Building Scalable Systems for Neural Architecture Search with RPC

Jeff Hajewski
Department of Computer Science
University of Iowa
Iowa City, IA, USA
jeffrey-hajewski@uiowa.edu

Suely Oliveira
Department of Computer Science
University of Iowa
Iowa City, IA, USA
suely-oliveira@uiowa.edu

***Abstract*—Building reliable systems for neural architecture search requires careful design consideration due to the high computational demands coupled with the necessity of fault-tolerance. In this domain, it is not uncommon for applications to crash due to GPU memory exhaustion, which makes MPI unsuitable as the communication layer for a distributed neural architecture search system. We propose an RPC-based system that is robust to node failures and provides elastic compute abilities, allowing the system to add or remove computational resources as needed. Our system achieves near linear scaling and is robust to multiple GPU node failures, allowing the failed nodes to restart and rejoin.**

***Index Terms*—distributed deep learning, neural architecture search, artificial intelligence, RPC, distributed system**

I. Introduction

The computational demands of neural architecture search (NAS) make it ideal for a parallel computing solution. Despite the implementation challenges of building a system that uses evolutionary algorithms or reinforcement learning to design neural networks, the main obstacle in neural architecture search systems is making them scalable and fault-tolerant. Scalability is important because training time and neural network complexity are typically correlated, thus as the search finds more complicated network architectures the training time per architecture typically increases. This illustrates the value in allowing the user to increase the compute resources available to the system as they are needed (e.g., at later stages in the search). Similarly, fault-tolerance is important to such a system because node failures are common, particularly GPU node failures. Neural networks typically have tens of millions of parameters and sometimes as much as hundreds of millions or even billions of parameters. Because NAS does not have any concept of the the *size* of the network it is generating, it may generate an unreasonably large network that will exhaust the available memory of the GPU being used to train the network. This memory exhaustion may crash the training program and result in the loss of a compute node. In this work, we propose a system architecture for distributed neural architecture search and build the underlying messaging system using remote procedure calls (RPC). Our system is robust to node failures and is able to add or remove compute resources while running. We demonstrate this system on the CIFAR-10 data set [17], achieving near linear scaling.

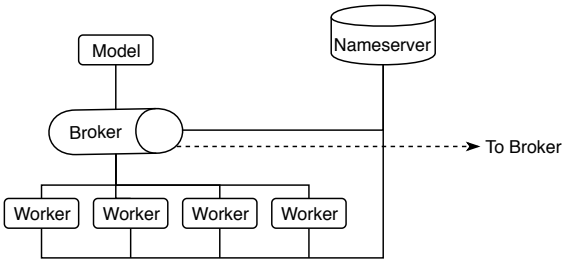

Fig. 1. Diagram of system architecture.

We chose the domain of neural architecture search because of its high computational demands, long application lifetime (when compared to the typical deep learning workload), and increased likelihood of node failures. We use this problem domain to illustrate four primary advantages to building a computationally intensive distributed system using RPC:

- RPC's higher level of abstraction when compared to MPI simplifies the process of building more complex systems and communication patterns.
- The user can avoid lower-level message serialization and deserialization (e.g., buffer allocation and deallocation) through RPC coupled with a framework such as Protocol Buffers [33] or Apache Thrift [2].
- RPC systems do not require underlying software or resource managers. A user can create an Amaxon Web Services (AWS) instance and immediately run their RPC-based code.
- RPC-based systems offer elastic compute abilities, allowing the addition or removal of nodes as needed without needing to stop or restart the system. This can be particularly useful when combined with technologies such as Kubernetes [29] or Aapche Mesos [11].

Figure 1 illustrates the system architecture, which consists of four separate pieces: a *model* that directs the search for a network architecture, a number of *workers* that perform the computational work of training and evaluating the network architectures, at least one *broker* that forms the data pipeline from model to workers, and a *nameserver* that simplifies the process of adding new brokers, workers, and models to the system in addition to managing system metadata. Using RPC

gives our system the ability to offer elastic compute resources, allowing an arbitrary number of workers to join during high computational loads as well as allowing workers to leave the system during periods of reduced computational load, decreasing the overall available compute, without needing to restart. The system is fault-tolerant to the loss of workers or brokers and is highly scalable due to the ability of the brokers to share work and compute resources. Perhaps most importantly from a usability perspective, our system is language agnostic. In our experiments, we use Python for our model and workers, which we use to build and train deep neural networks via PyTorch [24], and use Go to build the data pipeline of brokers. We use gRPC [36] to handle the generation of RPC stubs, but could have just as easily used Apache Thrift [2], which generates stubs in a larger range of languages such as Ocaml, Haskell, and Rust.

Although we are proposing RPC as an alternative to MPI, it is important to note that we are not claiming RPC is superior to MPI in every problem domain. In the domain of distributed deep learning, RPC offers many useful features. In other domains, such as distributed linear algebra, MPI is probably a better choice due to its built-in ability to efficiently broadcast messages to nodes within a system in a manner that takes advantage of the physical network architecture.

