Elastic Horovod

Auto-Scaling Distributed Deep Learning

Justification

The achilles heel of Horovod and other allreduce-based distributed training systems has been the inability to dynamically resize the number of workers in a job at runtime, due to failure or a change in available resources. This was less of an issue in the HPC world, where cluster resources are mostly static and hardware is generally very reliable.

In the cloud / data center world, the picture is a bit different. Hardware is generally commodity, meaning failures are more common. Resource demand is more unpredictable, with systems being highly multi-tenant and used for a variety of workloads of varying size and priority. Jobs are more cost-sensitive, meaning users are often willing to trade guaranteed hardware availability for significant reductions in

training costs.

Giving Horovod the ability to scale up and down *elastically* at runtime solves many customer challenges at once, including:

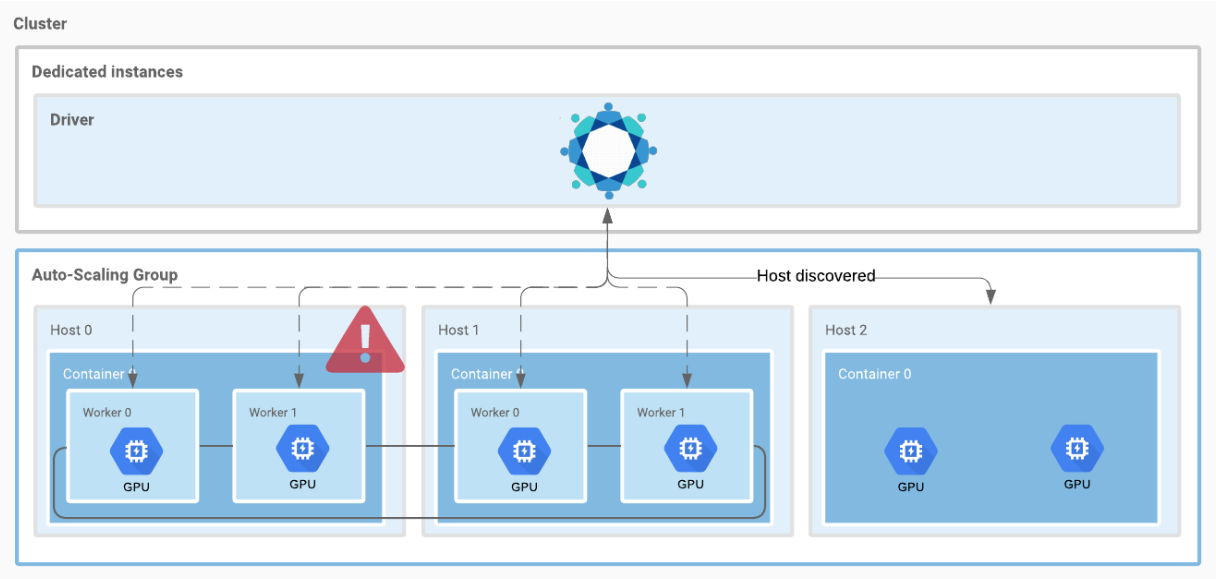
1. **Fault tolerance.** Even when running with a fixed number of processes, individual hosts can sometimes fail. The likelihood of such a failure only increases as the number of workers increases, making training at very large scales a challenge even on otherwise reliable hardware. When jobs fail, it requires restarting from the last checkpoint, which can cost upwards of an hour worth of compute. For large pipelines that train large models partitioned by geographical region or user segment, this can result in large amounts of wasted compute resources.
2. **Autoscaling spot and preemptible instances.** Many cloud providers offer so-called “spot” or “preemptible” instances that offer significant cost reductions over dedicated instances. The tradeoff is that instances can be given and taken away at any point during the job. As new instances become available, jobs will want to add them to the worker topology, and as workers are removed, training should continue with minimal interruption.
3. **Dynamic resource reallocation.** In on-prem data centers, there is limited capacity for accelerated hardware such as GPU instances. Between times of the day, days of the week, and months in the year, demand will fluctuate considerably. In the current system, users must pick an exact resource quantity upfront. Even if they pick an optimal number of resources at the time their job starts, it is likely that this amount will no longer be optimal deeper into training. During times of low demand, resources should be given freely to users to speed up their training jobs. As more users come online, those resources should be redistributed so that other users can make progress as well. This removes guesswork from users and prevents long-running training jobs from locking out other users.

The availability of spot instances in cloud offers pricing discounts as high as 80 - 90% raise the possibility of training 5 - 10x faster for the same price. After switching Horovod integration tests to use spot instances in AWS, we saw a reduction in costs from $0.90 per GPU per hour to $0.29, or a nearly 70% cost reduction.

