An automated framework for distributed model training on Chameleon cloud

Nisala Kalupahana
Computer Science
Vanderbilt University
Nashville, USA
nisala.a.kalupahana@vanderbilt.edu

Rohit Khurana
Computer Science
Vanderbilt University
Nashville, USA
rohit.khurana@vanderbilt.edu

Ethan H. Nguyen
Computer Science
Vanderbilt University
Nashville, USA
ethan.h.nguyen@vanderbilt.edu

***Abstract*—Rapid advancements in machine learning have enabled computers to perform increasingly difficult tasks with similar accuracy to humans. The more powerful and complex models require a larger number of parameters to be optimized and thus more computational resources. These resources are typically distributed in nature - for example, across nodes in a compute cluster or, at its simplest level, across GPUs on the same device. In particular, the cloud conveniently offers users a unique opportunity to access these on-demand resources for their downstream training tasks. Here, we present a unified framework for users and researchers to automatically scale their training efforts across multiple virtual machine instances without any manual intervention. Through this workflow, individuals will be able to train their model faster by distributing the workload across all devices registered on a node and across nodes. We demo this framework on the MNIST database using Chameleon cloud and observe decreases in overall training time with no performance degradation compared to a traditional single node setup. Code is publicly available: https://github.com/EthanHNguyen/distributed-ml**

***Index Terms*—cloud, distributed computing, machine learning, scalability**

I. Introduction

Advancements in deep learning (DL) have revolutionized how computers interact with data. In fields such as robotics, DL has been used to imbue into robots environmental awareness from largely unstructured sensor data [1]. In the biological sciences, DL has been applied to genomic data to predict disease-causing structural variants [2]. While extraordinarily powerful, DL carries an expensive computational burden, with the field's progress being bottlenecked by the amount of compute resources currently available [3]. Microsoft's partnership with OpenAI, for example, includes hardware infrastructure composed of thousands of networked GPUs, an unprecedented amount that has enabled the development of revolutionary tools such as ChatGPT [4]. This problem is compounded by the simultaneous explosion of data sources that need to be analyzed.

The cloud provides a convenient way for researchers to access on-demand resources in a completely elastic manner. Given the insufficient amount of local compute available to researchers to fully train DL models, cloud computing offers individuals the ability to scale their resources horizontally and vertically in a pay-as-you-go manner [5]. Naturally, the field of cloud computing couples nicely with DL given the cloud's emphasis on resource elasticity. Generally, users of the cloud can seamlessly transition from traditional single-node compute to a multi-node architecture without affecting the high-level application. In regards to DL, several proprietary solutions have emerged to tackle its distributed use cases. SageMaker, Amazon's platform for cloud-based ML, offers distributed training libraries that handle both data and model parallelism across AWS GPU instances [6]. Vertex AI, Google's platform for ML deployment, supports manual training cluster configuration across Google's cloud infrastructure [7]. Azure, Microsoft's cloud offering, markets distributed training workflows through the Azure Machine Learning SDK for Python [8]. Unfortunately, these offerings do not directly translate to other cloud infrastructure solutions (most notably, in the open-source realm). As far as we are aware at the time of writing, there exists no standardized toolkit for distributed DL on OpenStack, one such software project that vendors such as Chameleon cloud use in their underlying operations.

In this technical report, we propose a new framework for distributed deep learning on the cloud. Specifically, we prototyped an easy-to-use drop-in framework for Chameleon cloud (i.e., OpenStack) that allows users to

automatically spawn VMs in a given network topology and subsequently launch distributed DL training according to the user's specifications. We envision that this framework will enable researchers to easily scale their model development while ensuring no downstream performance degradation. We demo this toolkit on Chameleon cloud to train a simple MNIST model on a template-generated CPU-based VM cluster.

II. Overview of Technolgoies Used

A. PyTorch

PyTorch is an open-source machine learning framework that eases the development of DL models. PyTorch provides built-in capabilities for users to elastically scale their training across multiple CPUs and GPUs in a distributed fashion through APIs such as DistributedDataParallel (DDP) [9]. As a result, model development and validation can occur at a much faster rate relative to the traditional single-node, single-device setup.

There are two main paradigms to distributed training: data parallelism and model parallelism. The former partitions the training dataset and replicates the entire model across all registered devices whereas the latter shards the model itself but keeps the data intact on every device [10]. Under the hood (and, as suggested by its name), DDP follows the data parallelism approach. Through PyTorch, each spawned process controls one or more devices and has membership in a process group, an abstraction that enables distributed coordination, communication, and synchronization through a specified backend, gloo for distributed CPU training and NCCL for distributed GPU training. In particular, message passing across CPU-backed network-connected machines, the primary focus of this report, can be initialized from TCP or through a shared file system. The former was chosen in our efforts.

