A Scalable Architecture for Massively Parallel Deep Learning

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

I. Introduction

Building robust and scalable distributed applications is challenging — getting communications patterns correct, handling node failures, and allowing for elastic compute resources all contribute to a high level of complexity. These challenges are exacerbated for long-runnining applications such as training large deep learning models or neural architecture search. Among the many popular frameworks for distributed programming, MPI [**?**] combined with additional accelerator paradigms such as OpenMP [**?**], OpenACC [**?**], and CUDA [**?**] is the common choice for performance-critical numerical workloads. In the deep learning setting, MPI is less popular with many favoring mutli-threading in combination with multiple GPUs, and more recently experimenting with Kubernetes [**?**], [**?**]. Although part of this is simplicity of communication patterns, the fault tolerance required for long-running model training (which can last on the order of months in the industrial setting) makes MPI a poor choice for the underlying communication infrastructure.

In this work we propose an RPC-based system for distributed deep learning. We experiment with the proposed architecture in the domain of neural architecture search (NAS), an extremely computationally intensive problem that trains thousands of deep neural networks in search of an optimal network architecture. Or system 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 the models, a number of *brokers* that form the backbone of the data pipeline from model to workers, and a *nameserver* that simplifies the process of adding new brokers, workers, or models to the system. Our system offers elastic compute resources, allowing an arbitrary number of workers to join during high computational loads as well as allowing workers to leave the system, decreasing the overall available compute, without needing to restart or manual intervention. 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 or deep neural networks via PyTorch [**?**], and use Go to build the data pipeline of brokers. We use gRPC [**?**] to handle the generation of RPC stubs, but could have just as easily used Apache Thrift [**?**], which generates stubs in a larger range of languages such as Ocaml, Haskell, and Rust.

II. Neural Architecture Search

The goal of neural architecture search (NAS) is to find an optimal neural network architecture for a given problem. NAS is computationally intensive due to the requirement of having to train a candidate network in order to evaluate its effectiveness. Although recent novel approaches have dramatically reduced this cost [**?**], [**?**], these techniques fix certain elements of the design process, somewhat limiting the available architectures. Despite the computationally intense 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 eachother.

The two common approaches to NAS are reinforcement learning based approaches such as [**?**], [**?**], [**?**], and evolutionary approaches such as [**?**], [**?**], [**?**]. In our experiments we focus on the evolutionary approach due to its simplicity both in understanding and implementation.

A. Evolutionary Algorithms for NAS

We use a relatively simple approach to evolving neural network architectures. The problem domain we focus on is computer vision, so we restrict ourselves to convolutional layers for the hidden layers. The last two layers of the network are dense layer and are fixed to assure the number of categories is sensible for the given classification task. The challenge with evolving a neural network is in maintaining a valid neural network (a neural network with a path from the input layer to the output layer). We maintain this invariant as follows. Each time a layer is added to the network, it is randomly connected to another layer in the network; however, there is no guarantee that these two connected layers are part of the path from the input layer to the output layer. It is possible

they form a disconnected subgraph of the network graph. We also maintain a phantom last layer – it is not an actual layer but represents the first fixed last layer in the formed network. Maintaining this phantom layer separate from the modified layers allows us to always append layers without having to rearrange the last layer.

Algorithm 1 High-level outline of evolutionary algorithm.

procedure PRODUCEOFFSPRINT(N_1, N_2)
  if $|N_1| \neq |N_2|$ **then**
    $o_1 \leftarrow N_1$.`mutate()`
    $o_2 \leftarrow N_2$.`mutate()`
    return `ReturnFittest`(o_1, o_2)
  end if
  $o \leftarrow N_1$.`crossover`(N_2)
  return o.`mutate()`
end procedure
procedure MUTATE
  // Possibly append layer
  if `sampleUniform(0, 1)` $<$ `append_p` **then**
    `self.layers.append(randomLayer())`
  end if
  // Possibly add connection
  if `sampleUniform`$(0, 1) <$ `conn_p` **then**
    $n_{\text{conn}} \leftarrow$ `UniformInt`$(0,$ `self.n_layers`$)$
    for $i = 0$ to n_{conn} **do**
      Add random connection
    end for
  end if
end procedure