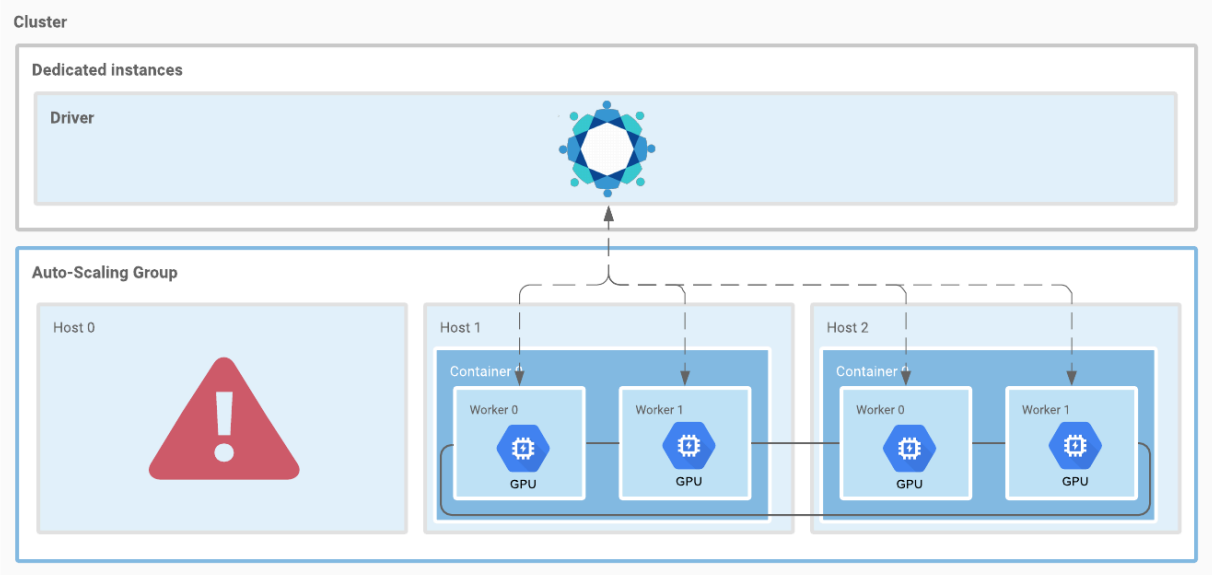
Overview

In standard Horovod operation, all processes — including the **horovodrun** driver process — are executed on a homogenous GPU hardware. The driver process launches some of the worker processes on its own host.

In Elastic Horovod, the driver instead executes on a separate dedicated CPU instance, while worker processes execute in a separate GPU auto-scaling group.



When a failure occurs and/or new workers are discovered, as illustrated in the figure above, the driver will be responsible for reforming the network by excluding failed hosts, launching new processes on added hosts, and performing rendezvous for surviving hosts.



In this approach, surviving workers never need to shut down due to an error in one of the other workers. As such, training can resume from in-memory state as opposed to needing to restore from disk (i.e., checkpoint).

The net effect is that a failure only costs at most one step (i.e., batch) of training as opposed to an entire checkpoint (i.e., epoch), and can be handled automatically without the need for specialized wrapper scripts.

User Experience

When to use Elastic Horovod

Elastic training provides a number of significant benefits, namely fault tolerance and auto-scaling capabilities. If your job runs at high scale for long durations (several hours), runs in a cloud environment where instance costs are a significant consideration, or runs on prem with fragmented GPU utilization, then elastic training can provide significant benefits.

There are also a number of potential downsides to consider:

- **Increased training script complexity.** As discussed in the *Training Script* section below, we need to ensure that the training script is structured so that workers can be added and removed from the process yet continue from the same point. However, once these changes are made, Horovod can continue to run in elastic or non-elastic mode without further modifications. In many cases, it's worth structuring the code for elastic from the beginning.
- **Increased model variance.** The outcome of the training process becomes unstable as the number of reset events (worker added or removed) increases. Because learning rates can change without warning, it can be difficult to tune learning rate schedules correctly, which can lead to model performance degradation if not properly managed. We discuss ways to work around this in the *Reliability and Variance* section below.
- **Increased per batch iteration time.** In elastic mode, we need to track additional state, including the last set of process state variables used for synchronizing in the event of a reset event. This results in a deep copy of several objects each step that can affect training throughput. Additionally, when partitioning data across workers, we need to track additional state (rows / files processed), which must be synced between workers each step. Users can trade off between *committing* state less frequently to speed up batch iteration time at the cost of needing to rollback to an older saved state in the event of a reset event.

While training script complexity is a fixed one-time cost, model variance and batch iteration time are things that need to be carefully considered by the user. By default, we will set a reasonably high limit on max resets, and commit state once per batch, but provide the tools for users to tune the reset limit if model variance becomes an issue, and only commit every N batches to speed up batch iteration time. In this way, users can adapt to the specific bottlenecks of their training process.

Training Script

To take advantage of Elastic Horovod in a training script, the user will only need abide by three principle constraints:

1. Retryable code must be wrapped in a function decorated with the **@hvd.elastic.run** decorator.
2. State to be kept in sync among the workers must be put into a **hvd.elastic.State** object. The state object must be passed to the retryable training function, and state must be *committed* at the end of each step. Any parameters that need to adjust when the state is reset (e.g., learning rate) need to be registered as callbacks to the state object.
3. No collective communication operations (e.g., allreduce, allgather, broadcast) can be made outside of the retryable training function.

Example in PyTorch:

```
import torch
import horovod.torch as hvd

hvd.init()
torch.cuda.set_device(hvd.local_rank())

model = ...
```

```
dataset = ...

@hvd.elastic.run
def train(state):
    for state.epoch in range(state.epoch, args.epochs + 1):
        dataset.set_epoch(state.epoch)
        dataset.set_batch_idx(state.batch_idx)
        for state.batch_idx, (data, target) in enumerate(dataset):
            state.optimizer.zero_grad()
            output = state.model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            state.optimizer.step()
            state.commit()

optimizer = optim.SGD(model.parameters(), lr * hvd.size())
optimizer = hvd.DistributedOptimizer(optimizer)

def on_state_reset():
    # adjust learning rate on reset
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr * hvd.size()

state = hvd.elastic.TorchState(model, optimizer, epoch=1, batch_idx=0)
state.register_reset_callbacks([on_state_reset])
train(state)
```

Example in TensorFlow Keras:

