Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems

Siyuan Zhuang[1,*] Zhuohan Li[1,*] Danyang Zhuo[2] Stephanie Wang[1]
Eric Liang[1] Robert Nishihara[1] Philipp Moritz[1] Ion Stoica[1]
[1]University of California, Berkeley [2]Duke University

ABSTRACT

Task-based distributed frameworks (e.g., Ray, Dask, Hydro) have become increasingly popular for distributed applications that contain asynchronous and dynamic workloads, including asynchronous gradient descent, reinforcement learning, and model serving. As more data-intensive applications move to run on top of task-based systems, collective communication efficiency has become an important problem. Unfortunately, traditional collective communication libraries (e.g., MPI, Horovod, NCCL) are an ill fit, because they require the communication schedule to be known before runtime and they do not provide fault tolerance.

We design and implement Hoplite, an efficient and fault-tolerant collective communication layer for task-based distributed systems. Our key technique is to compute data transfer schedules on the fly and execute the schedules efficiently through fine-grained pipelining. At the same time, when a task fails, the data transfer schedule adapts quickly to allow other tasks to keep making progress. We apply Hoplite to a popular task-based distributed framework, Ray. We show that Hoplite speeds up asynchronous stochastic gradient descent, reinforcement learning, and serving an ensemble of machine learning models that are difficult to execute efficiently with traditional collective communication by up to 7.8x, 3.9x, and 3.3x, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → **Distributed computing methodologies**.

KEYWORDS

Collective Communication, Distributed Systems

ACM Reference Format:
Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, Ion Stoica. 2021. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472897

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472897

1 INTRODUCTION

Task-based distributed systems (e.g., Ray [30], Hydro [19], Dask [44], CIEL [32]) have become increasingly popular for developing and running distributed applications that contain asynchronous and dynamic computation and communication patterns, including asynchronous stochastic gradient descent (SGD), reinforcement learning (RL), and model serving. Today, many top technology companies have started to adopt task-based distributed frameworks for their distributed applications, such as Intel, Microsoft, Ericsson, and JP. Morgan. For example, Ant Financial uses task-based distributed systems to run their online machine learning pipeline and serve financial transactions for billions of users [22].

There are two key benefits of building distributed applications on top of task-based systems. First, it is easy to express asynchronous and dynamic computation and communication patterns. A task-based system implements a *dynamic task* model: a caller can dynamically invoke a task A, which immediately returns an *object future*, i.e. a reference to the eventual return value. By passing the future as an argument, the caller can specify another task B that uses the return value of A even before A finishes. The task-based system is responsible for scheduling *workers* to execute tasks A and B and transferring the result of A to B between the corresponding workers. Second, fault tolerance is provided by the task-based system transparently. When a task fails, the task-based system quickly reconstructs the state of the failed task and resumes execution [49, 52]. Well-behaving tasks do not need to roll back, so failure recovery is low cost.

As a growing number of data-intensive workloads are moving to task-based distributed systems, supporting efficient collective communication (e.g., broadcast, reduce) has become critical. Consider an RL application where the trainer process broadcasts a policy to a set of agents that use this policy to perform a series of simulations. Without the support for collective broadcast, the trainer process needs to send the same policy to every agent which causes a network bottleneck on the sender side.

Efficient collective communication is a well-understood problem in the HPC community and in distributed data-parallel training. Many collective communication libraries exist today, e.g., OpenMPI [16], MPICH [31], Horovod [47], Gloo [14], and NCCL [34]. However, there are two limitations of traditional collective communication implementations that make them an ill fit for dynamic task-based systems.

First, a distributed application using traditional collective communication must specify the communication pattern *before* runtime. This allows the library to compute a *static* and efficient data transfer schedule (e.g., ring-allreduce). For example, for *synchronous*

*Equal contribution.

distributed data-parallel training, the application specifies that all workers participate in an allreduce communication, once per training round.

However, in task-based systems, the set of tasks or data objects participating in the collective communication is not known before runtime. One approach would be to wait until all the participating tasks and objects are ready and then compute a static data transfer schedule. Unfortunately, this design misses the opportunity to make partial progress before the entire set of participants are ready, which is critical for the performance of modern *asynchronous* applications, e.g., distributed RL.

Second, because of the synchronous nature of collective communications, one process failure can cause the rest of the processes to hang. Existing solutions leave the recovery up to the application. A typical approach is to checkpoint the state of the application periodically (e.g., every hour), and when a process fails, the entire application rolls back to the previous checkpoint and restarts. Unfortunately, this can be expensive for large-scale asynchronous applications, and does not exploit the ability of tasks that are still alive in the same collective communication group as a failed task to make progress.

This raises an important question: *how can we bring the efficiency of collective communication to dynamic and asynchronous task-based applications?* There are two requirements that are unique to this setting. First, the application must be allowed to specify the participants of a collective communication *dynamically* (i.e., at runtime). Second, the collective communication implementation must be *asynchronous*. This would allow tasks to make progress even if other tasks in the same communication group have failed.

We design and implement Hoplite, an efficient and fault-tolerant collective communication layer for task-based distributed systems. Hoplite combines two key ideas: (1) Hoplite computes data transfer schedule on-the-fly as tasks and objects arrive, and Hoplite executes data transfer schedule efficiently using fine-grained pipelining. Collective communication can make significant progress even if only a fraction of the participants are ready. (2) Hoplite dynamically adapts the data transfer schedule when a failure is detected to alleviate the effects of the failed task in collective communication. This allows the live tasks to make progress. The failed task can rejoin the collective communication after being restarted and complete the communication.

We apply Hoplite to a popular task-based framework, Ray [30]. This allows us to evaluate a wide range of existing workloads on Ray. Our evaluations show that Hoplite can speed up an asynchronous SGD by up to 7.8x, two popular RL algorithms (IMPALA [13], and A3C [29]) on RLlib [27] by up to 1.9x, and 3.9x, respectively, and improve the serving throughput time of an ensemble of ML models on Ray Serve [42] by up to 3.3x, with only minimal code changes and negligible additional latency in failure recovery.

This paper makes the following contributions:

- A distributed scheduling scheme for data transfer that provides efficient broadcast and reduce primitives for dynamic-task systems.
- A fine-grained pipeline scheme that achieves low-latency data transfers between tasks located both on the same node or on different nodes.

```python
def train(policy, num_agents, num_steps, batch_size):
  # Start some rollouts in parallel.
  grad_ids = [rollout.remote(policy)
              for _ in range(num_agents)]
  for _ in range(num_steps):
    for _ in range(batch_size):
      # Wait for the first rollout to finish.
      ready_id = ray.wait(grad_ids)
      # Update the policy with one gradient.
      policy += ray.get(ready_id) / batch_size
      # Remove this gradient from remaining gradients
      grad_ids.remove(ready_id)
    # Once one batch of agents finish, broadcast updated
    # policy to finished agents and start new rollouts.
    for _ in range(batch_size):
      grad_ids.append(rollout.remote(policy))
  return policy
```

(a) Dynamic tasks (Ray).

```
 for _ in range(num_steps):
-  for _ in range(batch_size):
-    # Wait for the first rollout to finish.
-    ready_id = ray.wait(grad_ids)
-    # Update the policy with one gradient.
-    policy += ray.get(ready_id) / batch_size
-    # Remove this gradient from remaining gradients
-    grad_ids.remove(ready_id)
+  # Reduce a batch of gradients
+  reduced_grad_id, unreduced_grad_ids = \
+  ray.reduce(grad_ids, num_return=batch_size, op=ray.ADD)
+  # Update the policy with the averaged gradient
+  policy += ray.get(reduced_grad_id) / batch_size
+  # Update remaining gradients
+  grad_ids = ray.get(unreduced_grad_ids)
   # Once one batch of agents finish, broadcast updated
   # policy to finished agents and start new rollouts.
   for _ in range(batch_size):
     grad_ids.append(rollout.remote(policy))
```

(b) Dynamic tasks + collective comm. (Ray + Hoplite).

Figure 1: Pseudocode for a typical RL algorithm to learn a policy. **(a)** Dynamic tasks with Ray. Each `train` loop waits for a *single* agent to finish, then asynchronously updates the current policy. The new policy is broadcast to a batch of finished agents. **(b)** Modifications to (a) to enable Hoplite. Each step reduces gradients from a subset of agents, updates the current policy, broadcasts the new policy.

- Algorithms to adapt the schedule of the data transfers for broadcast and reduce operations which allows live tasks to make progress when other tasks that participate in the collective communication have failed, and later allow those failed tasks to rejoin.
- We demonstrate the benefits of Hoplite on top of a popular task-based distributed system using several applications, including asynchronous SGD, RL, and serving an ensemble of ML models.

2 BACKGROUND

We first describe task-based distributed systems and their benefits for developing distributed applications. We then describe the challenges of integrating efficient collective communication into them.