II. Evolutionary Neural Architecture Search

The goal of neural architecture search (NAS) is to find an optimal neural network architecture for a given problem. Denote the training data as X_{tr} and $\mathbf{y}_{\text{tr}}$, and a trained neural network as $\psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})$, then the problem neural architecture search attempts to solve is given by Equation (1),

$$\psi^* = \arg\min_{\psi \in \mathcal{A}} \mathcal{L}(X_{\text{val}}, y_{\text{val}}; \psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})) \tag{1}$$

where X_{val} and $\mathbf{y}_{\text{val}}$ are validation data sets and $\mathcal{L}$ is a function that measures the loss from evaluating a network architecture, $\psi(\mathbf{x}; X_{\text{tr}}, \mathbf{y}_{\text{tr}})$, using validation data $(X_{\text{val}}, \mathbf{y}_{\text{val}})$. We use the negative log-likelihood loss function. The space of neural network architectures is denoted $\mathcal{A}$ and is an infinite dimensional space. Our goal is to find a neural network architecture $\psi \in \mathcal{A}$ that minimizes the loss $\mathcal{L}$.

Neural architecture search is computationally intensive due to the cost of evaluating networks, which involves both training and validating the network. Although recent novel approaches have dramatically reduced this cost [6], [25], these techniques fix certain elements of the design process, somewhat limiting the available architectures. Despite the computationally intensive nature of the NAS problem, the task itself is trivially parallelizable across the network evaluations — two separate networks can be trained simultaneously before being evaluated against each other.

Two common approaches to NAS are reinforcement learning based approaches such as [19], [25], [38], and evolutionary approaches such as [20], [22], [28]. In our experiments we focus on the evolutionary approach due to its simplicity of concept and implementation.

We use a relatively simple approach to evolving neural network architectures in that we only construct linear networks, rather than allow an arbitrary number of incoming and outgoing connections for any given layer. The motivation behind this is two-fold: it reduces the size of search space for the neural network architecture and simplifies the implementation. Because the focus of this work is the architecture for building a distributed system that performs neural architecture search, rather than neural architecture search, we feel the trade-off is reasonable.

The problem domain we focus on is computer vision, so we restrict ourselves to convolutional layers for the hidden layers. Repeated network modules form the core network architecture. A single module is found via evolutionary search and this module is repeated several times to build the network, as shown in Figure 2. Within a module, each convolutional layer uses zero padding to maintain the spatial dimensions of the input while allowing filter size and depth to be chosen through evolution. ReLU activation functions are inserted between layers and after reductions. We use the heuristic of doubling the number of filters via a 1x1 convolutional layer, which is similar to a learned scaling factor applied across all filters, prior to reducing the spatial dimension by a factor of two via a 2x2 max pooling layer with a stride of two. The intuition behind this technique is to control information loss across reduction layers. The spatial reduction results in a 75% loss in spatial information. Increasing the number of filters by a factor of two increases the total information transferred from the layer before the reduction layer to the layer immediately following the reduction layer to at most 50% instead of 25%.

Algorithm 1 describes the evolution process. Module evolution proceeds by either appending layers to the module or performing crossover, where two networks are split at random positions and recombined to form a new network. The maximum number of layers in a module is fixed (by the user), and evolution can add either 3x3 or 5x5 2D convolutional layers with filter depths of 16, 32, 64, or 128.

Algorithm 1 High-level outline of evolutionary algorithm.

```
procedure ProduceOffspring(N_1, N_2)
    if |N_1| ≠ |N_2| then
        o_1 ← N_1.mutate()
        o_2 ← N_2.mutate()
        return [o_1, o_2]
    end if
    o ← N_1.crossover(N_2)
    o.mutate()
    return [o]
end procedure
procedure Mutate
    self.layers.append(randomLayer())
end procedure
```

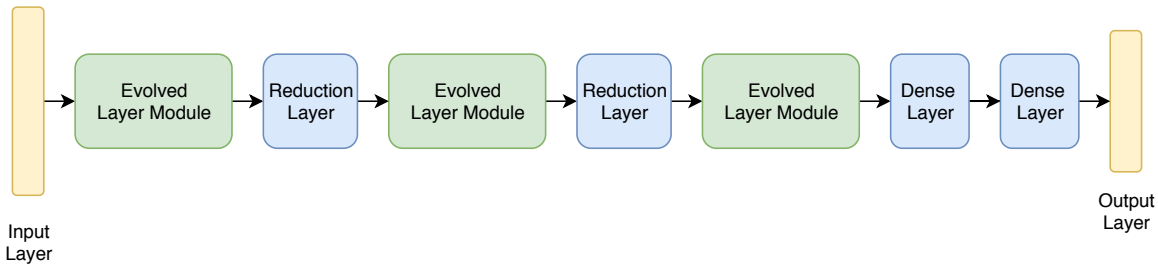

Fig. 2. Example of a constructed network.

III. RPC-based Communication

Remote Procedure Call (RPC) is a method of invoking a function on a remote computer with a given set of arguments. Most RPC frameworks involve an interface description language (IDL) to define the RPC service (that is, the API available to the caller) and some type of data serialization format. For example, gRPC uses Protocol Buffers (ProtoBufs) [33] as the serialization format for data sent across the network. RPC offers a number of advantages over MPI for network communication. It is robust to node failures or network partitions (the RPC invocation simply fails). The data sent across the network is compactly represented, giving way to high bandwidth and low latency communication. And lastly, the point-to-point communication allows for diverse communication patterns and paradigms. RPC forms the network communication infrastructure at Google [32], Facebook [2], as well as Hadoop [21], [31].

