiler for gpu collective communication. *arXiv preprint arXiv:2201.11840*, 2022.

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

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jangda, A., Huang, J., Liu, G., Sabet, A. H. N., Maleki, S., Miao, Y., Musuvathi, M., Mytkowicz, T., and Saarikivi, O. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 402–416, 2022a.

Jangda, A., Huang, J., Liu, G., Sabet, A. H. N., Maleki, S., Miao, Y., Musuvathi, M., Mytkowicz, T., and Saarikivi, O. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pp. 402–416, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507778. URL https://doi.org/10.1145/3503222.3507778.

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

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., and Guo, C. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 463–479, 2020.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pp. 6543–6552. PMLR, 2021.

Luo, L., West, P., Nelson, J., Krishnamurthy, A., and Ceze, L. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems*, 2:82–97, 2020.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021a.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021b.

NVIDIA. The nvidia collective communication library, 2018. URL https://developer.nvidia.com/nccl.

Patarasuk, P. and Yuan, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

Petit, O., Thome, N., Rambour, C., Themyr, L., Collins, T., and Soler, L. U-net transformer: Self and cross attention for medical image segmentation. In Lian, C., Cao, X., Rekik, I., Xu, X., and Yan, P. (eds.), *Machine Learning in Medical Imaging*, pp. 267–276, Cham, 2021. Springer International Publishing. ISBN 978-3-030-87589-3.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Rink, N. A., Paszke, A., Vytiniotis, D., and Schmid, G. S. Memory-efficient array redistribution through portable collective communication. *arXiv preprint arXiv:2112.01075*, 2021.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

Scott, D. S. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pp. 398–399. IEEE Computer Society, 1991.

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

Shah, A., Chidambaram, V., Cowan, M., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O., and Singh, R. Synthesizing collective communication algorithms for heterogeneous networks with taccl. *arXiv preprint arXiv:2111.04867*, 2021.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Unger, C., Jia, Z., Wu, W., Lin, S., Baines, M., Narvaez, C. E. Q., Ramakrishnaiah, V., Prajapati, N., McCormick, P., Mohd-Yusof, J., et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 267–284, 2022.

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

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019.

Xie, N., Norman, T., Grewe, D., and Vytiniotis, D. Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. *Proceedings of Machine Learning and Systems*, 4:548–566, 2022.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Zhang, H., Li, Y., Deng, Z., Liang, X., Carin, L., and Xing, E. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

Zhuang, S., Li, Z., Zhuo, D., Wang, S., Liang, E., Nishihara, R., Moritz, P., and Stoica, I. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 641–656, 2021.

A ARTIFACT APPENDIX

A.1 Artifact check-list (meta-information)

- **Compilation:** Compile the jaxlib binary, script provided;
- **Binary:** compiled jaxlib binary;
- **Run-time environment:** CUDA toolkit 11.1 or higher;
- **Hardware:** multiple GPU with hierarchical bandwidth;
- **Metrics:** Throughput;
- **Output:** Execution time and throughput;
- **How much disk space required (approximately)?:** up to 20 GB;
- **How much time is needed to prepare workflow (approximately)?:** up to 1 hour;
- **How much time is needed to complete experiments (approximately)?:** about 1 hour;
- **Publicly available?:** Merged to Alpa main branch online.
- **Code licenses (if publicly available)?:** Apache-2.0
- **Workflow framework used?:** Alpa, Flax
- **Archived (provide DOI)?:** Available online

A.2 Description

A.2.1 How delivered

We implement our method on top of Alpa, and our code is already merged into its main branch.

A.2.2 Hardware dependencies

Our work depends on the GPU environment, and the configuration of our experiment is in table 4

Table 4. hardware configurations

Hardware	Requirements
Instance number	2 to 5
GPU	4 Tesla V100 16 GB
GPU peer-to-peer	NVLINK
Processor	AWS custom Intel(R) Xeon(R) Scalable (Skylake) vCPUs
Memory	128 GB
Network bandwidth	10 Gbps

A.2.3 Software dependencies

The reader is supposed to install git and python3 with pip.

The user is also supposed to install nightly alpa and its prerequisites including cupy, nccl, and some python libs. The tutorial is available in `https://alpa.ai/install.html#method-2-install-from-source`.

A.3 Installation

We create a branch for the ease of artifact evaluation: https://github.com/alpa-projects/alpa/tree/mlsys23-artifact. To run our nightly code, the user is supposed to:

1. clone the code above;
2. compile our specified jaxlib following the instruction of alpa's installation guide;
3. install the compiled jaxlib under "build_jaxlib/dist" with the correct jax version under "third_party/jax";
4. install alpa under the cloned folder.

In "artifact_evaluation_scripts" folder, we provide a script to set up the environment("setup_env.sh").

A.4 Experiment workflow

We provide a script that first runs experiments in our micro-benchmark one by one, then executes the end-to-end benchmark: "run_benchmark.sh". The user can also use "microbenchmark.py" or "benchmark.py" to run specific benchmark cases.