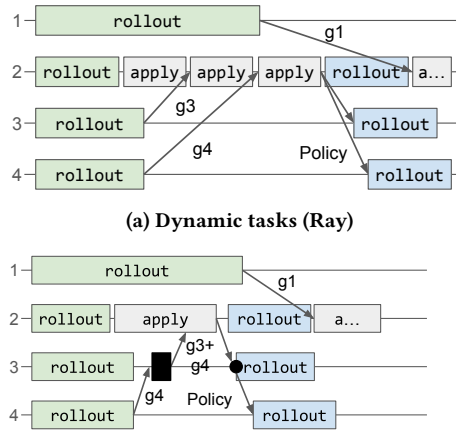
2.1 Task-Based Distributed Systems

The dynamic task programming model [4, 19, 30, 32, 44] allows applications to express asynchronous and dynamic computation

and communication patterns. For instance, Figure 1a shows how to implement an *asynchronous* RL algorithm that updates the policy with agent results one at a time, choosing them *dynamically* based on the order of availability. Once a batch of agent results have been applied, the resulting policy is sent to each finished agent to begin the next round of rollout. This allows an agent that has a fast rollout not need to wait for a worker that has a slow rollout (Figure 2a). Today, most RL algorithms [13, 29] leverage this type of asynchronous execution for efficient training.

To support this type of asynchronous communication, task-based distributed systems rely on a *distributed object store* to transfer objects between tasks. The object store consists of a set of *nodes*, each of which buffers a (possibly overlapping) set of application objects. Each node serves multiple workers, which can read and write directly to objects in its local node via shared memory. A sender task stores the output into the object store and exits, allowing it to release critical resources (e.g., CPU, GPU, memory) before the receiver tasks are even scheduled. When receiver tasks are ready, they directly fetch the object from the distributed object store. As is standard [30, 32], the object store enforces object immutability and uses a distributed object directory service to map each object to its set of node locations. In addition, task-based distributed systems support fast failure recovery [49, 52] by reconstructing the failed task. Well-behaving tasks do not roll back to keep recovery low cost.

However, if the gradients and the model are large enough in the above RL example, task-based distributed systems incur significant overheads from inefficient communication. For example, the trainer (agent 2) in Figure 2a can become a network throughput bottleneck since it has to receive the gradient and also send the new policy from/to each agent individually. This bottleneck becomes more severe when the number of agents increases.



(a) Dynamic tasks (Ray)

(b) Dynamic tasks + collective comm. (Ray + Hoplite).

Figure 2: Execution of a distributed RL algorithm. Each row is one agent. Boxes represent computations, and arrows represent data transfers. $g1$-$g4$ are the gradients produced by the agents. **(a)** Dynamic tasks (Ray). Gradients are applied immediately. A batch of three gradients is applied to the current policy before broadcasting. **(b)** Dynamic tasks but with efficient collective communication, in Hoplite. To reduce the network bottleneck at agent 2, agent 3 partially reduces gradients $g3$ and $g4$ (black box), and agent 3 sends the policy to agent 4 (black dot) during the broadcast.

2.2 Challenges in Collective Communication

Efficient collective communication has well-known solutions in HPC community and in distributed data-parallel SGD. Many traditional collective communication libraries exist, including Gloo [14], Horovord [47], OpenMPI [16], MPICH [31], and NCCL [34]. They can use efficient data transfer schedule (e.g., ring-allreduce, tree-broadcast) to mitigate communication bottlenecks in distributed applications.

There are two application requirements for using traditional collective communication libraries. First, the communication pattern has to be statically defined before runtime. This is easy for applications that have a bulk-synchronous parallel model. For example, in synchronous data-parallel SGD, all the workers compute on their partitioned set of training data and synchronize the model parameters using allreduce. Second, when any worker fails, all the workers participating in the collective communication hang, and applications are responsible for fault tolerance. For HPC applications, this is typically solved by checkpointing the entire application periodically (e.g., per-hour), and when a process fails, the entire application rolls back to a checkpoint and re-execute.

Unfortunately, these two assumptions are fundamentally incompatible with task-based distributed systems. First, tasks are dynamically invoked by the task-based system's scheduler. This means it is possible that, when collective communication is triggered, only a fraction of the participating tasks are scheduled. For example, on existing task systems, broadcast is implicit: a set of tasks fetch the same object. When only a subset of the receivers are scheduled, it is not possible to build a static broadcast tree without knowing how many total receivers and where and when the receivers will be scheduled. *Therefore, a collective communication layer for a task-based system should adjust data transfer schedule at runtime based on task and object arrivals.*

Second, fast failure recovery is an important design goal for task-based system [49, 52], because many asynchronous workloads have tight SLO requirement (e.g., model serving). In existing task systems, this is done by reconstructing and re-executing failed tasks only. If traditional collective communication libraries are used, a failed task causes the rest of the participating tasks to hang. Thus, *a collective communication layer for task-based systems has to be fault-tolerant: allowing well-behaving tasks to make progress when a task fails and allowing the failed task to rejoin the collective communication after recovery.*

3 DESIGN

Hoplite is an efficient and fault-tolerant collective communication layer for task-based distributed systems. At a high level, Hoplite uses two techniques: (1) decentralized fault-tolerant coordination of data transfer for reduce and broadcast, and (2) pipelining of object transfers both across nodes and between tasks and the object store.

We first present a send-receive example workflow using Hoplite's core API (Table 1). We then describe Hoplite's object directory service, pipelining mechanism to reduce latency, and fault-tolerant

```
def application():
  x_id = send.remote()
  recv.remote(x_id)
```

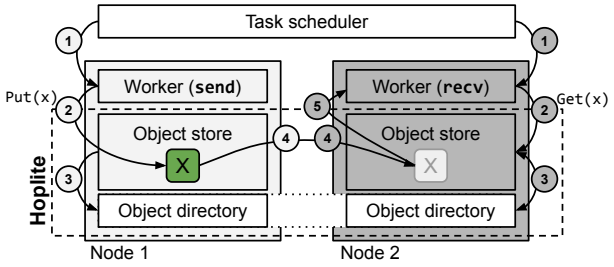


Figure 3: Example of a send and receive dynamic task program on a 2-node cluster (N1 and N2). The task-based system consists of a pool of workers per physical node and a scheduler. Hoplite consists of one local object store per node and a global object directory service, which is distributed across physical nodes.

receiver-driven coordination scheme for efficient object transfer in details.

3.1 Hoplite's Workflow

Our example creates a send task that returns x_id (a future), which is then passed into a recv task. In Hoplite, we use an ObjectID to represent a future or a reference to an object. During execution, the application first submits the tasks to the task scheduler. The scheduler then chooses a worker to execute each task (step 1, Figure 3), e.g., based on resource availability. According to the application, recv cannot start executing until it has the value returned by send. Note that the task-based system does not require the scheduler to schedule tasks in a particular location or order, i.e. the recv task may be scheduled *before* send.

In step 2, the task workers call into Hoplite to store and retrieve objects. On node 1, the send worker returns an object with the unique ID x_id. This object must be stored until the recv worker has received it. Thus, the send worker calls Put(x) on Hoplite, which copies the object from the worker into the local object store (step 2 on N1, Figure 3). This frees the worker to execute another task, but incurs an additional memory copy between processes to store objects.

Meanwhile, on node 2, the recv worker must retrieve the object returned by send. To do this, it calls Get(x) on Hoplite, which blocks until the requested object has been copied into the worker's local memory (step 2 on N2, Figure 3). In step 3, Hoplite uses the *object directory service* to discover object locations and coordinate data transfer, in order to fulfill the client's Put and Get requests. In the example, the Hoplite object store on node 1 publishes the new location for the object x to the directory (step 3 on N1, Figure 3). Meanwhile, on node 2, the Hoplite object store queries the directory for a location for x (step 3 on N2, Figure 3).

Hoplite's object directory service (§3.2) is implemented as a sharded hash table that is distributed throughout the cluster (Figure 3). Each shard maps an ObjectID to the current set of node locations. When there are multiple locations for an object, the directory service can choose a *single* location to return to the client. The object store also maintains information about objects that have only been partially created to facilitate object transfer pipelining (§3.3). For example, in Figure 3, the object store on node 1 publishes its location to the object directory as soon as Put(x) is called, even if the object hasn't been fully copied into the store yet. This allows node 1 to begin sending the object to node 2 while it is still being copied from the send worker.

Finally, in step 4, the Hoplite object store nodes execute the data transfer schedule specified by the object directory's reply to node 2. Node 1 is the only location for x, so node 2 requests and receives a copy from node 1 (step 4). Node 2 then copies the object from its local store to the recv worker (step 5 in Figure 3), which again can be pipelined with the copy over the network.