Let's suppose that a researcher wants to perform distributed training on a two-node system with N CPUs on node 1 and M CPUs on node 2 (Fig. 1). Each process is assigned a unique node-specific identifier termed the local rank that ranges from 0 to N - 1 on node 1 and 0 to M - 1 on node 2. A global identifier, termed the global rank, is also dished out to each process and ranges from 0 to N + M - 1 across the composite system. Typically, the process with global rank 0 serves as the master and is additionally responsible for process coordination as well as training configuration and initialization. Once each process has obtained a copy of the model as well as a unique shard of the training data, a forward pass is independently computed to ascertain the loss. A backwards pass is subsequently initiated to compute loss function gradients; these are broadcasted to all other processes via collective communication primitives such as AllReduce (details regarding its optimized algorithms are beyond the scope of this report) [11]. Gradients are synchronized by averaging them together, mathematically justifiable given the linearity of the gradient operator [12]. This ensures that each process is at the same starting point per iteration while preserving network bandwidth.

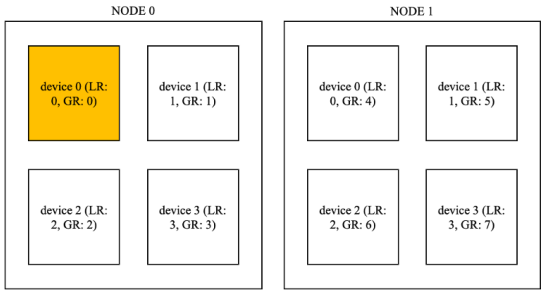

Fig. 1. Visualization and application of DDP terminology (LR = local rank, GR = global rank).

B. PyTorch Lightning

PyTorch Lightning is a high-level wrapper over PyTorch that simplifies DL model development. With PyTorch Lightning, the logic to enable distributed training is completely deferred to the Trainer module, allowing complex PyTorch code to seamlessly migrate to a fully distributed setup based completely on user specifications. Under the hood, PyTorch Lightning has the ability to wrap native PyTorch code around DDP, enabling the functionality described earlier in section II-A. PyTorch Lightning was used to ease model deployment across the VM cluster on Chameleon cloud.

C. OpenStack & Chameleon Cloud

One major problem with distributed ML training is that it's often difficult to set up. For instance, if one needs to set up a machine, clone some repository, download packages, and start training with specific parameters on multiple machines, it's highly likely that some sort of user error will occur. Additionally, the larger the model, the more machines are needed, and thus the more problems are likely to occur.

In order to solve this problem, many infrastructure solutions provide declarative deployment systems. This comes in the form of some file that matches a specification, which is used to determine what infrastructure should be deployed and how it should be configured.

The infrastructure system our solution is designed for is OpenStack, one of the most widely deployed cloud orchestration softwares out there. OpenStack manages pools of compute, storage and networking resources, which can be partitioned and allocated by users. Many cloud infrastructure systems use OpenStack. For instance, Chameleon Cloud, which is very commonly used by researchers for academic purposes, provides interfaces into several computing center clusters, all of which are managed using OpenStack.

In addition to managing infrastructure, OpenStack also has several additional components, including orchestration, fault management, and service management. The component we are specifically interested in for this project is orchestration – that is, the coordination of multiple computer services to serve a larger goal. In distributed ML, that's exactly what we are trying to do, and OpenStack provides an orchestration service called Heat that we use to accomplish our goals.

Heat uses templates, written in YAML, to describe what the final application should look like. They describe both individual resources and the relationships between them (e.g. this instance is part of this network). The system takes in templates and translates them into the appropriate API calls to create, manage, and destroy infrastructure based on actual versus desired state (Fig. 2). Each set of infrastructure is managed in something known as a "stack", which is described by one of these template files. Stacks manage the infrastructure created within them, so if a template is modified, the stack will change its managed infrastructure to mirror the template. This makes infrastructure cleanup incredibly easy – if you want to remove everything you've created in an experiment, just delete the stack; you can't cause side effects in the rest of your infrastructure. And because templates are declarative, every change you make to infrastructure is documented and repeatable – a template will create the same end result no matter what.

Templates are broken into two sections: resources and parameters. Resources are things like instances or networks – the actual parts of your infrastructure. For example, to create a network, you would add the following to your template:

```
internal_net:
   type: OS::Neutron::Net
```

Now, let's say you wanted to add a subnet to this network. This could be done like this:

```
internal_subnet:
   type: OS::Neutron::Subnet
   properties:
     network_id: { get_resource:
        ↪ internal_net }
     cidr: "10.8.1.0/24"
     dns_nameservers: [ "8.8.8.8",
        ↪  "8.8.4.4" ]
     ip_version: 4
```

This shows how resources can relate to each other in a template – this subnet is part of internal_net, the resource we just created.