A.5 Evaluation and expected result

The evaluation results will be shown on screen during the experiment workflow. For microbenchmarks, the result will be recorded in a json file of each test suite. For end-to-end experiments, the result will be recorded in a tsv file.

The user is expected to have the same result shown in our paper: in micro-benchmarks, resharding_mode="broadcast" is supposed to outperform resharding_mode="send_recv", whether use local allgather or not; For the "resharding_loadbalance_mode" term, "loadbalance_order" is supposed to outperform "loadbalance_size", which is better than "no_loadbalance". In end-to-end benchmarks, overlapping with specific pipeline schedule is better than overlapping without the schedule, which is better on the u-net than turning off the overlapping.

A.6 Experiment customization

There are some other configurations for users with different numbers of GPUs in the "suite_unet.py" and "suite_manual_gpt.py" files. The user can also select which case of the microbenchmark to run by configuring the "suite_microbenchmark.py" file.

Though we run our microbenchmark assuming the sender and receiver are one different node, we provide an alternative to let them share the same node by adding –functional-test.

B Decomposing to Unit Tasks

B.1 Example of all unit tasks from a cross-mesh resharding

We examples of all unit tasks decomposed from cross-mesh resharding tasks in Figure 2. The unit tasks for cross-mesh resharding task 1 are listed in Figure 10, and that for cross-mesh resharding tasks 2 are in Figure 11.

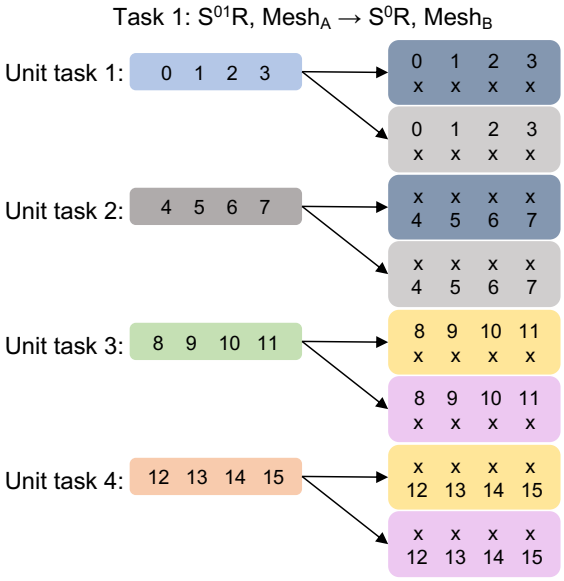


Figure 10. Unit tasks for cross-mesh resharding case 1.

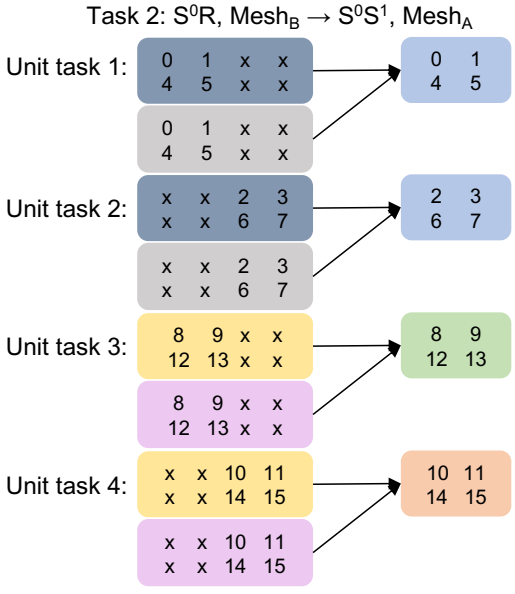


Figure 11. Unit tasks for cross-mesh resharding case 2.

B.2 Algorithm to decompose a cross-mesh resharding

We decompose a cross-mesh resharding into unit tasks by two steps: first, we split a tensor into slices based on the sharding spec of both sender and receiver mesh.

After that, for each slice, we create a unit task whose senders are all send devices owning the slice, and receivers are all receive devices requiring the slice.

To create slices, we enumerate the tensor dimensions. Each dimension d has a set of cutpoints, denoted by $Cut(d)$. We use l_d to represent the length of dimension d. $Cut(d)$ is initialized with value $\{0, l_d\}$ for each dimension d

If dimension d is sharded on the sender mesh with a degree of k, the dimension is split into k equal size replicas, and thus we add $l_d/k, 2l_d/k, \dots$ into the cutpoint set Cut_d. The same applies for the sharding spec on receiver dimension as well. Eventually, for a tensor with n dimensions $d_0, d_1 \dots d_{n-1}$, we get n corresponding cutpoint sets.

Let c_d be the size of the cutpoint set Cut_d. For each cutpoint set $Cut_d = \{p_0, p_1, \dots p_{c_d-1}\}$ where $p_0 < p_1 < \cdots < p_{c_d-1}$, we create the corresponding interval set $I_d = \{(p_0, p_1), (p_1, p_2) \dots (p_{c_d-2}, p_{c_d-1})\}$. The final slice set is represented by the cross product of the interval set for all tensor dimensions, i.e. $I_0 \times I_1 \times \dots I_{d-1}$.