Hoplite provides two efficient collective communication schemes. Hoplite implements efficient broadcast through coordination between the object directory service and the workers (§3.4), without an explicit primitive. For reduce, Hoplite exposes an explicit Reduce call to the task-based system. It is necessary because this lets Hoplite know that these objects are indeed reducible (i.e., the operation is commutative and associative). Because an ObjectID is a future that the object value may not be ready yet, the Reduce call also has a num_objects input in case the user wants to reduce a subset of the objects, giving Hoplite the flexibility to choose which num_objects objects to reduce given their arrival time in the future. Figure 1b shows how to modify the RL example to use Hoplite. This allows the trainer to aggregate gradients from a dynamic set of agents efficiently (Figure 2b).

Whenever a task fails, Hoplite recomputes a data transfer schedule to avoid using the failed task in the collective communication, and all the rest of the tasks can keep making progress (§3.5). Hoplite does not change how task-based distributed system tolerate failures. The underlying task-based distributed system can quickly reconstruct the state of the failed task using their built-in mechanism [52]. Once the state of the task is reconstructed, the task resumes.

3.2 Object Directory Service

The object directory service maintains two fields for each object: (1) the size of the object, and (2) the location information. The location information is a list of node IP addresses and the current progress of the object on that node. We use a single bit to represent the object's progress: either the node contains a partial or a complete object. We store both so that partial object copies can immediately act as senders, for both broadcast and reduce (§3.4).

Hoplite's directory service supports both synchronous and asynchronous location queries. Synchronous location queries block until corresponding objects are created and locations are known. Asynchronous location queries return immediately, and the object directory service publishes any future locations of the object to the client.

A node writes object locations to the object directory service in two conditions: when a local client creates an object via Put and when an object is copied from a remote node. In each case, the node notifies the object directory service twice: once when an object is about to be created in the local store and once when the complete object is ready. We differentiate between partial and

Core Interfaces:	Description
Buffer buffer← Get(ObjectID object_id)	Get an object buffer from an object id.
Put(ObjectID object_id, Buffer buffer)	Create an object with a given object id and an object buffer.
Delete(ObjectID object_id)	Delete all copies of an object with a given object id. Called by the task framework once an object is no longer in use.
Reduce(ObjectID target_object_id, int num_objects, {ObjectID source_object_id, ...}, ReduceOp op)	Create a new object with a given object id from a set of objects using a reduce operation (e.g, sum, min, max).

Table 1: Core Hoplite APIs. The application generates an `ObjectID` with a unique string and can pass an `ObjectID` by sending the string.

complete objects so that object store nodes with complete copies can be favored during a broadcast or reduce (§3.4).

Optimization for small objects. Querying object location can introduce an excessive latency penalty for fetching small objects, and the overhead of computing efficient object transfer schedule is usually not worthwhile for small objects in our use cases. Therefore, we implement a fast path in the object directory service. For small objects (<64KB), we simply cache them in the object directory service, and when a node queries for their location, the object directory service directly returns the object buffers. Similar to object in the per-node stores, cached objects must be freed by the application via the `Delete` call when no longer in use.

3.3 Pipelining

Hoplite uses pipelining to achieve low-latency transfer between processes and across nodes for large objects. This is implemented by enabling a receiver node to fetch an object that is incomplete in a source node. An object can be incomplete if the operation that created the object, either a `Put` from the client or a copy between object store nodes, is still in progress. To enable fetching incomplete objects, as shown in the previous section (§3.2), the object directory service also maintains locations of incomplete copies. Then, when an object store receives a `Get` operation, it can choose to request the object from a store with an incomplete copy.

By pipelining data transfers across nodes using the object directory service as an intermediary, it becomes simple to also pipeline higher-level collective communication primitives, such as a reduce followed by a broadcast (Figure 2b). Within the reduce, a node can compute a reduce of a subset of the input objects and simultaneously send the intermediate result to a downstream node. The downstream node can then compute the final reduce result by computing on the intermediate result as it is received and simultaneously send the final result to any broadcast receivers that have been scheduled. A broadcast receiver can then also simultaneously send the final result to any other broadcast receivers.

Piplining between the task worker and local store on the same node is also important to hide `Put` and `Get` latency for large objects (steps 2 and 5 in Figure 3). The reason is that using the distributed object store requires two additional data copies other than the minimum needed to transfer data over the network. The sender task worker must copy to its local store, and then the receiving local store must also copy to its local worker. Our observation is that the additional memory copy latency can be masked by the network transfer if the memory copy is asynchronous. When a sender task calls `Put`, Hoplite immediately notifies the object directory service that the object is ready to transfer. A receiver can then fetch the object before the entire object is copied into the sender node's local store. The receiver side's pipelining mechanism is similar. When the receiver task calls `Get`, the receiver task starts to copy the object from the local store before the local store has a complete object.

By combining cross-node and in-node pipelining, Hoplite enables end-to-end object streaming between the sender and receiver tasks, even when there are multiple rounds of collective communication in between.

Optimization for immutable get. Although Hoplite objects are immutable, the receiver task still copies the object data from its local store during a `Get`, in case it modifies the buffer later on. However, if it only needs read access to the object, then Hoplite can directly return a pointer inside the local store. Read-only access can be enforced through the front-end programming language, e.g., with `const` in C++.

3.4 Receiver-Driven Collective Communication

Hoplite's receiver-driven coordination scheme optimizes data transfer using distributed protocols. In Hoplite, data transfer happens in two scenarios: either a task calls `Get` to retrieve an object with a given `ObjectID`, or a task calls `Reduce` to create a new object by reducing a set of other objects with a reduce operation (e.g., sum, min, max).

3.4.1 Broadcast. Broadcast in a task-based distributed system happens when a group of tasks located on multiple nodes want to get the same object from its creator task. Specifically, a *sender* task from node `S` creates an object with `Put` and a group of *receiver* tasks `R1`, `R2`, ... fetches it using `Get`. For the receiver tasks that locate on different nodes from the sender task, their corresponding receiver nodes will fetch the object from sender node's local object store to the receiver nodes' local object store. To simplify the description of our method, we assume that the sender task and the receiver tasks locate on different nodes and use the sender `S` and the receiver `R1`, `R2`, ... to also refer to the local object store on the nodes.

Broadcast in a task-based distributed system is challenging because we have no knowledge of the tasks, including where these tasks are located and when these tasks fetch the object. If all receivers simply fetch the object from the sender, the performance will be restricted by the sender's upstream bandwidth. Traditional collective communication libraries can generate a static tree where the root is the sender node to mitigate the throughput bottleneck. The goal of Hoplite's receiver-driven coordination scheme is to achieve a similar effect but using decentralized protocols. Inspired

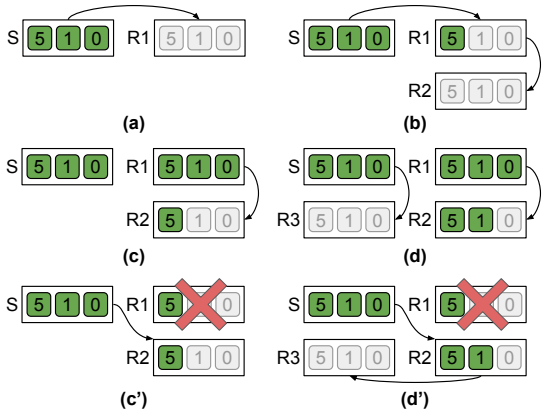


Figure 4: An example of broadcasting an object (integer array {5, 1, 0}) from a sender (S) in Hoplite, when the receivers (R1-R3) arrive at different times. **(a)** - **(d)** show the broadcast process without failure. **(c')** and **(d')** show the broadcast process when R1 fails after **(b)**.

by application-level broadcast [5, 6] in peer-to-peer systems that uses high-capacity nodes to serve as intermediate nodes in the broadcast tree, we use receivers who receives the object earlier than the rest as intermediates to construct a broadcast tree.

When a receiver R wants to fetch a remote object, it first checks if the object is locally available, or there is an on-going request for the object locally. If so, the receiver just waits until it gets the completed object. This avoids creating cyclic object dependencies. Otherwise, R queries the object directory service for the object's location. The object directory service first tries to return *one* location with a complete copy. If none exist, then the object directory service returns one of the locations holding a partial copy. This is so that partial objects can also act as intermediate senders, but locations with complete copies are favored.

When the location query replies, R also removes the location returned from the directory and immediately add itself to the object directory as a location with a partial copy to enable pipelining. Once the data transfer is complete, the receiver adds the sender's location back to the object directory service and mark itself as a location with a complete copy. This makes sure that, for each object, a node can only send to one receiver at a time, thus mitigating bottlenecks at any single node.

Figure 4 shows an example of a broadcast scenario in Hoplite. In Figure 4a, the first receiver R1 starts to fetch the object from the sender S. In Figure 4b, S is still sending to R1, so it does not appear in the object directory when the second receiver R2 arrives. Thus, R2 fetches the object from R1, the partial copy. In Figure 4c, R1 has finished receiving, but is still sending to R2. Then, the object directory contains S and R2 as a complete and partial location, respectively. In Figure 4d, R3 queries the object directory, which chooses S over R2 as the sender because S has a complete object.

