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 [4] combined with additional accelerator paradigms such as OpenMP [3], OpenACC [19], and CUDA [9] 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 [14], [15]. 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 [10], and use Go to build the data pipeline of brokers. We use gRPC [18] to handle the generation of RPC stubs, but could have just as easily used Apache Thrift [1], 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 [2], [11], 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 [5], [11], [20], and evolutionary approaches such as [6], [8], [12]. In our experiments we focus on the evolutionary approach due to its simplicity both in understanding and implementation.

A. Evolutionary Algorithms for NAS

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 [17] 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 [16], Facebook [1], as well as Hadoop [7], [13].

A. RPC vs MPI

IV. System Architecture

As previously mentioned, our system consists of four different components. Figure 1 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 2, 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.

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 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.

V. Experiments

VI. Related Work

VII. Conclusion

References

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

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

[3] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[4] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

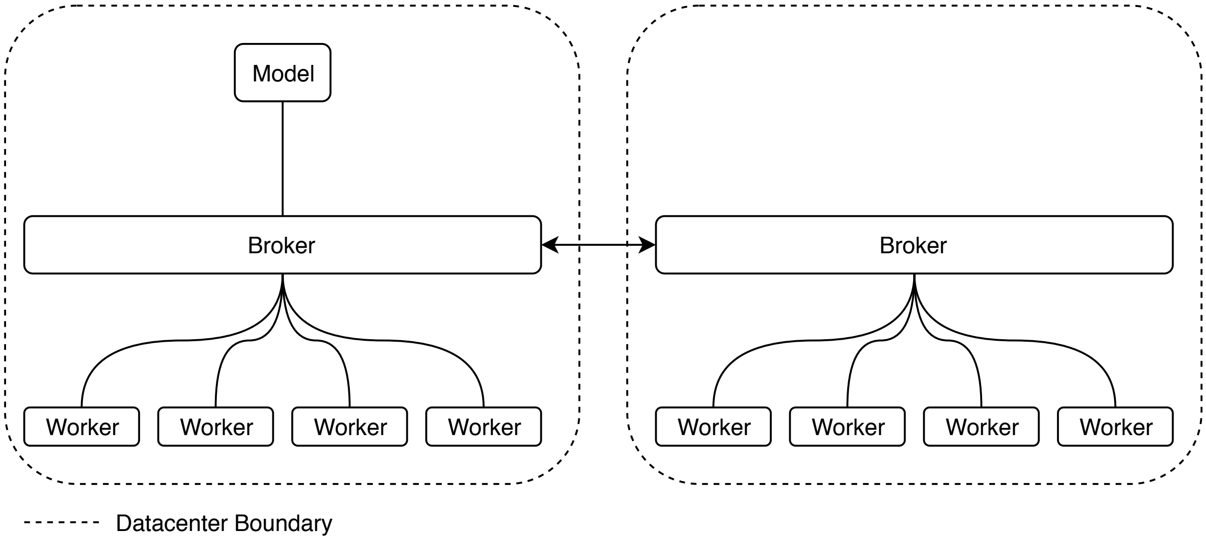

Fig. 1. Diagram of system architecture.

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

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

Fig. 2. Example Task API in Python.

[5] 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.

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

[7] 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.

[8] 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.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Song, L. Deng, J. Gong, and H. Luo. Gaia scheduler: A kubernetes-based scheduler framework. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 252–259, Dec 2018.

[15] P. Tsai, H. Hong, A. Cheng, and C. Hsu. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 145–150, Sep. 2017.

[16] 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.

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

[18] 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.

[19] S. Wienke, P. Springer, C. Terboven, and D. an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

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