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd

hvd.init()

config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
tf.keras.backend.set_session(tf.Session(config=config))

model = ...
dataset = ...

opt = tf.keras.optimizers.Adadelta(1.0 * hvd.size())
opt = hvd.DistributedOptimizer(opt)

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=opt,
```

```
                metrics=['accuracy'])

def on_state_reset():
    tf.keras.backend.set_value(model.optimizer.lr, lr * hvd.size())

state = hvd.elastic.KerasState(model, epoch=1, batch_idx=0)
state.register_reset_callbacks([on_state_reset])

callbacks = [
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),
    hvd.callbacks.MetricAverageCallback(),
    hvd.elastic.callbacks.CommitStateCallback(state),
    hvd.elastic.callbacks.UpdateEpochStateCallback(state),
    hvd.elastic.callbacks.UpdateBatchStateCallback(state),
]

if hvd.rank() == 0:
    callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))

@hvd.elastic.run
def train(state):
    state.model.fit(dataset,
                    batch_size=batch_size,
                    callbacks=callbacks,
                    steps_per_epoch=train_batches // hvd.size(),
                    epochs=epochs - state.epochs,
                    verbose=1 if hvd.rank() == 0 else 0))

train(state)
```

The purpose of the **state** and **hvd.elastic.run** decorator are to create a synchronization point so that training can rollback to the last step when an error occurs, and new workers that join late in the training process can be brought to the same point as the other workers. From this synchronization point, workers can proceed in lock-step together until another failure occurs or worker is added. This is why no collective calls can be made outside the hvd.elastic.run function, as they cannot be synchronized among workers.

Data Access

For data parallel training, there are broadly two approaches to distributed data access among the N workers:

1. **Dataset Partitioning**. Each epoch, the dataset is randomly divided into N equal shards (remainders are typically discarded to enforce this, or Horovod's join operator can be used). Each shard is shuffled and batched independently. This ensures every training example is processed at most once per epoch.
2. **Random Sampling**. Each epoch, the data is logically shuffled and batched on each worker independently. Every step, each of the N workers will independently sample a batch from the dataset with replacement. This approach does not ensure that every sample is processed at most

once, but does not require any logic unique to distributed training.

Both approaches are commonly used in practice, though more advanced users will typically opt for dataset partitioning as it provides more robust theoretical guarantees about model convergence in a distributed setting.

Elastic Horovod users opting for a random sampling approach will not need to make any changes to their existing training scripts for data access. However, dataset partitioning systems will need to be changed to accommodate fault tolerant training. These changes will need to be made on a framework-by-framework basis.

Running Jobs

Launching a training job with Elastic Horovod will use the same **horovodrun** command-line and API interfaces used to launch ordinary Horovod jobs today. Existing args will be supported, but elastic training will additionally support four new command-line args in the initial release:

1. **--min-np**: Minimum number of processes running for training to continue. If the number of available processes dips below this threshold, then training will wait for more instances to become available (up to a timeout). If unspecified, then this will default to --num-proc/-np, meaning that the number of instances at the start of training is also the minimum allowed.
2. **--max-np**: Maximum number of training processes, beyond which no additional processes will be created. If not specified, this will also default --num-proc/-np. This information is useful for users who wish to train with a fixed learning rate (and use local gradient aggregation) or for determining dynamic data partitioning strategies.
3. **--host-discovery-script**: An executable script that will print to stdout every available host (one per newline character) that can be used to run worker processes. Optionally specify the number of slots on the same line as the hostname as: "hostname:slots". If slots are not provided, they can be inferred to be homogenous from --slots-per-host.
4. **--slots-per-host**: Number of worker processes to launch on each host. In most cases this should be equal to the number of GPUs on each host in a homogenous worker pool. If slots are provided by the output of the host discovery script, then that value will override this parameter. Will default to 1 process per host if unspecified and not ascertainable through other means.

Not all of these parameters are required to be used together. Both **--min-np** and **--max-np** can be omitted as **--num-proc** is still required (the number of worker processes that must be available for training to start). The **--host-discovery-script** will almost always be used, with the exception of running Elastic Horovod purely for fault tolerance (no auto-scaling). If **--min-np** is used without **--host-discovery-script**, then elastic mode will be enabled. But if both are absent, Horovod will run in standard mode. Note that **--max-np** cannot be used without **--host-discovery-script**, and **--slots-per-host** can be used without elastic mode (to infer slot numbers from hostnames).

Example 1 - Elastic with Auto-Scaling:

horovodrun -np 4 --min-np 2 --max-np 8 --host-discovery-script discover.sh python pytorch_mnist_elastic.py

Example 2 - Elastic without Auto-Scaling (discovery only):

horovodrun -np 2 --host-discovery-script discover.sh python pytorch_mnist_elastic.py

Example 3 - Elastic without Auto-Scaling (fault tolerance only):

horovodrun -np 2 --min-np 2 -H worker-0,worker-1 --slots-per-host 2 python pytorch_mnist_elastic.py

Cloud Provider Integration

Though the host discovery script approach is the most general — allowing Elastic Horovod to run on bare metal as well as any containerized cloud environment — it can be redundant to ask every user to write the same discovery scripts for common cloud providers like AWS, Azure, GCP, and IBM.

Special command line arguments will be exposed for these providers that can be used in place of the host discovery script. Adding a new cloud provider can be supported by implementing the common host discovery interface and adding the proper command line arguments to hook up the necessary authorization to use the cloud provider APIs.

For example, on AWS, instance information can be obtained via the auto-scaling group, exposed through a simple Python API[1]. The obtained instance IDs can then be mapped to IP addresses using the EC2 client[2]. This approach doesn't require the user to pre-configure their job, but requires platform logic be implemented in Horovod itself.

Reliability and Variance

One of the major reasons synchronous distributed SGD frameworks like Horovod are used in practice over alternatives like asynchronous parameter servers is that training is largely deterministic. There is minimal variance in the training process, so training the same model with the same config should result in largely the same trained model.

In the elastic domain, this guarantee is harder to make. Learning rate can fluctuate dynamically at training time, and learning rate schedules can be disrupted from frequent changes to the worker set.

In practice, users have observed the following:

- For some models, it may be desirable to set a hard *limit* on the number of resets allowed before the job fails. In such cases, this limit would be decided based on past experiments using a variety of settings on the particular model before elastic training is enabled.
- Deep neural network models can be very tolerant to resets, given sufficiently large datasets. This has been demonstrated empirically for classification, detection, image segmentation, and machine translation models.
- It is important to implement a learning rate warmup following a reset event.

We expose an additional parameter to help ensure reduce variance and ensure model training reliability:

- **--max-resets**: Hard limit on the number of reset events (rendezvous) that can occur before the job fails (raises MaxResetLimitExceededError). Defaults to unlimited.

Using the State object, users can also configure the frequency with which Horovod will check for added hosts and reset:

```
state.set_check_hosts_added_delay(commits)
```

Architecture

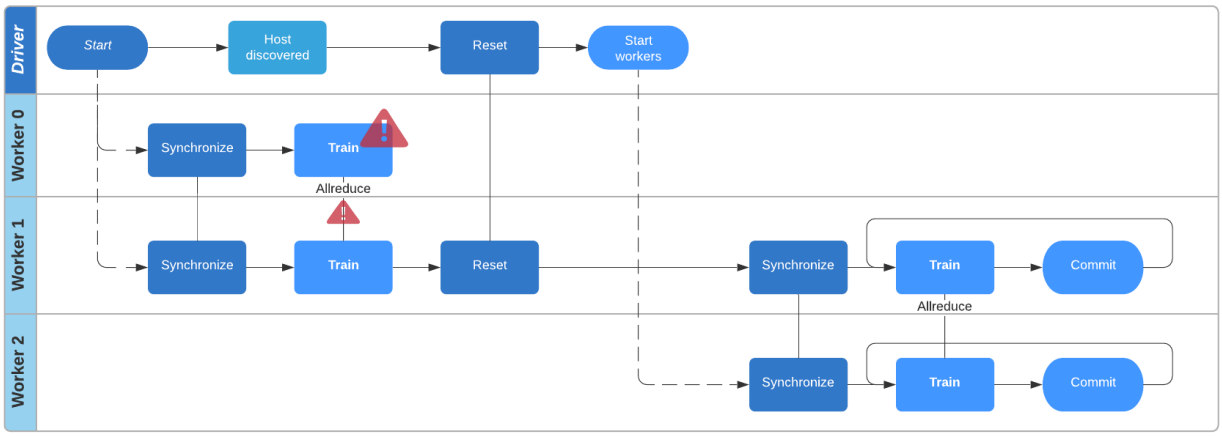
The major architectural differences in Elastic Horvod from existing Horovod are as follows:

---

[1] https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/autoscaling.html
[2] https://docs.aws.amazon.com/code-samples/latest/catalog/python-ec2-describe_addresses.py.html

- All collective operations are coordinated within a hvd.elastic.run function.
- State is synchronized between workers before the user's training function is executed.
- Worker failure or worker added events will result in a reset event on other workers.
- Reset events act as barriers to:
    - Determine whether the job should continue based on worker exit codes.
    - Blacklist failing hosts.
    - Launch workers on new hosts.
    - Update the rank information on existing workers.
- State is synchronized following a reset event.

An example timeline is shown below:



In the above figure, training starts with 2 workers. The first worker has a system failure resulting in an exception being raised on the second worker. This causes a reset event where the driver launches a new process on a newly discovered host, then training resumes after worker synchronize state.

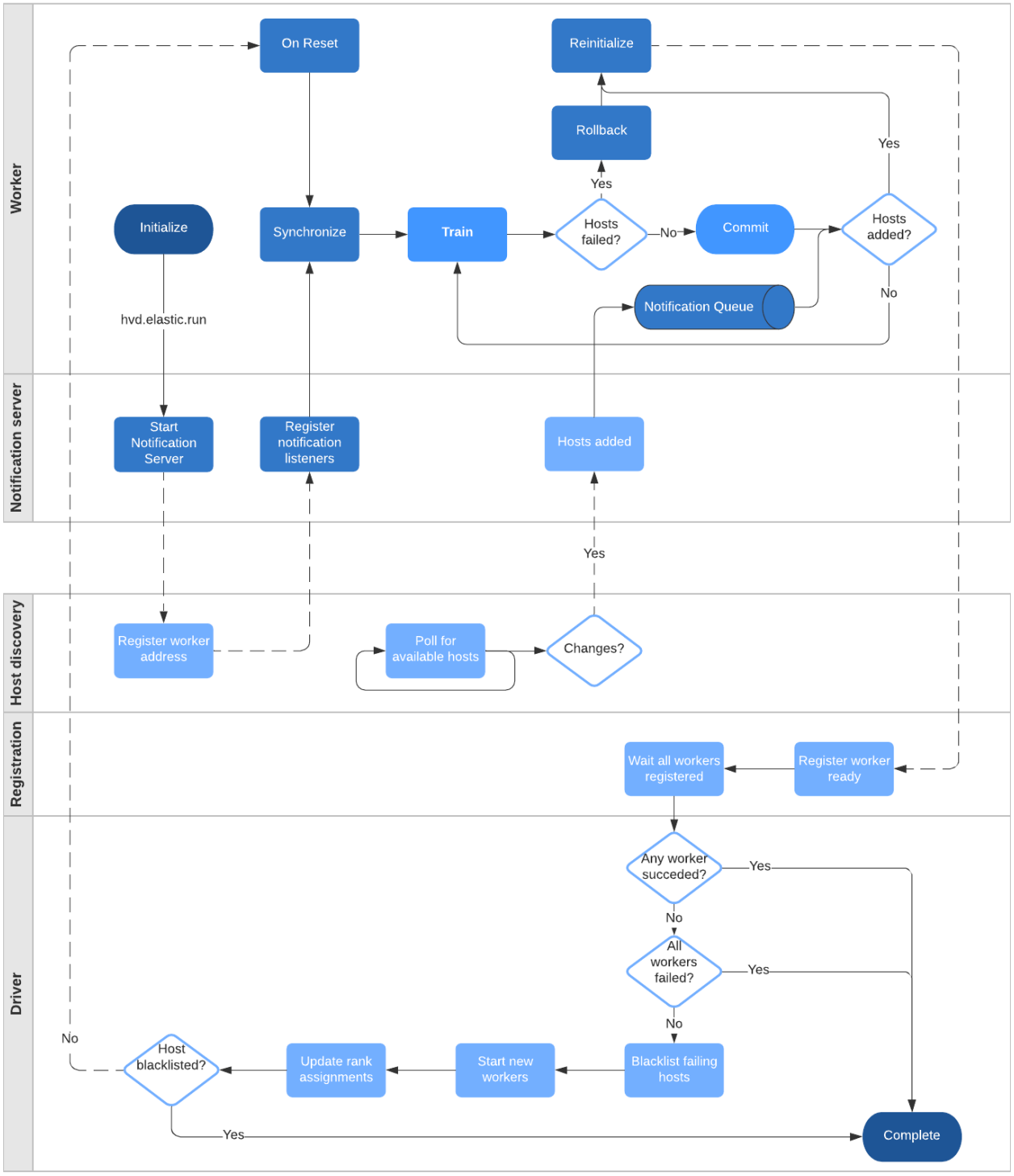
Worker

The **worker** processes are where the user code is executed in parallel across multiple hosts / GPUs. In Horovod, we execute one worker process per GPU in most cases.

Under the hood, the hvd.elastic.run decorator process looks like this (pseudo-code):