The fault-tolerant nature of RPC comes from its underlying mechanism of sending requests. Within the framework of gRPC this is handled via HTTP. This is important because it means connections between nodes are stateless once a request has been completed. It also means that lost packets on the network will be resent, giving reliable transmission. MPI has reliable data transmission, but does not allow for nodes to be removed or added during the lifetime of an application (or at least not without great programmer effort). Part of the challenge for MPI is that it relies on node rank (assigned at application start-up) as the address of messages and there is no clear way to add or remove these ranks without increasing the routing logic of the application. As an example, consider a single broker and worker. Suppose in the MPI application the broker is assigned rank 1 and the worker rank 2. If the broker and worker both fail and two new brokers and workers join, they may be given ranks 3 and 4 (broker and worker, respectively) or 4 and 3. There is no easy method of determining *a priori* which rank corresponds to which process. Under RPC we can simply have the broker listening at a given address and port and the worker listening on another address and port. If either of them fails, a new node is started and the network DNS maps the assigned address and port to the new node. This is done automatically in a system managed by Kubernetes [29] or Apache Mesos [11].

Figure 3 gives an example of a gRPC service definition for a heartbeat service, which is very similar to the heartbeat service definition used in our system. Distributed systems use heartbeats to signal that members of the system are available. The heartbeat is a node's way of tell the system "I'm still here" and is a common technique in distributed systems. The ProtoBuf compiler uses this definition to generate server stubs and service clients in a number of languages (C++, Java, Python, Go, etc.). The receiver of the RPC call must complete the server stubs by implementing the defined interface. For example, in the heartbeat service defined in Figure 3, the receiver of the heartbeat message would implement a `SendHeartbeat` function whose body would handle the logic of *receiving* a heartbeat from another process, such as updating a timestamp for the given process ID. The caller of `SendHeartbeat` uses the generated Heartbeat service client to invoke the heartbeat RPC and is only responsible for constructing the request body, `HeartbeatMsg`.

```
service Heartbeat {
  rpc SendHeartbeat(Heartbeat) returns (HBResp) {}
}
message Heartbeat {
  string id = 1;
}
message HBResp {}
```

Fig. 3. Example gRPC service definition of a heartbeat service.

While the robustness to node failures and finer-grained point-to-point communication capabilities of RPC are core to building resilient distributed systems, the more powerful feature we capitalize on is the ability to build a system that is agnostic to the type of data flowing through its pipes. Figure 4 gives an example of constructing a Protocol Buffer message type that can be used to transport arbitrary data types through the system. Protocol Buffers (and similarly, Thrift messages) support variable length byte arrays. This means the user can send a serialized object stored in the `Task`'s `task_obj` field without having to modify the system. A user can switch between running models and tasks using Java to running models and tasks using Python without modifying the data pipeline. In fact, the system can transport and run these tasks (in both Java and Python) simultaneously. By encapsulating the tasks (and results) as serialized objects stored as byte arrays, the entire system can be data type agnostic.

There is a slight catch. If information within the serialized

```
message Task {
  string id = 1;
  enum type = 2; // or string type
  bytes task_obj = 3;
}
```

Fig. 4. Example of a data agnostic message type.

```
service Task {
  // Called by a worker to request a task.
  rpc RequestTask(TaskRequest)
    returns (TaskResponse) {}
}
service Result {
  // Called by a worker to return result.
  rpc SendResult(ResultMsg) returns (ResultResponse) {}
}
```

Fig. 5. Example service definitions for tasks and results.

object is needed to properly schedule or transport the task/result to its destination, this information will need to be added to the message definition. This requires recompiling the message types and regenerating the gRPC (or Thrift) stubs. This is a minor inconvenience, as incorporating this new information into the system requires modifying the system infrastructure.

Aside from the run-time characteristics, using gRPC to define the RPC API of the system has the advantage that the service definition itself acts as documentation on the flow of information within the system. A user can look at these definitions and see how information is meant to flow through the system. Understanding the communication patterns of a system built with a lower-level message passing framework such as MPI or ∅MQ [12] requires reading through the source code. This may not be an issue for simple MPI applications where a majority of the communication logic is in a main run-loop, but for larger, more complex projects this increases the cognitive load on the user.

IV. RPC vs MPI

From a performance perspective, some prior work [27] has found RPC to provide lower latency and higher bandwidth for some tasks. That does not tell the entire story. MPI comes with built-in complex communication primitives that are capable of taking advantage the physical network architecture of the cluster. Additionally, MPI is capable of bulk broadcasts to groups of nodes using custom communicator definitions. These primitives save the programmer time and effort when building HPC applications. Settings such as large, distributed matrix multiplications or iterative optimization algorithms are particularly well suited for MPI's communication primitives. MPI is more likely to have better throughput than RPC in these types of application domains, primarily because the collective communication primitives such as `MPI_Bcast` (broadcast), `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Allreduce` are able able to more efficiently handle message routing than what is possible using RPC without mirroring the logic and implementation (which would be an incredible undertaking).