3.4.2 Reduce. Reduce happens when a task in a task-based distributed system wants to get a reduced object (e.g., summed or maximal object) from a list of objects. In Hoplite, this happens via a Reduce call. Similar to broadcast, we assume that each object to reduce is located on a separate node and we use R1, R2, ... to represent both the object and the local object store on a node that stores the corresponding object. Note that in a task-based distributed system, the objects to reduce can become ready to reduce in any arbitrary order.

How to reduce objects efficiently to accommodate dynamic object creation is more challenging than broadcast. Broadcast is simpler because a receiver can fetch the object from any sender, and Hoplite thus has more flexibility to adapting data transfer schedule. For reduce, we need to make sure all the objects are reduced once and only once: when one object is added into a partial reduce result, the object should not be added into any other partial results.

In Hoplite, we choose to use a tree-structured reduce algorithm, while the question is what type of tree to use. Let's think about reducing n objects. Without the support of collective communication in task-based distributed systems, each node sends the object to a single receiver. Let's assume that the network latency is L , network bandwidth is B, and the object size is S. This approach's total reduce running time is $L + \frac{nS}{B}$. The L term is due to the network latency, and $\frac{nS}{B}$ is due to the receiver's bandwidth constraint, because every node has to send the object in to it. This is a special kind of tree where the degree of the root is n. When object size is very small (i.e., $\frac{S}{B}$ is negligible), the performance of this kind of tree is the best.

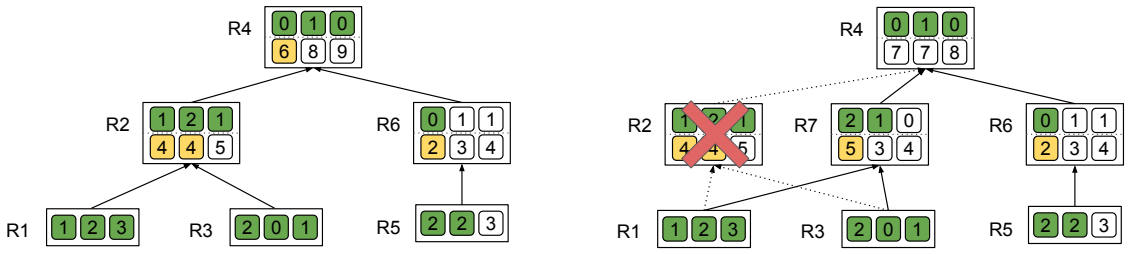
To mitigate the bandwidth bottleneck at the receiver, we can generalize this n-nary tree to a d-nary tree. When we use a d-nary tree, the total running time is $L \log_d n + \frac{dS}{B}$. It reduces the latency due to the bandwidth constraint but incurs additional latency because the height of the tree grows to $\log_d n$. If an object is very large (i.e., $\frac{S}{B} \gg L$), we can set $d = 1$. This means all the nodes are in a single chain, and its running time is $nL + \frac{S}{B}$. Note that we only need to incur $\frac{S}{B}$ for transferring the actual content of the object, because we use fine-grained pipelining, i.e., intermediate nodes send the partially reduced object to the next node. As we can see here, the optimal choice of d depends on the network characteristics, the size of the object, and the number of participants (objects). In other words, we choose the d to minimize the total latency:

$$T(d) = \begin{cases} nL + \frac{S}{B} & \text{if } d = 1; \\ L \log_d n + \frac{dS}{B} & \text{otherwise.} \end{cases} \quad (1)$$

During runtime, Hoplite will automatically chooses the optimal d based on an empirical measure of these three factors.

Once the topology of the tree is determined, we need to assign nodes into the tree. Here we want to allow Reduce to make significant progress even with a subset of objects. To do so, we assign arriving objects with a generalized version of in-order tree traversal. For a d-nary tree, for each node, we traverse the first child, the node itself, the second child, third child, ..., and the d-th child. Figure 5a shows an example for reducing 6 objects with a binary tree. Note that though MPI also supports tree-reduce, our method is completely different: MPI's tree is constructed statically, and our tree is constructed dynamically taking the object arrival sequence into consideration.

If a task only wants to reduce a subset of objects (i.e., `num_object` is smaller than the size of the source object list in Reduce), the tree construction process stops when there are `num_object` objects in the tree. For example, if the task wants to reduce 6 out of 10 objects,



(a) Reduce Tree. **(b) Reduce Tree with failure.**

Figure 5: Examples of reduce where the objects arrive in the order of R1, R2, ..., R6. The numbers on the top of each node (and the numbers in leaf nodes) represent the object to reduce and green blocks means the fraction of the object that is ready. The numbers on the bottom of each node represent the reduced result and yellow blocks means the fraction of the object that has been reduced. Each intermediate node is responsible to reduce the subtree rooted at it. **(a)** An example reduce tree consists of 6 objects. **(b)** The reconstructed reduce tree after R2 fails.

then the earliest arriving 6 objects are in the reduce tree structured as Figure 5a.

An application can also specify the inputs of a Reduce incrementally, i.e. by passing the ObjectID result of one Reduce operation as an input of a subsequent Reduce operation. The data transfer for composed Reduce operations will naturally compose together. In particular, as soon as the first Reduce output is partially ready, it will be added to the object directory service, where it will be discovered by the downstream Reduce coordinator. The first output can then be streamed into the downstream Reduce.

3.4.3 AllReduce. AllReduce is a synchronous collective communication operation that is useful for synchronous data-parallel training. Optimizing allreduce is not our design goal: people usually do synchronous data-parallel training on specialized distributed systems that are optimized for bulk-synchronous workloads (e.g., TensorFlow [1], PyTorch [37]) rather than on task-based distributed systems. In Hoplite, a developer can express allreduce by concatenating reduce and broadcast.

3.5 Fault-Tolerant Collective Communication

In the previous subsection, we assume that there is no task failures. However, task failures can happen in a task-based distributed systems for various reasons, including (1) the node that the task is running on crashes, (2) the node runs out of available memory and has to kill the task, and (3) the task encounters a runtime error. Task-based distributed systems already support transparent fault-tolerance to tasks [52], but adding collective communication support requires us to dynamically change data transfer schedule when a fraction of the tasks fail when participating in the collective communication. This is because we do not want a failed task to block collective communication, and we want to allow a recovered task to rejoin an existing collective communication.

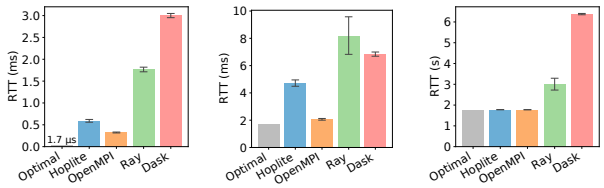
3.5.1 Broadcast. When a sender failure is detected by the receiver in broadcast, the receiver immediately locate another sender by querying the object directory again. The new sender only needs to send the remaining object that the receiver does not have. A failed task can rejoin broadcast transparently because the failed task can simply call Get on the same ObjectID to fetch the object. Implementing this feature naively would cause cyclic object transfer dependencies. For example, it is possible that two nodes try to fetch the same object from each other. It is because when the a receiver locates an alternative sender, the object directory can return the address of another node which fetches the object from the receiver. To avoid cyclic dependencies, we need to track the dependencies of Get if the sender is not the original task that creates the object. If a sender fails, the receiver only resumes if it can find another sender whose dependencies do not include the receiver itself. Figure 4c' shows the previous example if R1 fails. R2 resumes the fetch from S, and when R3 comes, R3 can fetch from R2 (Figure 4d').

3.5.2 Reduce. When a task fails during Reduce, this node is immediately removed from the tree by the coordinator, and will be replaced by the next ready source object. The guarantee is that to reduce n objects from m source objects, as long as at least n objects can be created (i.e., $m - n$ tasks can fail), Reduce will return successfully. Otherwise, Reduce completes when enough failed tasks are reconstructed by the underlying task-based system's recovery mechanisms. A failed tree node causes its parent, its grandparent, and all its ancestors to clear the reduced object. In the previous example, Figure 5b shows the adapted tree after R2 fails. If the task Reduce 6 out of 10 objects and R2 is recovered after R7 arrives, R7 replaces R2's position in the tree. (R7 can also be the rejoined R2.) R4 has to clear all the current reduced the object, because the final result should be the Reduce result of R1, R3, R4, ..., R7. Any intermediate result that contains R2's object has to be cleared. Overall, at most $\log_d n$ nodes have to clear the current object.

4 IMPLEMENTATION

The core of Hoplite is implemented using 3957 lines of C++. We provide a C++ and a Python front-end. The Python front-end is implemented using 645 lines of Python and 275 lines of Cython. We build the Python front end because it is easier to integrate with Ray [30] and other data processing libraries (e.g., Numpy [35], TensorFlow [1], PyTorch [37]). The interface between the Python front-end and the C++ backend is the same as Hoplite's API (Table 1).