```
def hvd_elastic_run(state, *args, **kwargs):
    notification_manager.init()
    notification_manager.register_listener(state)
    while True:
        state.sync()
        try:
            return train(state, *args, **kwargs)
        except HorovodInternalError:
            state.restore()
        except WorkersAvailableException:
            pass
        reinitialize()
```

The full process from the perspective of an individual worker is shown below:



1. **Initialize.** Initialization including any non-retryable startup logic (model definition, dataset preprocessing, wrapping the optimizer, etc.).
2. **Start notification server.** Entering the **hvd.elastic.run** function, a notification server is started to receive push notifications from the driver when a new host is available. This will also do a one-time pull for any outstanding hosts that were added in the gap between process start and server initialization. A push-based model from driver to worker is favored here to avoid

bottlenecking the training process each step on the worker sending an HTTP request to the driver (host adds should be infrequent).

3. **Register worker address.** Update a map within the host discovery thread on the driver for routing new hosts to the active workers.
4. **Register notification listeners.** The state object registers a callback with the notification server, so that every time a host is pushed it can be consumed by the state object during the next commit.
5. **Synchronize.** State is synchronized between the workers, broadcasted from worker 0. This includes model weights, optimizer state (momentum, etc.), epoch number, batch index, etc.
6. **Train.** User-provided training function is executed with the state object and any additional user parameters.
7. **Hosts failed?** If an internal error occurs during an allreduce or other collective op within the training function, then it will be caught in hvd.elastic.run.
    a. **Rollback.** State will be rolled back to the last committed checkpoint to undo any partial updates (for example, partially applied gradients).
8. **Commit.** Any changes made to variables of the state object will be deep-copied and persisted for use in the event of a rollback.
9. **Hosts added?** Following a commit, an exception will be raised if workers have been added and it will be caught in hvd.elastic.run. This event is handled similar to a host failure, but does not result in a rollback. This frequency of checking for host failures can be configured to balance between adding new hosts and making progress with a fixed set of workers. If host add events are more common, it may be worth adding a delay between these checks (i.e., once every N batches).
    a. If no hosts were added or other failures occurred, continue the training process.
10. **Reinitialize.** Shutdown the Horovod context and recreate it, resulting in each worker pinging the driver to perform rendezvous, updating the rank and size information in the process. Older workers will take on lower ranks to ensure that the oldest active worker is assigned rank 0, responsible for broadcasting state to the other workers during the synchronize step.
11. **Register worker ready.** The driver uses a registration system to record the status of each worker (ready, success, failure) following a reinitialization. A worker that is still active will register itself as *ready*, meaning it is awaiting new rank and size information.
12. **Wait all workers registered.** The registration system features a *barrier* that waits for all processes to register (a timeout will fail the entire job). Once all workers have registered, a callback is initiated to determine whether the job should continue.
    a. **Any worker succeeded?** If any worker succeeded, then the job will complete, even if some of the workers are still in the ready state. Unless all the workers are collectively in the success state, then the overall job will fail.
    b. **All workers failed?** If all workers failed, then the job will also fail. This accounts for problems with the user code where the training script is incapable of training successfully. We do not want to continue infinitely retrying in this instance. If only some of the workers failed, then the job can still continue to train.
13. **Blacklist failing hosts.** In some cases, a host may fail but still appear to be *alive* from the perspective of the host discovery thread. This can happen, for example, in the case of faulty hardware (bad GPU, memory, disk, etc.). When a worker fails, we proactively *blacklist* the host to ensure jobs are not placed on that machine until after a *cooldown* period has elapsed. This cooldown period will increase exponentially as more failures occur on the host.
14. **Start new workers.** Any hosts that were added will at this point have new worker processes launched on them. This follows a standard recalculation of the rank and size assignments that ensures older workers receive lower ranks.
15. **Update rank assignments.** Ready workers will receive their updated rank assignments following the start of the new workers.