Applications that benefit from specialized communication patterns or use a service-oriented architecture are typically better suited for RPC-based communication. Consider, for example, an application that sends some amount of work to a number of worker nodes and waits for the work to be completed, such as a distributed deep learning workload [8]. In the synchronous setting, the entire system will only be as fast as its slowest node and will cause idle resources as the faster nodes wait for the slowest node to finish. In the asynchronous setting, the faster nodes can continue to process additional work while the slower nodes work on their original tasks. MPI can handle this setting using the asynchronous APIs `MPI_Isend` and `MPI_Irecv`; however, the MPI application logic will become increasingly complex if the number and type of workers is dynamic. This is because MPI uses process IDs for communication and requires the user to use message routing logic based on these IDs. When the communication numbers and patterns change, this approach can become difficult to modify whereas a more flexible approach such as RPC can handle the changes with no change to application logic (e.g., simply start more worker processes). In the system we propose, there are four classes of processes: the model, the broker, the nameserver, and the worker. It is conceptually simpler to think of these as four different services and build each of them separately using a service-oriented architecture, rather than a single program that determines its behavior based on its assigned communicator rank (as done in MPI).

One short-coming of RPC is message size limitations. This can be troublesome in distributed machine learning settings where highly parameterized models such as deep neural networks are sent across the network. The solution is to send a representation of the model, rather than the model itself, across the network. The advantage of this approach is reduced network bandwidth utilization. The disadvantage of course is the loss of trained models at the worker nodes. In practice, this may not be a major issue as final model architectures are typically trained in a specialized manner (e.g., for a large number of epochs). Additionally, models can be saved to a distributed file system such as HDFS [31].

An interesting advantage of RPC is the ability to take advantage of a container orchestration system such as Kubernetes [29] to ensure a specified number of worker nodes remain available. If a worker node goes down, or the desired number of worker nodes increases, Kubernetes restores the system to the desired state automatically. This is only possible because RPC allows us to create stateless communication channels that do not require a persistent connection between communicating nodes. While this may seem excessive for a small collection of nodes, the ability to have Kubernetes manage system state within a large system can be quite valuable in a production system.

A unique aspect of the domain of neural architecture search is that it does not require low-latency communication. Each neural network can take anywhere from minutes to hours to

```
class BaseTask:
  def run(self):
    raise NotImplementedError()

class Worker:
  def process_task(self, task):
    assert isinstance(task, BaseTask)
    result = task.run()
    self.broker_client.send_result(result)

  def serve(self):
    while True:
      task = self.request_task()
      self.process_task(task)
```

Fig. 6. Example Worker Task API in Python.

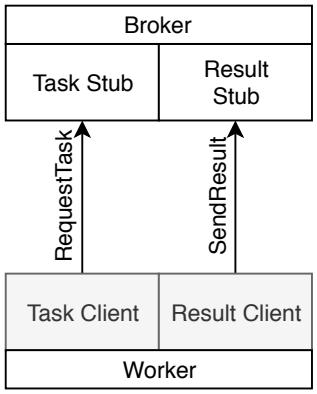

Fig. 7. Communication pattern between a worker and broker.

train, which means the frequency of task and corresponding result messages is fairly low. One prior work [28] even used a shared file system to communicate between a model and the worker nodes. The long time interval between sending the task to the worker and receiving the result from the same worker means a single server could handle a potentially large number of connections with workers, certainly on the order of hundreds and possibly even thousands.

V. System Architecture

As previously mentioned, our system consists of four different components, as illustrated in Figure 1. A model is a problem specific implementation that controls what is sent to the system for evaluation and handles the result it receives. The brokers form the data pipeline of the system, moving work from the models to the available workers. Workers form the other customizable piece of the system because they need to know how to perform their assigned work. Lastly, the nameserver maps known brokers to their network address – this is useful for connecting to brokers, such as a model connecting to a broker, a broker connecting to a broker (for broker-broker peering), or a worker connecting to a broker. The following sections will detail each components' functionality.

A. Model

The model is the problem-specific, user-defined logic that determines what work should be performed next and how the results of previously assigned work should be processed. The only requirement of the model is that is uses a broker client stub (generated by gRPC) to push work to the system and implements the result service interface, described in Figure 5, to allow the broker to push results back to the model. Figure **??** shows the flow of data between the model and broker.

The model needs to track outstanding tasks that have been sent to the broker. While the system is fault-tolerant for most brokers and all workers, if the broker the model is sending work to fails, the work the model is waiting to receive will be lost and the model will need to resend the lost work to a new broker. The simplest approach to handle this state is via a heartbeat.

B. Worker

Workers are the other user-defined and implemented portion of the system. While a single worker implementation can work for multiple model implementations, there is no general worker implementation that will work across all languages and models.

Using an API similar to that shown in Figure 6, one can use the same worker implementation for any task that inherits from the `BaseTask` class. This means the same worker applications can be used from any problem that designs its tasks to inherit from `BaseTask`, regardless of whether that problem is training and evaluating neural networks or factoring large numbers. As long as the task implements a `run()` method, the worker can execute the task without a change in logic.

Figure 7 shows the communication pattern between the broker and a worker. The most notable aspect of this pattern is that requests only flow from the worker to the broker; the broker never sends a request to the worker. In fact, the broker is unaware of any workers within the system except for those to whom it has sent a task. This is what allows the system to add as many workers as needed. Losing a worker means the broker must add the lost task back into the task queue. The addition of a worker has no impact on the system until that worker requests a task, at which point only the broker knows of the worker's existance and only while the worker is working on the given task.