We implement the object directory service using a set of gRPC [17] server processes distributed across nodes. Each directory server can push location notifications directly to an object store node. Each object store node in Hoplite is a gRPC server with locally buffered



(a) 1 KB (b) 1 MB (c) 1 GB

Figure 6: Round trip latency for point-to-point data communication on Hoplite, OpenMPI, Ray, and Dask. We also include the theoretical optimal RTT (i.e. total bytes transferred divided by the bandwidth).

objects. Upon a transfer request from a remote node (e.g., during `Get`), the node sets up a direct TCP connection to the remote node and pushes the object buffer through the TCP connection.

In our experiments, we observe that setting d to 1, 2, or n in the tree reduce algorithm is enough for our applications. When a task calls `Reduce`, Hoplite picks d from 1, 2 and n that minimizes the estimated total latency based on the network latency L, bandwidth B, and the object size S. Appendix B shows the effect of different choices of d.

5 EVALUATION

We first microbenchmark Hoplite on a set of popular traditional network primitives (e.g., broadcast, reduce, allreduce). We then evaluate Hoplite using real applications on Ray [30], including asynchronous SGD, reinforcement learning, and serving an ensemble of ML models. We also test Hoplite with synchronous data-parallel training workloads to estimate how much performance is lost if people choose to run these static and synchronous workloads on task-based distributed systems. Each application requires <100 lines of code changes, most of which are for object serialization. All experiments are done on AWS EC2. We use a cluster of 16 m5.4xlarge nodes (16 vCPUs, 64GB memory, 10 Gbps network) with Linux (version 4.15). We run every test 10 times, and we show standard deviations as error bars.

5.1 Microbenchmarks

We use two popular task-based distributed systems, Ray [30] (version 0.8.6) and Dask [44] (version 2.25), as our baselines. In addition, we compare Hoplite with OpenMPI [16] (version 3.3) and Gloo [14]. We chose OpenMPI because OpenMPI is the collective communication library recommended by AWS. We did not choose Horovod because Horovod has three backends: OpenMPI, Gloo, and NCCL. We have already tested OpenMPI and Gloo individually. We currently do not support GPU, so we do not test NCCL.

5.1.1 Point-to-Point Data Communication. We first benchmark direct point-to-point transfer. On our testbed, writing object locations to the object directory service takes 167 µs (standard deviation = 12 µs), and getting object location from the object directory service takes 177 µs (standard deviation = 14 µs).

Hoplite's point-to-point communication is efficient. We test round-trip time for different object sizes using OpenMPI, Ray, Dask, and Hoplite. Figure 6 shows the result. We also include the optimal RTT, which is calculated by object_size/bandwidth × 2.

For 1 KB and 1 MB object, OpenMPI is 1.8x and 2.3x faster than Hoplite. For 1 GB objects, Hoplite is 0.2% slower than OpenMPI. Ray and Dask are significantly slower. OpenMPI is the fastest because MPI has the knowledge of the locations of the processes to communicate. Ray, Dask, and Hoplite need to locate the object through an object directory service. Hoplite outperforms Ray and Dask because (1) Hoplite stores object contents in object directory service for objects smaller than 64 KB (§3.2) and (2) Hoplite uses pipelining (§3.3) to reduce end-to-end latency. Ray does not support pipelining, so it suffers from the extra memory copy latency in the object store. Our pipelining block size is 4 MB, and thus larger object (1 GB) has better pipelining benefits. On 1 GB object, Hoplite achieves similar performance as the underlying network bandwidth despite it has additional memory copies. This is because fine-grained pipelining successfully overlaps memory copying and data transfer.

5.1.2 Collective Communication. Next, we measure the performance of collective communication on OpenMPI, Ray, Dask, Gloo, and Hoplite, with arrays of 32-bit floats and addition as the reduce operation (if applicable). We measure the time between when the input objects are ready and when the last process finishes. For both Hoplite and Ray, we assume that the application uses a read-only `Get` to avoid the memory copy from the object store to the receiver task (§3.3). Gloo only implements broadcast and allreduce. For allreduce, Gloo supports several algorithms. We evaluated the performance for all of them, and for presentation simplicity, we only show the two algorithms with the best performance on our setup: (1) ring-chunked allreduce and (2) halving doubling allreduce.

Figure 7 shows the results for medium (1MB) to large (1GB) objects.[2] We present the results for small objects (1KB, 64KB) in Appendix A because small objects are cached in object directory service in Hoplite, and there is thus no collective communication to begin with. In summary, Hoplite achieves a similar level of performance as traditional collective communication libraries, such has OpenMPI and Gloo. Hoplite significantly outperforms Ray and Dask, because Ray and Dask do not support efficient collective communication. Gloo's ring-chunked allreduce is the fastest allreduce implementation for large objects in our tests.

Broadcast. We let one node first `Put` an object, and after the `Put` succeeds, other nodes `Get` the object simultaneously. The latency of broadcast is calculated from the time all nodes call `Get` to the time when the last receiver finishes. Hoplite and OpenMPI achieve the best performance for all object size and node configurations. This is because Ray, Dask, and Gloo do not have collective communication optimization for broadcast. Hoplite slightly outperforms OpenMPI because of fine-grained pipelining.

Gather. We let every node first `Put` an object, and after every node's `Put` succeeds, one of the nodes `Get` all the object via their `ObjectIDs`. The latency of gather is the `Get` duration. OpenMPI and Hoplite outperforms the rest for all object size and node configurations. This is because both Ray and Dask need additional memory copying between workers and the object store. Hoplite also needs additional memory copying, but the latency is masked by fine-grained pipelining between workers and the object store.

[2] OpenMPI's latency does not increase monotonically because OpenMPI chooses different algorithms on different conditions (e.g., number of nodes, whether the number of nodes is a power of two, object size).

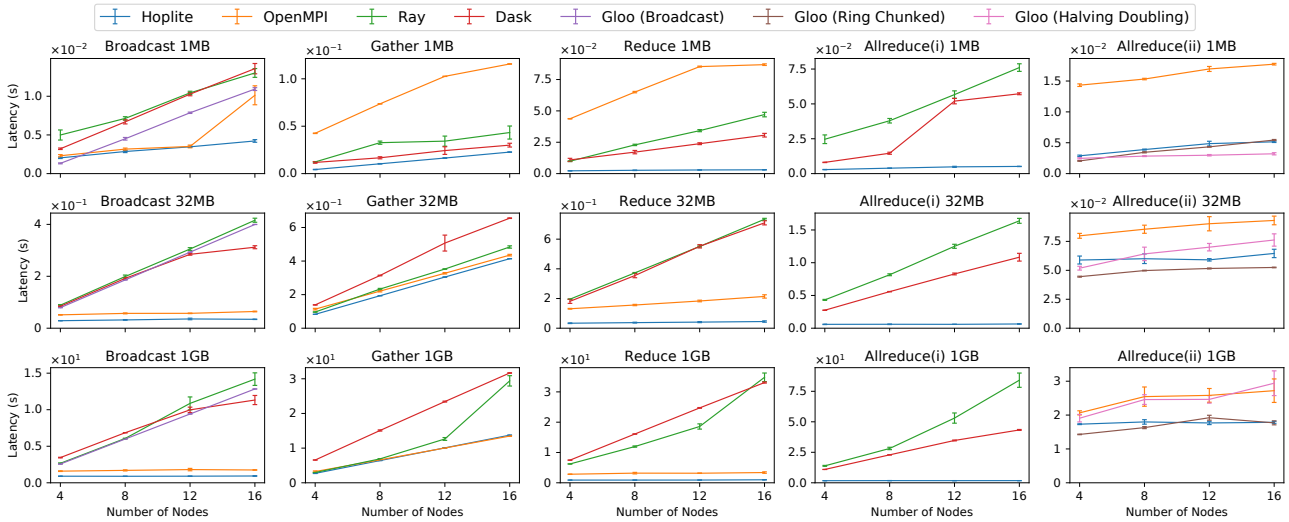
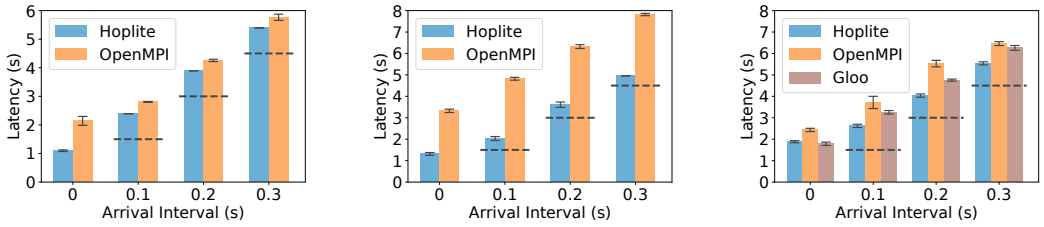


Figure 7: Latency comparison of Hoplite, OpenMPI, Ray, Dask, and Gloo on standard collective communication primitives (e.g., broadcast, gather, reduce, allreduce). To show the results more clearly, we split the results of Allreduce into two groups: group (i) includes Hoplite, Ray, and Dask, and group (ii) includes Hoplite, OpenMPI, and two different allreduce algorithms in Gloo.