    a. **Host blacklisted?** If the worker is ready but the host was blacklisted (another worker co-located on the same machine failed), then the worker will receive a special rank assignment of -1 indicating that the worker should shut itself down.
16. **On Reset.** Following the rendezvous and reconstruction of the Horovod context, any callbacks registered with the state object (for example, learning rate adjustment) will occur. After this,

processing will resume from (5) with a synchronization of the state.

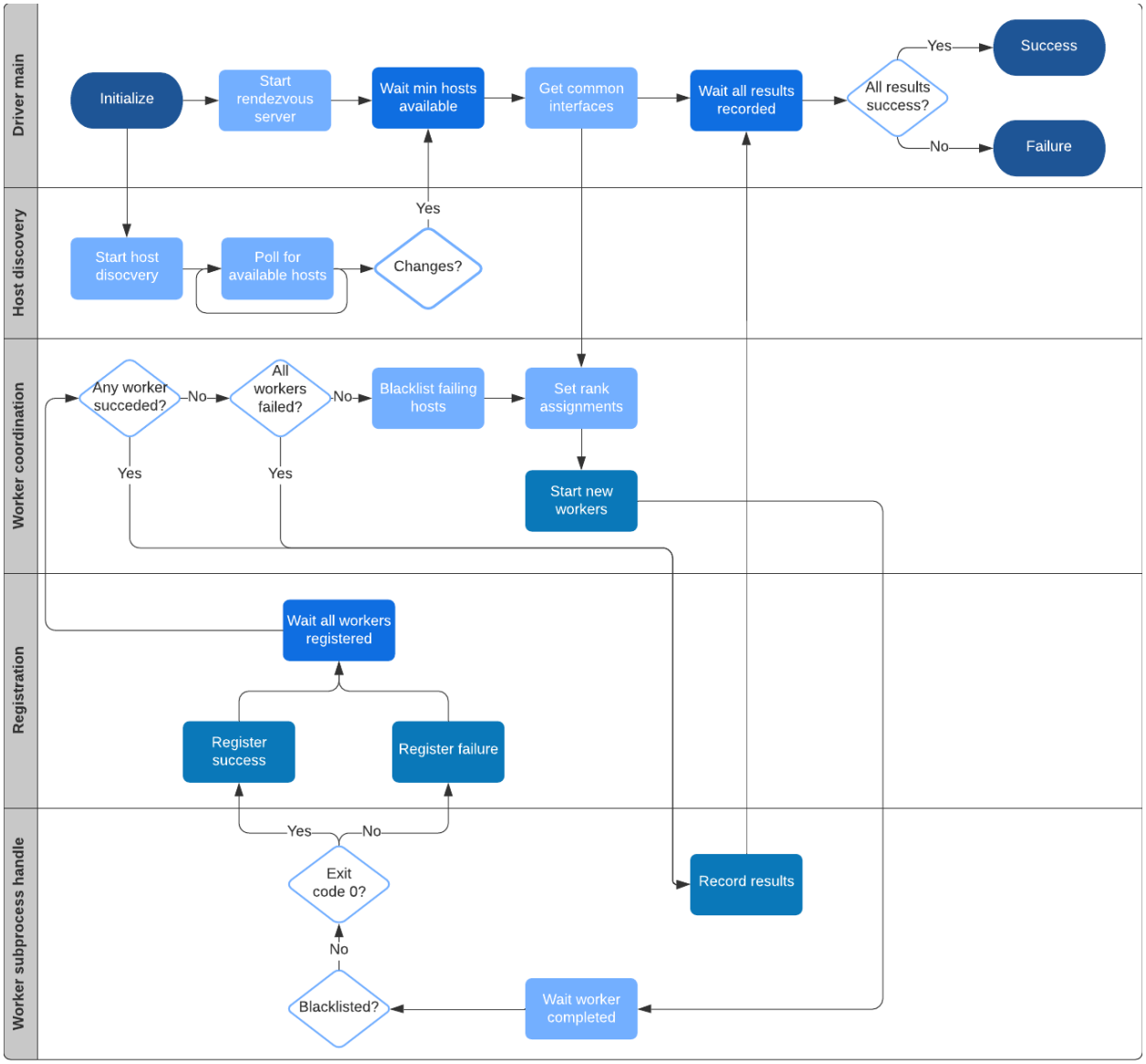
17. **Complete**. Worker will exit. In the event that the host has been blacklisted, this will manifest as a runtime error. In the event of a total job shutdown, it will be a SIGTERM from the driver.

Driver

The **driver** process is run on a dedicated instance separate from the worker pool. Because it does not handle training or execute any user code, it can run on a cheap CPU instance with modest resources. This process is responsible for:

1. Launching processes.
2. Handling *rendezvous* (registration and lookup of worker IP addresses and ranks).
3. Discovering available hosts for launching additional worker processes.
4. Notifying workers when new hosts become available.
5. Joining process results when training completes (either due to success or failure in the workers).

The full process is illustrated in the figure below:



1. **Initialize**. Parse command line arguments and setup the runtime environment.
2. **Start host discovery**. Launch a background thread to check for changes to the available host pool.
3. **Start rendezvous server**. Run the rendezvous server used to coordinate workers when job membership changes.
4. **Wait min hosts available**. Wait for --num-proc slots to become available as hosts are discovered. Timeout after some amount of time and fail the job.
5. **Get common interfaces**. Check for common network interfaces among the available hosts. These will be used for communication in Gloo and NCCL.
6. **Set rank assignments**. Assign each host and slot to a rank in the job, and set the worker environment variables with this information.
7. **Start new workers**. Launch every worker as a subprocess in a separate thread on the driver by SSHing into remote workers and executing the training script in a new Bash shell.
8. **Wait worker completed**. On the same thread used to launch the subprocess, grab the exit code and timestamp that the worker completed.
    a. **Blacklisted?** Then return and ignore this worker's exit, as we wish to exclude it from future

rendezvous and results until it is removed from the blacklist.
    b. **Exit code 0?** Indicates success.
        i. **Register success**. Training script completed successfully, so we should end the training job.
        ii. **Register failure**. Training script did not complete successfully, so we should retry without this host.
9. **Wait all workers registered**. Barrier to wait for all workers to register success, failure, or ready.
    a. **Any worker succeeded?** End the training job (failure if not all succeeded).
    b. **All workers failed?** End the training job with failure.
    c. **Blacklist failing hosts**. The host of any failing worker will be blacklisted for a time.
10. **Record results**. The job is being shut down, so record the exit code of the process.
11. **Wait all results recorded**. Wait for all worker threads to record results before finishing.
    a. **All results success?** Then exit 0 from the driver, else exit 1 and record failing workers.

Error Handling

We consider three types of errors that can occur during training:

1. **Driver failure**. This is considered a *non-recoverable* failure in this design. The driver should run on non-preemptable hardware, preferably something without a GPU allocation to keep costs down. In the future, we may explore adding driver fault tolerance through primary-backup or consensus protocols like RAFT.
2. **Host failure**. If an entire worker instance fails, we treat it as a *recoverable* failure.
3. **Process failure**. Individual processes can fail in either a *recoverable* or *non-recoverable* manner. In practice, it can be difficult to determine which kind of failure has occurred. In v1, we interpret process failure the same as a host failure, by blacklisting the failing host.

Blacklisting

When any kind of failure occurs, the driver will identify it by the safe_shell_exec call from the worker process thread returning a non-zero exit code. The background thread that spawned the process will register with the rendezvous service within the driver that a failure has occurred on a given *host* and *local rank*.

If, after rendezvous, the training job is allowed to continue, then the hosts of the failing processes will be blacklisted. Any processes still running on the host will receive a SIGTERM signal, and the processes' exit codes will be ignored.

A blacklisted host will remain blacklisted so long as it is not listed as *available* during the host discovery process. When a blacklisted host becomes available again, it can be removed from the blacklist after a *cooldown* period. This cooldown period will increase with each successive failure until the host is permanently blacklisted.