III. RPC-BASED COMMUNICATION

Remote Procedure Call (RPC) offers a method of invoking a function on a remote computer with a given set of arguments. Most RPC frameworks involve a DSL used 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) [**?**] as the serialization format for data sent across the network. RPC offers a number of advantages for network communication. It is robust to node failures or network partitions (the RPC invokation simply fails). The data sent across the network is compactly represented, giving way to high bandwidth and low latency communication. The point-to-point communication allows for diverse communication patterns and paradigms. RPC forms the network communication infrastructure at Google [**?**], Facebook [**?**], as well as Hadoop [**?**], [**?**].

RPC frameworks typically use an IDL to represent the RPC service. Figure 1 gives an example of a gRPC service definition for a heartbeat service. 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 1, 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 and is only responsible for constructing the `HeartbeatMsg`.

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

Fig. 1. Example gRPC service definition of a heartbeat service.

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

Fig. 2. Example of a data agnostic message type.

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 2 gives an example of constructing a Protocol Buffer[CITE] message type that can be used to transport arbitrary data types through the system. Protocol Buffers (and similarly, Thrift messages) support arbitrary length byte arrays [1] 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 system. 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 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 inconveninece, as incorporating this new information into the system requires modifying the system infrastructure.

A. RPC vs MPI

From a performance perspective, some prior work [CITE] 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

[1] In practice these arrays are limited in size. The failure rate of the RPC system typically increases with the size of the messages.

of the cluster. Additionally, MPI is capable of bulk broadcasts to groups of nodes using custom communicator definitions. These primitives save the programmer a lot of overhead when building HPC applications. Settings such as large, distributed matrix multiplications or iterative optimization algorithms are particuarly well suited for MPI's communication primitives.

However, applications that benefit from specialized communication patterns or that are asynchronous in nature 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. 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 work. This scenario was a motivation for [DIST DEEP LEARNING PAPER].

Of course, MPI does offer asynchronous communication, but it is not as flexible as that of RPC-based systems. In an asynchronous setting there are usually classes of nodes whose behavior are class-dependent. 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 processes/programs and build each of them separately, rather than a single program that determines its behavoir based on its assigned communicator rank (as typically done in MPI).

IV. System Architecture

As previously mentioned, our system consists of four different components. Figure 3 shows an overview of the system architecture. 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 to available workers. Workers form the other customizable part 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 individual and within the system as a whole.

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 model service interface to allow the broker to push results back to the model.

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 to a new broker.

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 5, one can use the same worker implementation for any task that inherits from the `BaseTask` class.

C. 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.

Brokers can request the nameserver send them an address of another broker with which they can link and share resources. Rather than force the user to specify which brokers should form a peering link, a broker will receive a random broker address.

Care must be taken with the nameserver as this is the single point of failure within the system. Although the system can continue to function without a nameserver, new brokers will be unable to join. This means eventually the system will fail as brokers leave the system (e.g., power outage, hardware failure, etc.). The simple solution is to use an orchestration system such as Kubernetes to make sure there is always at least one nameserver running. Because the nameserver can force brokers to re-register, restoring a failed nameserver simply forces all incoming heartbeat requests to re-register and thus restoring the state of the nameserver broker registry prior to the crash. This mechanism greatly simplifies the implementation of the nameserver. The only state that must persist between crashes of the nameserver is that of the next available broker ID. If the nameserver does not persist this to some type of durable medium (e.g., HDFS [**?**]) then it is possible for multiple brokers to receive the same broker ID.