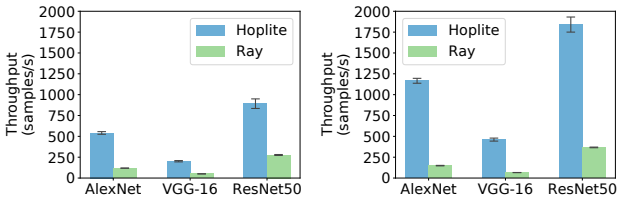
(a) Broadcast **(b) Reduce** **(c) AllReduce**

Figure 8: Latency of 1 GB object broadcast/reduce/allreduce on 16 nodes when tasks start sequentially with a fixed arrival interval. Arrival interval equals to 0 means that all the tasks start at the same time. The dashed lines denote the time the last task arrives.

Reduce. We let every node first `Put` an object, and after every node's `Put` succeeds, one of the nodes `Reduce` the objects via their `ObjectID`s to create a new `ObjectID` for the result. The node then calls `Get` to get the resulting object buffer. The latency of reduce is calculated from the time the node calls `Reduce` to the time the node has a copy of the reduce result. OpenMPI and Hoplite consistently outperform the rest for all object size and node configurations since Ray and Dask do not support collective communication. Hoplite can slightly outperform OpenMPI because of fine-grained pipelining.

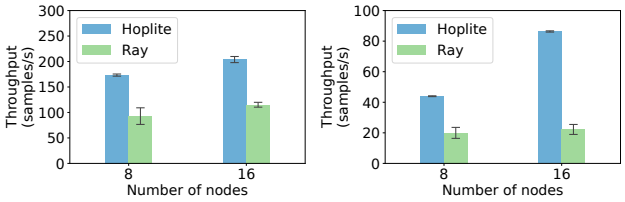
AllReduce. In Hoplite, we simply concatenate reduce and broadcast to implement allreduce. The latency of allreduce is calculated from the time a node starts to `Reduce` all the objects to the last node `Get` the reduce result. We divide the results into two groups in Figure 7. Hoplite significantly outperforms Ray and Dask because of the collective communication support of broadcast and reduce in Hoplite. Note that efficient allreduce is not our design goal since allreduce is a static and synchronous collective communication operation. However, Hoplite still achieves comparable performance with static collective communication libraries such as OpenMPI and Gloo.

5.1.3 Asynchrony. Hoplite's performance is robust even when processes are not synchronized, which is typical in task-based distributed systems. We measure broadcast, reduce, and allreduce latencies when the participating tasks arrive sequentially with a fixed arrival interval. For broadcast (Figure 8a), OpenMPI makes some progress before the last receiver arrives (§7). However, the algorithm is static (i.e. based on process *rank* [16]), while Hoplite achieves a lower latency with a dynamic algorithm that does not depend on the particular arrival order. We do not include Gloo because it does not optimize its broadcast performance (Figure 7). For reduce (Figure 8b) and allreduce (Figure 8c), both OpenMPI and Gloo have to wait until all processes are ready, while Hoplite can make significant progress before the last object is ready. This allows Hoplite to even outperform Gloo's ring-chunked allreduce when objects do not arrive at the same time.



(a) Number of Nodes = 8 (b) Number of Nodes = 16

Figure 9: Training throughput (number of training samples per second) for asynchronous SGD.



(a) IMPALA (b) A3C

Figure 10: RLlib's training throughput (number of training samples per second) on Ray and Hoplite.

5.2 Asynchronous SGD

Asynchronous stochastic gradient descent (SGD) is one way to train deep neural networks efficiently, and it usually uses a parameter server framework [11, 25, 25, 26, 26]: clients fetch the parameters from a centralized server, evaluate the parameters on its own portion of data (e.g., performing forward and backward propagation on a neural network), and send the updates (e.g., gradients) back to the server independently. The parameter server needs to broadcast parameters to and reduce from an uncertain set of workers.

Here we evaluate Hoplite with Ray's example implementation of an asynchronous parameter server [41]. We use three widely-used standard deep neural networks, AlexNet [23] (model size = 233 MB), VGG-16 [48] (model size = 528 MB), and ResNet-50 [18] (model size = 97 MB). We test two cluster configurations: 8 p3.2xlarge nodes and 16 p3.2xlarge nodes on AWS. p3.2xlarge instance has the same network performance as m5.4xlarge instance but with an additional NVIDIA V100 GPU to accelerate the execution of the neural networks. The asynchronous parameter server collects and reduces the updates from the first half of worker nodes that finish the update and broadcast the new weights back to these nodes.

We show the results in Figure 9. Hoplite improves the training throughput of the asynchronous parameter server. Comparing to Ray, it speedups training on asynchronous parameter server for 16 nodes by 7.8x, 7.0x, and 5.0x, for AlexNet, VGG-16, and ResNet-50, respectively. Ray is slow because the parameter server has to receive gradients from each worker and send the updated model to each worker one by one. This creates a bandwidth bottleneck at the parameter server. In Hoplite, these operations are optimized by our broadcast and reduce algorithms.

5.3 Reinforcement Learning

RL algorithms involve the deep nesting of irregular distributed computation patterns, so task-based distributed systems are a perfect fit for these algorithms. We evaluate Hoplite with RLlib [27], a popular and comprehensive RL library on Ray. Distributed RL algorithms can be divided into two classes: In *samples optimization* (e.g., IMPALA [13], Asynchronous PPO [46]), a centralized trainer periodically broadcasts a policy to a set of workers and gather the rollouts generated by the workers to update the model. In *gradients optimization* (e.g., A3C [29]), the workers compute the gradient with their rollouts, and the trainer updates the model with the reduced gradients from the workers.

We evaluate two popular RL algorithms, IMPALA [13] and A3C [29], one from each class. We test two cluster configurations: 8 nodes (1 trainer + 7 workers) and 16 nodes (1 trainer + 15 workers). The trainer broadcast a model to the first half workers that have finished a round of simulation (in IMPALA) or gradient computation (in A3C). We use a two-layer feed-forward neural network with 64 MB of parameters. Figure 10 shows the training throughput. Training throughput is calculated by the number of simulation traces (in samples optimization) or gradients (in gradients optimization) the RL algorithm can process in a second.

Hoplite significantly improves the training throughput of both IMPALA and A3C. Hoplite improves the training throughput of IMPALA by 1.9x on an 8-node cluster and 1.8x on a 16-node cluster. The reason Hoplite outperforms Ray is because IMPALA has to broadcast a model of 64 MB frequently to the workers. We expect more improvement when the number of nodes is higher, but we already achieve the maximum training throughput: IMPALA is bottlenecked by computation rather than communication using Hoplite with 16 nodes (15 workers). For A3C, Hoplite improves the training throughput by 2.2x on the 8-node configuration and 3.9x on the 16-node configuration. Unlike IMPALA, A3C achieves almost linear scaling with the number of workers. A3C on Ray cannot scale linearly from 8 nodes to 16 nodes because of the communication bottleneck.

5.4 ML Model Serving

Machine learning is deployed in a growing number of applications which demand real-time, accurate, and robust predictions under heavy query load [3, 10, 36]. An important use case of task-based distributed system is to serve a wide range of machine learning models implemented with different machine learning frameworks [30].

We evaluate Hoplite with Ray Serve [42], a framework-agnostic distributed machine learning model serving library built on Ray. We set up an image classification service with a majority vote-based ensemble of the following models: AlexNet [23], ResNet34 [18], EfficientNet-B1/-B2 [50], MobileNet V2 [45], ShuffleNet V2 x0.5/x1.0 [28], and SqueezeNet V1.1 [20]. We test two cluster configurations: 8 p3.2xlarge nodes and 16 p3.2xlarge nodes on AWS. For 8 nodes setting, we serve a different model on each node. For 16 nodes setting, each model is served by two different nodes and the two nodes serve the model with two different versions of weight parameters. Each query to the service includes a batch of 64 images of size 256×256. During serving, the service will broadcast the

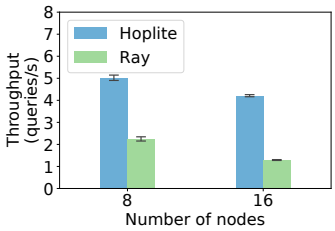


Figure 11: Ray Serve's performance (queries per second) on Ray and Hoplite for an ensemble of image classification models.