C. Broker

Brokers form the data pipeline of our system. Work is sent from a model to a broker, which in turn sends the work to a free worker or to another broker via peering and returns the result to the original model. This data flow is shown in Figures 7, 8, and 9. At its core, a broker is essentially just a process with a owned task queue, a helper task queue (tasks received from other brokers via peering), a processing queue, and a results queue. Work in the owned task queue is work that was sent from a model directly to the broker – this is the work that will be lost if the broker crashes. Work in the helper task queue is work that has been sent from other brokers that

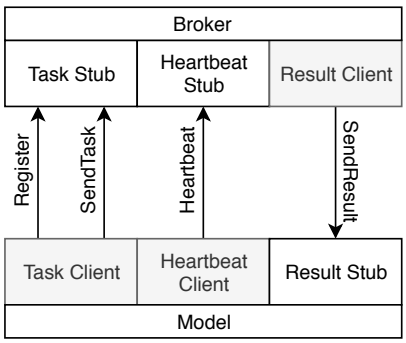

Fig. 8. Communication pattern between the broker and model.

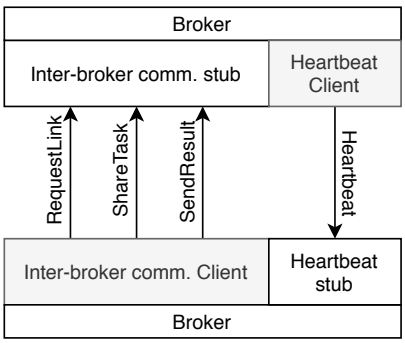

Fig. 9. Communication pattern for broker-broker communication.

the respective broker has agreed to help with. If the broker crashes, this work will *not* be lost as the other brokers will see the failure and can recover the task from their processing queue. The processing queue stores tasks that have been sent to workers or other brokers. When a result is received from a worker or another broker, the corresponding ID will be removed from the processing queue and the result will be added to the result queue. The broker pulls tasks from the result queue and sends the result to its owner, which may be a model or another broker.

D. Nameserver

The nameserver simplifies bookkeeping when starting new broker instances. Rather than forcing the user to specify which brokers a newly started broker should link up with, the nameserver stores and shares that information with all registered brokers. During start-up, each broker registers with the nameserver and begins sending heartbeat messages. The nameserver tracks which brokers have sent heartbeats recently (via a user-modifiable timeout setting) and drops brokers that have timed-out. If a broker sends a heartbeat *after* the nameserver has dropped the connection, the nameserver responds by telling the broker it must re-register with the nameserver.

The nameserver simplifies broker-broker peering by providing a central location where brokers can request addresses of other brokers. It also provides a metadata store for the system, which is important in multi-GPU scenarios. A worker running on a server with multiple GPUs needs information about what GPU to use. Storing this metadata on the nameserver allows the nameserver to manage GPU resources, rather than requiring the user to specify the GPU ID for every worker that is running, which is not scalable for hundreds or thousands of workers.

VI. Experiments

Our experiments explore the scalability and robustness of a system built using RPCs for communication to perform a long-running, computationally intensive task. We use the CIFAR-10 [17] data set because it is small enough to allow us to train reasonably performing neural networks in a short amount of time when compared with data sets such as CIFAR-100 [18] or ImageNet [9]. All experiments were run on Amazon's AWS EC2 platform using `p2.8xlarge` instances consisting of 8 Nvidia K80 GPUs. The first run on a single GPU sets the baseline number of models evaluated per generation. The successive experiments analyze how this value changes with the addition of worker nodes as well as how the system reacts to losing worker nodes.

A. Scalability

Figure 10 shows the scaling results, which are computed as the geometric mean across five generations of neural network evolution. We chose to stop after only five generations primarily due to resource limits. The relative scaling decreases as the number of workers (GPUs) increases primarily due to an increase in the rate of idle workers and worker failures. More workers means the workers are able to evaluate the candidate networks (composing a generation) in a shorter total amount of time; however, not all networks take the same amount of time to evaluate, leading some workers to remain idle while waiting for other workers to finish (at the end of a generation when there are no outstanding network architectures to evaluate).

The other issue impacting the scalability of the system is worker failures. With larger worker counts the system encountered larger models sooner and some of these models exhausted the GPU's memory. We also suspect there was a GPU memory leak in our PyTorch code but were unable to find the root cause. We manually restarted failed workers; however, we were not constantly monitoring worker status and as a result the higher worker runs were sometimes running several workers down, reducing the overall network evaluation throughput of the system. This is a great example of a scenario where an orchestration system such as Kubernetes, Apache Mesos, or similar offerings from Amazon's AWS, Google's GCP, and Microsoft Azure. The ability to automatically restart failed workers would have improved the overall scalability results.