D. 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 and returns the result to the original model. At its core, a broker is essentially just a process with a owned task queue, a helper task queue (tasks received from other brokers), a processing queue, and a results queue. Work in the owned task queue is work that was sent from a model directly to the

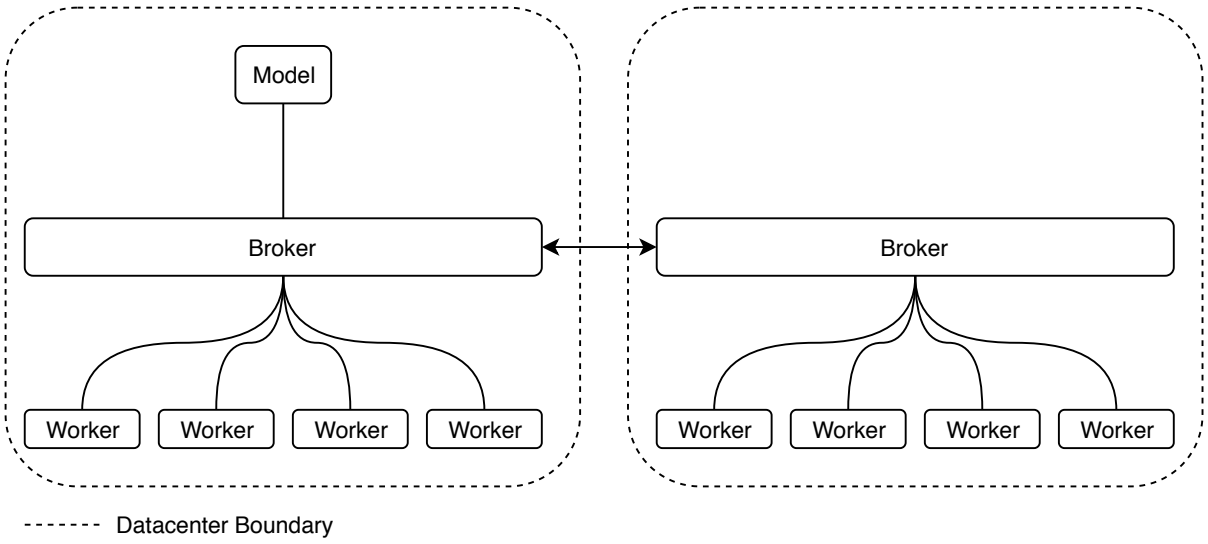

Fig. 3. Diagram of system architecture.

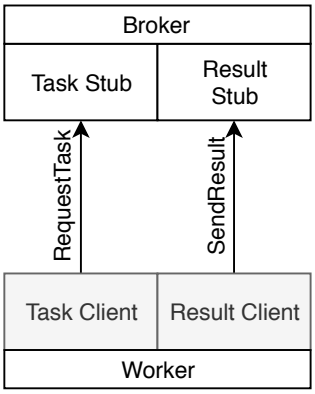

Fig. 4. Communication pattern between a worker and broker.

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

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

Fig. 5. Example Task API in Python.

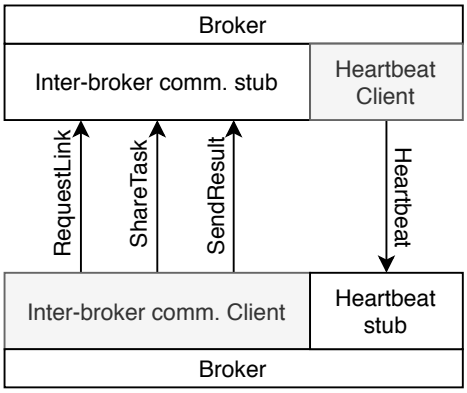

Fig. 6. Communication pattern for broker-broker communication.

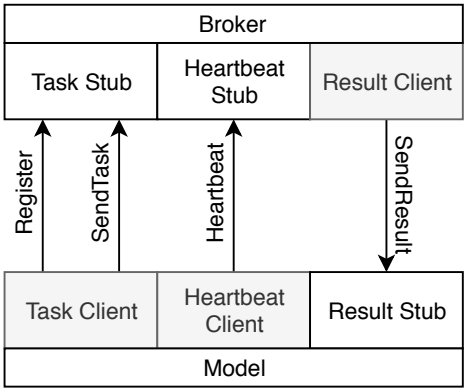

Fig. 7. Communication pattern between the broker and model.

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 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.

Brokers can establish links with other brokers.

V. Experiments

VI. Related Work

VII. Conclusion

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. An interesting avenue for future work is combining the two. Consider a workload whose core unit of work is amenable to an MPI-based system but the individual units of work are independent of eachother 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.