(a) Ray Server latency. **(b) Async SGD latency.**

Figure 12: Latency when a pariticipating task fails and rejoins on (a) Ray Serve and (b) async SGD.

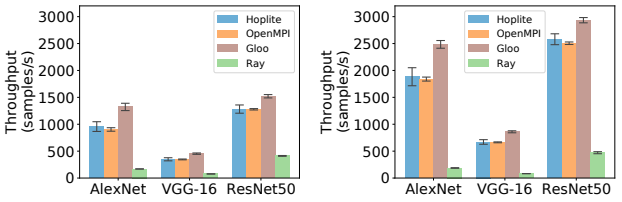
query to all the nodes to evaluate on different models, gather the classification results, and return the majority vote to the user.

We visualize the results in Figure 11. Hoplite improves the serving throughput for serving an ensemble of image classification models. Comparing to Ray, it speedups the serving throughput by 2.2x for 8 nodes and 3.3x for 16 nodes. This shows that the optimized broadcast algorithm in Hoplite helps Ray Serve to improve the serving throughput.

5.5 Fault Tolerance

We evaluate the failure recovery latency before and after we apply Hoplite to Ray. We rerun our model serving with 8 models and async SGD workloads with 6 workers, and we manually trigger a failure. We do this experiment 10 times. Figure 12 shows one particular run. The y-axis shows the latency per query (in model serving) or per iteration (in async SGD), and the x-axis shows the index of the query or the iteration. Hoplite significantly improves Ray's performance. Ray's failure detection latency is 0.58 ± 0.13 second, and after we apply Hoplite to Ray, Ray's failure detection latency increases to 0.74 ± 0.05 second. The additional 28% latency introduced by Hoplite is because Hoplite has a different failure detection mechanism. Ray detects failure by monitoring the liveness of the worker process. Hoplite detects failure by checking the liveness of a socket connection.

After the failure, Ray Serve's latency drops because it only needs to broadcast to less receivers. The latency comes back to normal after the failed worker rejoins. For Hoplite, the latency difference is negligible because of the efficient broadcast algorithm. Hoplite takes more queries between the task fails and the task rejoins. This is because Hoplite is efficient and has processed more queries during the recovery window (the time between the failure and task



(a) Number of Nodes = 8 **(b) Number of Nodes = 16**

Figure 13: Training throughput (number of training samples per second) for synchronous data-parallel training.

rejoin). In async SGD, latency for training each iteration increases in the recovery window because of the temporary loss of a worker. The difference in recovery latency (duration of the recovery window) between Ray and Hoplite is negligible because both use Ray's mechanism to reconstruct the failed task.

5.6 Synchronous Data-Parallel Training

Synchronous data-parallel training involves a set of workers, each runs on a partition of training data, and the workers synchronize the gradients each round using allreduce [15]. Speeding up synchronous data-parallel training workloads is not our design goal, and they do not require the flexibility provided by task-based systems. Instead, they can run directly on specialized distributed systems that are optimized for static and synchronous workloads (e.g., TensorFlow [1], PyTorch [37]). These systems rely on efficient allreduce implementations in traditional collective communication frameworks (e.g., OpenMPI, Gloo).

However, an interesting question to ask is how much performance developers have to pay if they choose to run these static and synchronous workloads on task-based distributed systems. Our cluster setup is the same as the asynchronous parameter server experiment. In addition to Ray, we evaluate Gloo and OpenMPI. We evaluate the Gloo baseline through PyTorch, which chooses ring-chunked allreduce as its choice for Gloo's algorithm.

We show the results in Figure 13. Hoplite significantly improves the synchronous data-parallel training for Ray. Ray is slower than Hoplite, OpenMPI, and Gloo, with the similar reason as in asynchronous parameter server. Hoplite achieves similar speed with OpenMPI. However, Hoplite is 12-24% slower than Gloo. This is expected because ring-allreduce is more bandwidth efficient than the tree-reduce plus broadcast in Hoplite.

6 DISCUSSION

Garbage collection. Hoplite provides a `Delete` call (Table 1) that deletes all copies of an object from the store. This can be used to garbage-collect an object whose `ObjectID` is no longer in scope in the application. However, it is still the task framework or application's responsibility to determine when `Delete` can and should be called, since only these layers have visibility into which `ObjectIDs` a task has references to. The guarantee that Hoplite provides is simple: when `Put` is called on an `ObjectID`, the object copy that is created will be pinned in its local store until the framework calls `Delete` on the same ID. This guarantees that there will always be at least one available location of the object to copy from, to fulfill

future `Get` requests. Meanwhile, Hoplite is free to evict any additional copies that were generated on other nodes during execution, to make room for new objects. The overhead of eviction is very low, since Hoplite uses a local LRU policy per node that considers all unpinned object copies in the local store.

Framework's Fault tolerance. Hoplite ensures that collective communication can tolerate task failure. A task-based distributed system has a set of control processes that can also fail, and they usually require separate mechanisms to tolerate failures. For example, the object directory service can fail and requires replication for durability. These failures are handled by the underlying framework independent of whether Hoplite is used.

Network Heterogeneity. The design of Hoplite assumes that the network capacity between all the nodes is uniform. Accommodating heterogeneous network can achieve higher performance (e.g., using high bandwidth nodes as intermediate nodes for broadcast, fetching objects from a node which has lower latency). This can be done by monitoring network metrics at run time. We do not need this feature for our use cases because our cloud provider ensures uniform network bandwidth between our nodes.

Integration with GPU. Hoplite currently does not support pipelining into GPU memory. If training processes need to use GPU, the application has to copy data between GPU and CPU memory. In the future, we want to extend our pipelining mechanism into GPU memory.

7 RELATED WORK

Optimizing data transfer for cluster computing. Cluster computing frameworks, such as Spark [53] and MapReduce [12], have been popular for decades for data processing, and optimizing data transfer for them [7–9, 24, 39] has been studied extensively. AI applications are particularly relevant because they are communication-intensive, and traditional collective communication techniques are widely-used [14, 47, 51]. Pipelining is also a well-known technique to improve performance [33, 38]. Our work focuses on improving task-based distributed systems [19, 30, 44]. Applications on these frameworks have dynamic and asynchronous traffic patterns. To the best of our knowledge, Hoplite is the first work to provide efficient collective communication support for task-based distributed systems.

Using named objects or object futures for data communication. Using named objects or object futures for data communication is not new. In serverless computing, tasks (or functions) cannot communicate directly. As a result, tasks communicate through external data stores [40], such as Amazon S3 [2] or Redis [43]. There, the storage and compute servers are disaggregated, and computer servers do not directly communicate. We target a standard cluster computing scenario, where data is directly transmitted between compute servers. Object futures are a useful construct for expressing asynchronous computation. Dask, Ray, Hydro, and PyTorch [37] all use futures to represent results of remote tasks. Our work is complementary to them, showing that efficient collective communication can co-exist with named objects or object futures.

Asynchronous MPI. MPI supports two flavors of asynchrony. First, similar to a non-blocking POSIX socket, MPI allows an application to issue asynchronous network primitives and exposes an `MPI_Wait` primitive to fetch the result. Second, depending on the MPI implementation, some collective communication primitives can make some progress with a subset of participants. For example, in `MPI_Bcast`, the sender generates a static broadcast tree. If the receivers arrive in order from the root of the tree to the leaves of the tree, the receivers can make significant progress before the last receiver arrives. If not, then a receiver must wait until all its upstream ancestors are ready before making any progress (evaluated in Figure 8). In Hoplite, the broadcast tree is generated dynamically at runtime, so the arrival order does not matter. In addition, asynchronous MPI primitives still require applications to specify all the participants before runtime. In Hoplite, the communication pattern can be expressed dynamically and incrementally, allowing Hoplite to work with existing task-based distributed systems.

Collective communication in other domains. Optimizing data transfer has been studied extensively in other domains. Application-level multicast [5, 6] for streaming video on wide-area networks. IP multicast [21] enables a sender to send simultaneously to multiple IP addresses at the same time. These work mostly focus entirely on multicast rather than general-purpose collective communication in distributed computing frameworks.

8 CONCLUSION

Task-based distributed computing frameworks have become popular for distributed applications that contain dynamic and asynchronous workloads. We cannot directly use traditional collective communication libraries in task-based distributed systems, because (1) they require static communication patterns and (2) they are not fault-tolerant. We design and implement Hoplite, an efficient and fault-tolerant communication layer for task-based distributed systems that achieves efficient collective communication. Hoplite computes data transfer schedules on the fly, and even when tasks fail, Hoplite can allow well-behaving tasks to keep making progress while waiting for the failed tasks to recover. We port a popular distributed computing framework, Ray, on top of Hoplite. Hoplite speeds up asynchronous SGD, RL, model serving workloads by up to 7.8x, 3.9x, and 3.3x, respectively. Hoplite's source code is publicly available (https://github.com/suquark/hoplite). This work does not raise any ethical issues.