B. Node Failures

The number of failures in Figure 11 might be somewhat surprising. The 8 and 16 node runs experienced around a 100% failure rate, while the one and two node runs had no failures—since we ran both for the same number of generations we would expect the size of the found networks to be the same

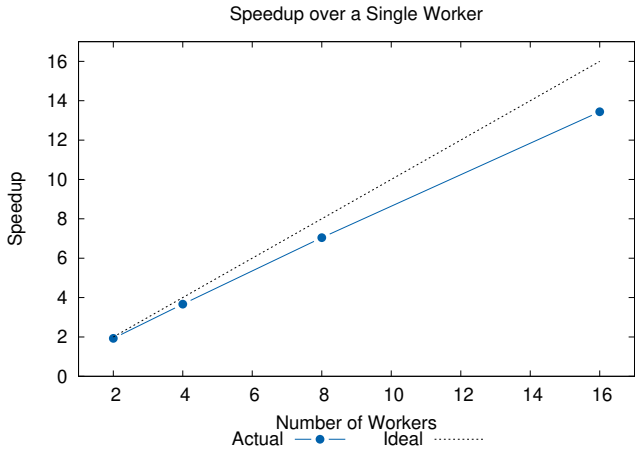

Fig. 10. Speedup as a function of the number of workers over a single worker, calculated as the geometric mean across 5 generations.

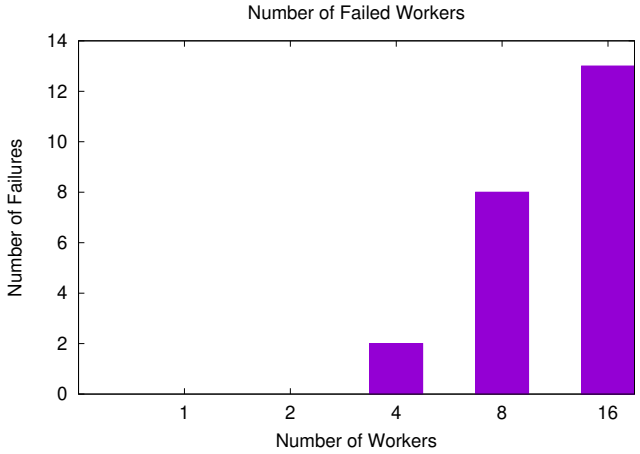

Fig. 11. Number of worker failures for a given number of workers.

with the main difference in the runs being the overall run-time. Part of this is luck in that the smaller GPU count runs simply didn't find some of the larger parameter network architectures in the later generations. The main difference is the larger GPU count runs allowed the system to try a larger number of large network architectures before we stopped the system. At the single and two worker runs we simply stopped the system before the workers failed (workers typically failed at the later generations when the architectures were larger). As mentioned in Section VI-A, we were not constantly monitoring runs and therefore would not immediately notice a worker failure. Using a system such as Kubernetes to maintain a specified system state such as "maintain eight workers" would have been useful at our scale, but is essential at larger scales.

The node failures cause another issue, which is the time it takes the system to realize there was a failure. For example, if a heartbeat needs to occur every two minutes, that is two minutes until the system realizes it needs to resend a task. One solution to reduce this latency is to switch from generational selection (selecting survivors of one generation to the next at the same time) to a steady-state selection approach and use tournament selection (selecting between pairs of individuals as they are evaluated), where survival is determined by comparing two individuals and keeping the fitter of the two. The advantage to this approach is the number of outstanding network evaluations does not impact the ability of the model to generate new candidates. As network architectures are evaluated new architectures are evolved in a steady-state of evolved individuals. This is an interesting avenue for future work.

C. Found Architectures

Figure 12 shows the best found architecture after five generations, which had a validation accuracy of 76.8%. Somewhat unsurprisingly, there were many other architectures that performed similarly to this architecture. One characteristic common in all of the high performing architectures is a high number of filters in the first layer of the module. This characteristic is not a sufficient condition for strong performance. Some networks with a very large filter depth in the first layer performed poorly, possibly due to over-fitting, while others with an average filter depth but small filter spatial dimensions also performed poorly.

These results illustrate the difficulty of designing effective neural networks—seemingly similar networks may have dramatically different performances. Consider Figure 13, which shows two neural networks with similar architectures. The network on the left had a validation accuracy of 72.6% (that is, it got about three out of every four classifications correct on previously unseen data) while the network on the right had a validation accuracy of 10.1%. This also illustrates the importance of neural architecture search as seemingly similar architectures can have wildly different results.

VII. Related Work

The idea of a brokered message queue is not new. RabbitMQ [26] is a general purpose message broker that supports the same functionality demonstrated in this paper, but messages are delivered based on a routing key, rather than the first available worker. Similar to the RPC message definitions used in this work, RabbitMQ uses a collection of bytes as the message body. For streaming data, Apache Kafka [16] is a good choice. Kafka requires a ZooKeeper [13] instance and is generally more complex to set up and run. Both RabbitMQ and Kafka are robust to failures. At the other end of the spectrum, ∅MQ [12] is a low-level messaging library that can be used to build a performant, brokered messaging system similar to the one described in this paper. Unfortunately, ∅MQ is not suitable for use in a domain where implementation language may change frequently due to its lack of an IDL. Lacking an IDL means the client and server code will need to be rewritten in each language in addition to requiring new users to read the source code to understand the pattern of communication rather than simply reading the IDL specification. Of course, a good system architecture and design document alleviates the need