Horovod on Spark

The same architecture used for running elastic training described above can be applied to running Horovod on Spark. The only changes needed to be made are to the way that **horovod.spark.run** launches the Spark job and the individual Horovod workers.

The first version of Elastic Horovod on Spark will support Gloo only. An MPI version will only be possible where the entire Horovod cluster is brought down on re-scaling and brought up again with the new host assignment. Min and max number of worker processes will default to np which represents fault tolerance mode.

In the host discovery process, we will use the existing SparkDriverService to keep track of all active executors along with their addresses. The Horovod driver's host discovery thread will ping this server periodically to check for new executor hosts. Within the Horovod on Spark task function, individual tasks will register their addresses with the SparkDriverService as they are started. Then the Horovod driver will launch new workers on those tasks as part of the standard worker startup process. The SparkDriverService needs to discover host failures. For this, the SparkDriverService pings all registered SparkTaskServices in each host discovery cycle.

In the existing process, **horovod.spark.run** starts a Spark process that creates the required number of workers. For this, an equivalent number of Spark tasks is created, one for each required worker. Each task executes **horovod.spark.runner._task_fn**. This initializes the Spark worker for Horovod and starts SparkTaskServices which basically serves as an **sshd**. Horovod with Gloo runs **horovod.spark.driver.rsh** to launch **gloo_exec_fn.py** on the workers via SparkTaskServices, which makes a request to the SparkDriverService for the training function to execute.

In Elastic Horovod, a **--max-np** number of Spark tasks is created. As Spark workers become available over time, these tasks will be executed and workers get initialized for Horovod. The **task_fn** method registers the workers with the driver and makes them available for re-scaling. Spark can only create a fixed number of Spark tasks, so a **--max-np** must always be given when running in elastic mode on Spark..

A Spark task that throws an exception will be marked as FAILED and restarted, possibly on a different host. Horovod can therefore make Spark start a worker by throwing an arbitrary exception. There is a configurable limit of how often a Spark task can fail before the entire Spark job fails. When Spark tasks exit successfully the Spark task is considered completed and will not be restarted on a different Spark worker. It is lost for re-scaling.

When training finishes, all running tasks can terminate. Then Spark will launch remaining tasks on those Spark workers that just became idle. These tasks must be notified by the driver to also shutdown immediately.

For Spark jobs, we can set **--min-np**, **--max-np** and --no to **spark.default.parallelism**, which puts Horovod into fault tolerance mode. Running Spark in dynamic allocation mode we can set them to **spark.dynamicAllocation.minExecutors** and **spark.dynamicAllocation.maxExecutors**, respectively. The value for --np can be derived from **spark.dynamicAllocation.initialExecutors**. Because Spark workers are assumed to be unreliable, we will relax the blacklisting constraint to set a very low cooldown period once an executor is brought back up.

Changes made to support elastic mode running on Spark executors will be made available to the Horovod Spark Estimators by default. No changes will need to be made to user code. Advanced properties like max resets and commit frequency will be exposed as additional parameters to the Backend and Estimator, respectively.

Data Access

Data access in Elastic Horovod is complicated by the fact that when a reset event occurs (worker added or removed), the data access framework needs to perform the following steps:

1. Reparation the dataset for the new world size.
2. Ignore any examples that have already been processed successfully in this epoch.

This poses a challenge for existing data access frameworks. While some data storage layouts allow for random access to individual examples, others store groups of examples into individual blocks or files (e.g., Parquet). The solution we provide for Horovod needs to be general enough to account for both one-to-one file to row formats as well as one-to-many formats. The most general elements will be handled

by Horovod, while some considerations will be handled at the data framework level.

Petastorm

Petastorm is an open source framework for reading Parquet files into distributed TensorFlow or PyTorch datasets. Originating from Uber, it's become a popular framework among Horovod users, particularly users working with Apache Spark for data processing.