ACKNOWLEDGEMENTS

We thank our shepherd Kai Chen and the anonymous reviewers for their insightful feedback. We also thank Hong Zhang and many others at the UC Berkeley RISELab for their helpful discussion and comments. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Danyang Zhuo is supported by an IBM Academic Award.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Amazon S3 2020. Amazon S3. Object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/s3/.

[3] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[4] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[5] M. Castro, P. Druschel, A. . Kermarrec, and A. I. T. Rowstron. 2002. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications* 20, 8 (Oct 2002), 1489–1499. https://doi.org/10.1109/JSAC.2002.803069

[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. 2003. SplitStream: High-Bandwidth Multicast in Cooperative Environments. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 298–313. https://doi.org/10.1145/1165389.945474

[7] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 393–406. https://doi.org/10.1145/2785956.2787480

[8] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. 2011. Managing Data Transfers in Computer Clusters with Orchestra. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 98–109. https://doi.org/10.1145/2043164.2018448

[9] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/2619239.2626315

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[12] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1, 107–113. https://doi.org/10.1145/1327452.1327492

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*. PMLR, 1407–1416.

[14] Gloo 2020. Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[16] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2005. Open MPI: A flexible high performance MPI. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 228–239.

[17] gRPC 2020. gRPC. https://grpc.io/.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Hydro 2020. Hydro. https://github.com/hydro-project.

[20] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[21] IPMulticast 2020. IP Multicast Technology Overview . https://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/ip_multicast/White_papers/mcst_ovr.html.

[22] keynote 2020. Keynote: Building a Fusion Engine with Ray. https://ray2020.sched.com/event/eGOL/keynote-building-a-fusion-engine-with-ray-dr-charles-he-chief-architect-of-storage-and-compute-ant-group.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[24] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-Driven Bandwidth Guarantees in Datacenters. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 467–478. https://doi.org/10.1145/2740070.2626326

[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[26] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, Vol. 6. 2.

[27] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*. PMLR, 3053–3062.

[28] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. 116–131.

[29] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 1928–1937.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[31] MPICH 2020. MPICH. https://www.mpich.org/.

[32] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing.

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[34] NCCL 2020. The NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[35] NumPy 2020. NumPy. https://numpy.org/.

[36] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[38] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3341301.3359642

[39] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-Distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 421–434. https://doi.org/10.1145/2785956.2787505

[40] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[41] Ray Parameter Server 2020. Parameter Server. https://ray.readthedocs.io/en/latest/auto_examples/plot_parameter_server.html.

[42] Ray Serve 2021. Ray Serve. https://docs.ray.io/en/master/serve/.

[43] Redis 2020. Redis. https://redis.io/.

[44] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Citeseer.

[45] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG]

[48] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[49] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. 2020. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[50] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

[51] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 172–186. https://proceedings.mlsys.org/paper/2020/file/43ec517d68b6edd3015b3edc9a11367b-Paper.pdf

[52] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 338–352. https://doi.org/10.1145/3341301.3359653

[53] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, and et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A MICROBENCHMARKS ON SMALL OBJECTS

We present the microbenchmarks for multiple collective communication primitives for small objects (1KB, 32KB) in Figure 14. Note that Hoplite stores object contents in object directory service for objects smaller than 64 KB (§3.2), so there is no collective communication for Hoplite. Again, we compare with Ray, Dask, OpenMPI, and Gloo. We do not compare with Horovod for the same reason that Horovord has three backends: OpenMPI, Gloo, and NCCL. We have already compared with OpenMPI and Gloo. NCCL is for GPU, and Hoplite currently does not support GPU.

Hoplite is the best or close to the best among all these alternatives. Gloo has the best performance for broadcast and allreduce. Hoplite is more efficient than Ray, and Dask because Hoplite uses stores the object data directly in object directory service.

B ABLATION STUDY ON REDUCE TREE DEGREE

Here we study the choice of d in the AWS EC2 setting (§5). The best choice of d depends on network characteristics, the size of the object to reduce, and the number of participants. We compare three choices of d: 1 (a single chain), 2 (a binary tree), and n (a root connects everyone else). The results are in Figure 15. As expected from our analysis in (§3.4), when the object size is small, $d = n$ is the best because the main bottleneck is the network latency. When the object size is medium (256KB, 1MB), $d = n$ becomes unstable for reduce. We suspect that this is due to incast or due to gRPC characteristics. When object size is 4MB or 8MB, we need to choose between $d = 1$ and $d = 2$ based on the number of participants. This is because both network latency and network throughput can be a bottleneck in tree reduce. When object size is 16MB or larger, we choose $d = 1$ to mitigate the throughput bottleneck in reduce.

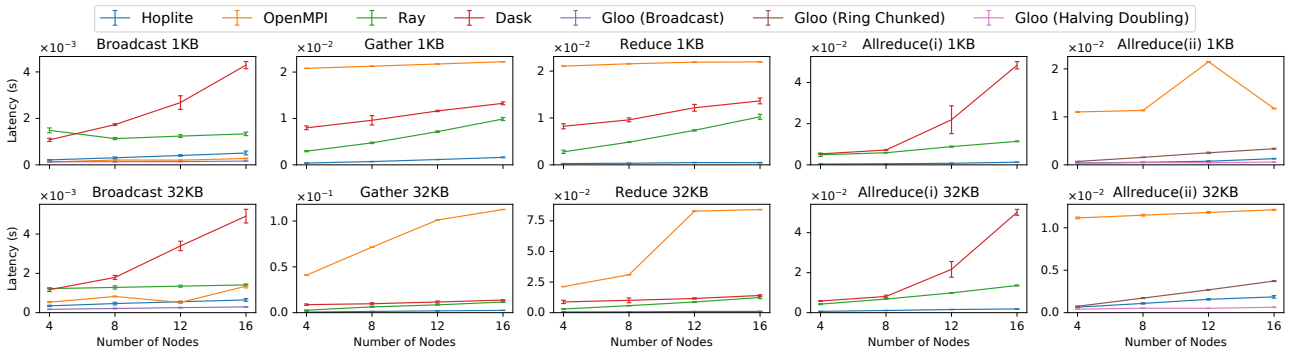


Figure 14: Latency comparison of Hoplite, OpenMPI, Ray, Dask, and Gloo on standard collective communication primitives (e.g., broadcast, gather, reduce, allreduce) on 1KB and 32KB objects. To show the results more clearly, we split the results of Allreduce into two groups: group (i) includes Hoplite, Ray, and Dask, and group (ii) includes Hoplite, OpenMPI, and two different allreduce algorithms in Gloo.

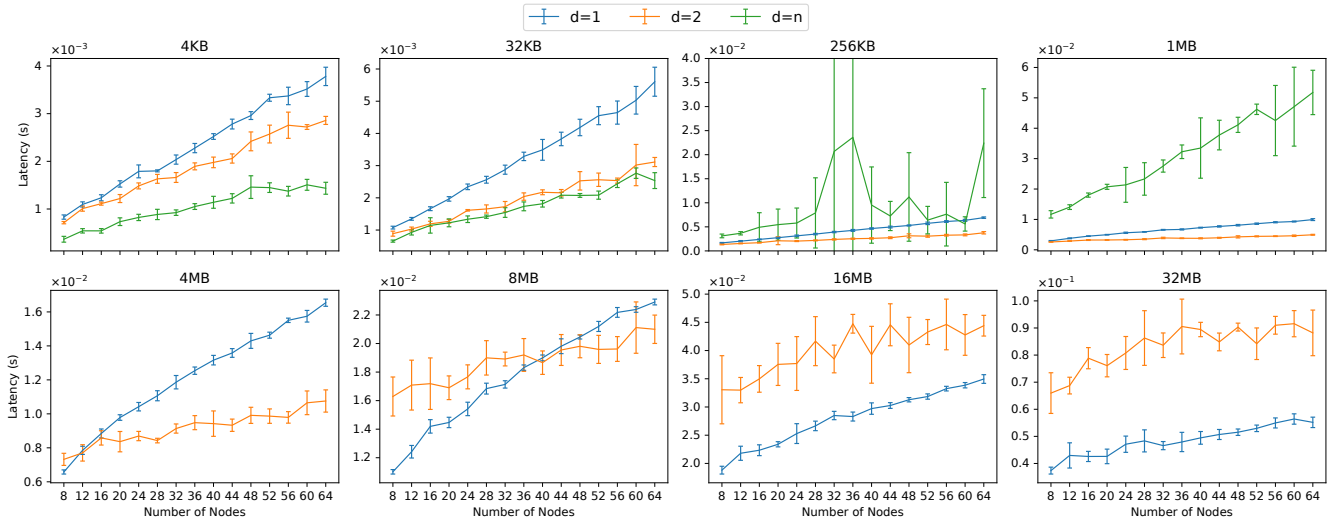


Figure 15: Ablation study of reduce latency on the reduce tree degree d with different object size and number of participants.