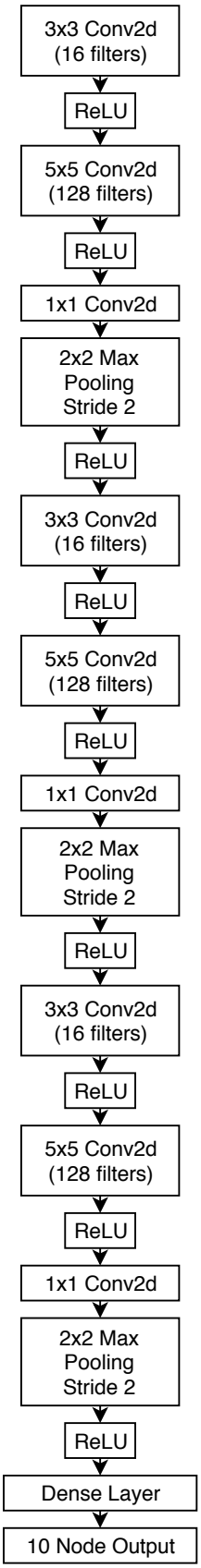

Fig. 12. Architecture of best found network.

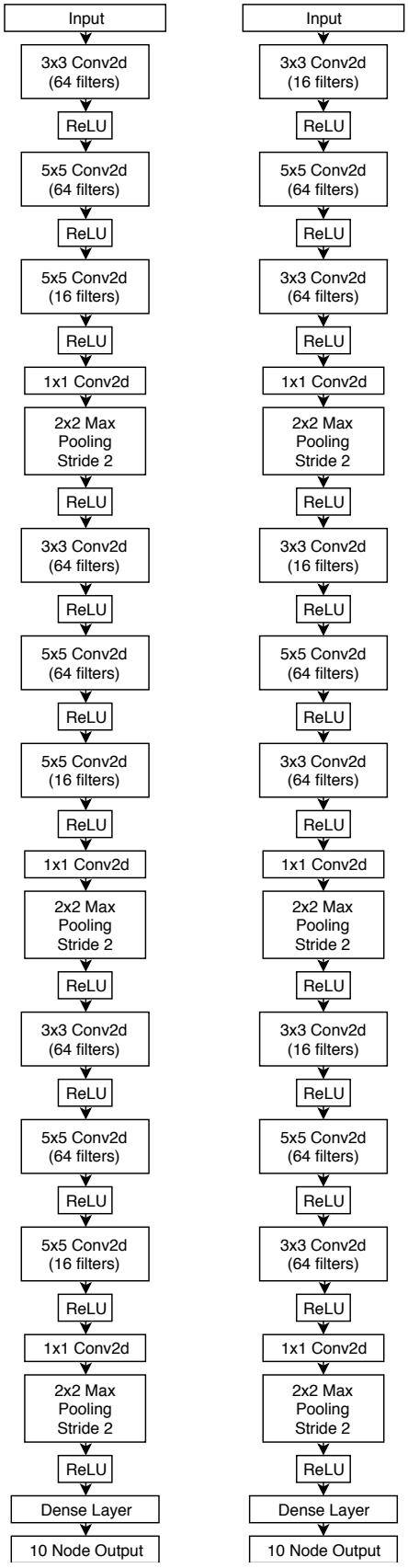

Fig. 13. (left) Architecture of a network performing close to the best found architecture. (right) Architecture of a network performing at the bottom of all found networks.

to read the source code, regardless of the chosen language or technology.

A natural question is why we would bother building our own brokered system when there are industrial strength alternatives available. These message passing systems[1] strictly deal with sending and delivering messages in a scalable and robust manner. Our broker does more than simply relay messages—the broker is determining retry logic, handling lost tasks, balancing worker load, etc. These technologies are substitutes for RPC rather than alternatives to the proposed system architecture.

Both PyTorch [24] and Tensorflow [1] offer some distributed training facilities. Other work such as Horovod [30] has improved upon Tensorflow's built-in distributed training facilities. None of these handle node failure by default, though we note that Tensorflow's distributed training framework does support running on Kubernetes.

Much of the previous work on neural architecture search uses some form of a distributed architecture consisting of a model, (possibly) a coordinator, and workers. The coordinator handles assignment of work to the workers. While many of these works don't detail the specifics of their system, we reviewed some of the available code on Github. A popular paradigm is using Python's `multiprocessing` module to run multiple models on a multi-GPU machine. These GPUs are fed from a thread-safe queue. PyTorch offers a data parallel module that handles this functionality but currently struggles to fully utilize all available GPUs in settings with 8+ GPUs, per the PyTorch documentation. Tensorflow provides similar functionality for distributed training but instead relies on gRPC.

A number of prior works use MPI in conjunction with deep learning libraries taking a data parallel approach [4], [5], [7], [34]. This is familiar to other work that takes advantage of multi-threading (such as Python's `multiprocessing` module) but utilizes much larger systems. Awan et. al. [3] build a model parallel system using MPI, relying on MPI's efficient communication primitives to avoid excessive blocking when cross-node dependencies. Unfortunately, all of these works require MPI to be installed on the cluster in which they are running, which is not an issue if the application is running on a University research cluster or at a national lab, but requires the user to set up the underlying MPI installation if run on a provisioned AWS, GCP, or Azure cluster.