Petastorm's approach is to shard Parquet row groups (on the order of tens to hundreds of MB) among workers each epoch. Every worker will independently stream in row groups to fill an in-memory shuffle buffer on the client (to decorrelate row group elements and create unique batches each epoch). In Elastic Horovod, because the shuffle buffer is filled on the client, there's no reasonable way to track what examples have already been processed this epoch when a reset event happens.

For example, suppose that every row group contains 100 rows, a TensorFlow shuffle buffer is allocated with enough room for 1000 rows, and every batch contains 10 rows. In this scenario, a part of our data access pipeline, we first need to unbatch the row group into individual elements in order to fill the shuffle buffer. As each example is sampled from the buffer, a new element is inserted from the unbatched row group. Once enough elements have been sampled from the shuffle buffer to fill a batch, it will be processed in the forward/backward passes of the training loop. If a failure occurs at this point, there will be some number of row groups pre-fetched in memory, an unbatched row group being placed in the shuffle buffer, and 1000 rows sitting in the shuffle buffer.

How do we know which row groups have already been processed? It is possible that for a single row group, some of the rows have been put into a batch and processed, some are sitting in the shuffle buffer waiting to be sampled, and some are still in the unbatched stream waiting to be inserted into the shuffle buffer.

Our solution is as follows:

1. At each commit, track the row indices that were processed in the last batch. Allgather them so all workers have a global view of all the rows that have been processed.
2. When a reset event occurs, redistribute the unprocessed and partially processed row groups to (a) distribute the total number of unprocessed examples as evenly as possible, and (b) avoid shuffling row groups from one worker to another.
3. When resetting previously active workers, drain the shuffle buffers of any rows that have been moved to other workers.
4. When filling the shuffle buffers of new workers, ignore any rows that have already been processed.

The full details of the data access system for Petastorm is explained in a separate RFC.

Alternative Architectures

Host Self-Registration / No Discovery

In this setup, worker hosts are launched with information about the driver address, and register themselves with the driver when they become available. This eliminates the need for the host discovery script and background thread on the driver. However, it complicates the startup process and makes it so that elastic training cannot easily be run on bare metal. This approach should only be used when unavoidable, for instance when running on Spark.

The cost of the host discovery thread is very low, and makes launching new jobs flexible to many different host discovery methods including cloud provider APIs as well as pinging orchestration systems directly for

available instances.

Detached Workers / Replicated Driver

This architecture would have the Driver launch workers but not wait for their completion, instead tracking their process host and ID but otherwise forgetting about the worker once launched. The benefit of this approach is that we could add a primary-backup level of fault tolerance to the driver itself, making Elastic Horovod tolerant to dedicated instance failure as well as worker failure. However, by not holding the subprocess handle in the driver, we cannot quickly and easily detect process completion or send SIGTERM signals to the workers when the job is shut down by the user or other conditions. This can slow down training by increasing rendezvous time (we don't know if a job failed or is slow to rendezvous, for example).

This is something we should continue to investigate as Elastic Horovod begins being used in production. If it is found that driver failure becomes a significant concern, we will reconsider this approach. However, for now the cost of increased rendezvous time, and general reliability of dedicated CPU instances in cloud and on premise, make us reluctant to pursue this approach unless it becomes necessary.

For now, driver failure will result in job failure, but when coupled with checkpointing and automatic retry, we believe this will result in minimal downtime for most use cases. At worst, loss of the driver will result in loss of a single epoch worth of training.

Independent Workers / Fully Decentralized Training

In this approach, we eliminate the driver entirely by having each process self-launch with the training script and its own outer-loop of coordination logic. Using a consensus protocol like RAFT or Zookeeper locks, the workers elect worker 0, which then performs the functions of the driver until it fails. This approach has similar drawbacks to the Detached Workers approach, but adds additional complexity in having to elect a leader and manage consensus each time a reset event occurs. This also makes running Elastic Horovod very different from running a normal Horovod job, as the horovodrun application could no longer be used. As such, it would be impossible to run Elastic Horovod in a bare metal environment for fault tolerance alone (no auto-scaling).