Other work [15], [23] uses Spark [37] to train deep learning neural networks. Spark *can* handle the addition or loss of compute nodes and is robust to node failures (via HDFS), but struggles with iterative algorithms. Both of these works use Spark to distribute Caffe [14] models to GPU compute nodes and implement asynchronous SGD. The main issue with Spark in this context is that either the user must create a wrapper for the model in Scala or use PySpark. Using PySpark is relatively simple, as long as your model is written in Python. If your model is written in something other than Python, such as C++, then you must decide between creating a wrapper in Scala (if you know Scala) or a Swig interface and call it from Python. Both of these options are more complex than implementing a server stub to send models and using a pre-generated client to request models to train as well as report the results.

Implementing algorithms that are iterative in nature in Spark poses another issue. Spark does not handle iterative-style algorithms particularly well, primarily due to the shuffle-stage of MapReduce (since Spark is built on top of Hadoop).

[1]We use this as a generic term, since Kafka is not really a message queue.

VIII. Future Work

It can be tempting to think of system design as an all-or-nothing decision—either build a system with MPI or build a system using RPCs, but this is not the case. Consider a workload whose core unit of work is amenable to an MPI-based system but the individual units of work are independent of each other with the exception of possible boundary interactions. A combination of RPC and MPI communication might be ideal, using MPI within a single cluster to complete individual units of work and RPC to communicate between clusters. In this way, individual clusters become the workers of the system and can join and leave at will, while the broker backbone manages sending tasks to the individual clusters and returning their results to the model.

One unexplored avenue of potential future work is reusing the discarded neural network architectures in an ensemble. Much of the NAS literature finishes the search with the best found architecture—it would be interesting to explore a comparison between the best architecture and an ensemble of architectures taking into account communication overhead and voting required by the ensemble.

Another interesting avenue of work is a more in-depth comparison of building systems with RPC versus other message passing frameworks like ∅MQ, RabbitMQ, and Kafka. Specifically, what are the settings in which the alternatives are particularly well suited and what are the settings in which they struggle. Just as importantly, are there notable differences in message latency, bandwidth, and usability? These are all interesting considerations that may be known in industry through experience but, as far as we know, have not been formally studied.

A final worthwhile investigation is on the impact of steady-state evolution on system performance both in terms of throughput and quality of evolved individuals within the domain of neural architecture search. Some prior work as been done comparing generational and steady-state evolutionary algorithms in a parallel setting, such as [10], [35], [39]. A steady-state approach would allow a truly asynchronous system with minimal idle nodes and increased scalability.

IX. Conclusion

We have introduced a scalable and fault-tolerant system architecture for neural architecture search. Although the focus of this paper has been neural architecture search, this system design is applicable in other domains that benefit from elastic compute and fault tolerance, such as general deep learning research. Despite MPI's short-comings in this specific domain,

MPI remains the de-facto choice for most HPC applications. More importantly, as mentioned in the preceding section, the choice between RPC or MPI is never an all-or-nothing choice.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

[3] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl? In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI'18, pages 2:1–2:9, New York, NY, USA, 2018. ACM.

[4] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 193–205, New York, NY, USA, 2017. ACM.

[5] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda. Efficient large message broadcast using nccl and cuda-aware mpi for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 15–22, New York, NY, USA, 2016. ACM.

[6] A. Brock, T. Lim, J. M. Ritchie, and N. Weston. SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344, 2017.

[7] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1337–1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[10] R. Enache, B. Sendhoff, M. Olhofer, and M. Hasenjäger. Comparison of steady-state and generational evolution strategies for parallel architectures. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 253–262, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[12] P. Hintjens and M. Sustrik. ∅mq. http://zeromq.org/.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[15] H. Kim, J. Park, J. Jang, and S. Yoon. Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility. *CoRR*, abs/1602.08191, 2016.

[16] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB*, 2011.

[17] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research).

[18] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-100 (canadian institute for advanced research).

[19] G. Kyriakides and K. G. Margaritis. Neural architecture search with synchronous advantage actor-critic methods and partial training. In *Proceedings of the 10th Hellenic Conference on Artificial Intelligence*, SETN '18, pages 34:1–34:7, New York, NY, USA, 2018. ACM.

[20] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436, 2017.

[21] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 641–650, Washington, DC, USA, 2013. IEEE Computer Society.

[22] R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[23] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. Sparknet: Training deep networks in spark. In Y. Bengio and Y. LeCun, editors, *ICLR (Poster)*, 2016.

[24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[25] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[26] Pivotal. Rabbitmq. https://www.rabbitmq.com.

[27] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18:38–44, 01 2006.

[28] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911. PMLR, 2017.

[29] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[30] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[32] J. Van Winkel and B. Beyer. The production environment at google, from the viewpoint of an sre. *Site Reliability Engineering: How Google Runs Production Systems*, 2017.

[33] K. Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008.

[34] A. Vishnu, C. Siegel, and J. Daily. Distributed tensorflow with MPI. *CoRR*, abs/1603.02339, 2016.

[35] J. Wakunda and A. Zell. Median-selection for parallel steady-state evolution strategies. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, pages 405–414, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[36] X. Wang, H. Zhao, and J. Zhu. Grpc: A communication cooperation mechanism in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(3):75–86, July 1993.

[37] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.

[38] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. 2017.

[39] A.-C. Zăvoianu, E. Lughofer, W. Koppelstätter, G. Weidenholzer, W. Amrhein, and E. P. Klement. Performance comparison of generational and steady-state asynchronous multi-objective evolutionary algorithms for computationally-intensive problems. *Know.-Based Syst.*, 87(C):47–60, Oct. 2015.