Resources can also relate to infrastructure that already exists in a project, which is how stacks (or infrastructure not managed by a stack) can talk to each other. For example, a router that needs to access the internet could be declared like this:

```
internal_router:
   type: OS::Neutron::Router
   properties:
     external_gateway_info: {
        ↪ network: public }
```

As another example, a port for an instance can refer to default security groups (although these could also be declared as resources for maximum portability):

```
main_port:
   type: OS::Neutron::Port
   properties:
     network: { get_resource:
        ↪ internal_net }
     fixed_ips:
       - ip_address: "10.8.1.250"
     security_groups:
       - default
       - ENABLE_SSH
```

The problem of configuring each instance individually is also solved by templates, which provide a "user data" attribute that allows you to specify a script to be run when the instance is first started up. This could be a standalone script, or something that installs Puppet or Ansible for further managed set-up.

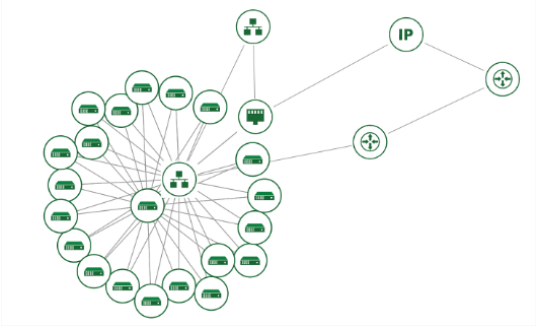

Fig. 2. Sample network topology on Chameleon cloud spawned by the template generator.

```
main_server:
   type: OS::Nova::Server
   properties:
     flavor: m1.medium
     image: CC-Ubuntu20.04
     key_name: { get_param: Key }

     networks:
       - port: { get_resource:
           ↪ main_port }
     user_data: |
       #!/bin/bash
       # Anything you want!
```

You'll notice that the SSH key is not declared directly in this template, but is instead pulled using get_param. This brings us to the second part of templates: parameters. Parameters are values that the user can give when a template is deployed, so that way a generic template can be used by many different people. This is why we've made SSH key a parameter – the user can specify whichever one they'd like to use based on what they already use in their cloud environment. Parameters can also be constrained to specific resources (like SSH keys), giving the user of the template a better UX.

```
Key:
   type: string
   description: Name of a key
      ↪ pair to enable SSH access
      ↪  to the main instance
   constraints:
   - custom_constraint: nova.
      ↪ keypair
```

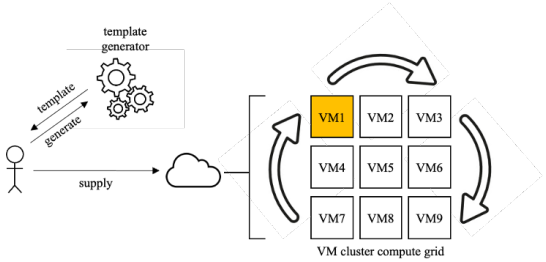

Fig. 3. High-level overview of the solution architecture.

III. Architecture & Methodology

MNIST, a dataset of 60,000 28x28 greyscale pixel images of handwritten digits from 0-9, was chosen as the empirical dataset for its simplicity and popularity in the machine learning community [13]. For simplicity, the neural network used a one hidden-layer multi-layer perceptron (784 nodes $\rightarrow$ 100 nodes $\rightarrow$ RELU [14] $\rightarrow$ 10 nodes) to classify the digits. To pre-process the data, we normalize and flatten the samples. Cross entropy was used as the loss function and Adam [15] as the optimizer. The learning rate was set to $1e-3$ and a batch size of 64.

In order to train the model, the user first needs to generate a template (Fig. 3). This generation could be made fully generic using multiple templates and parameters, but because Chameleon Cloud's GUI puts limitations on certain features of templates, we instead created a Python file that generates a template based on certain parameters – namely, the number of worker nodes the user would like, the Git repository the user would like to clone, the working directory in the repository for the model, and the model training entrypoint. This generates a template.yml file, which the user can simply drop into Chameleon Cloud on the "Stacks" tab to create all of the requisite infrastructure and automatically start training. All of the worker nodes are set up to connect to the main node (which has a fixed IP address in the subnet), and to shut down when training is over to save money. These types of optimizations would ordinarily be an extra burden on a researcher trying to train their model, but because of our system, they can be abstracted away.

IV. Experiments

Performance testing was accomplished by establishing various VM configurations on Chameleon cloud through the developed template generator. All VMs utilized Ubuntu 20.04 and were equipped with 2 VCPUs and

TABLE I
PERFORMANCE RESULTS FOR VARIOUS CLUSTER CONFIGURATIONS ON CHAMELEON CLOUD.

Node Count	Time to completion (s)
1	173
2	146
5	102
17	29

4GB of RAM. The MNIST model was allowed to run for 10 epochs and the time to completion was monitored.

Table I displays the performance results for various cluster configurations to train the MNIST model. As expected, distributed deep learning training significantly improves time of execution as the number of nodes increases. Moreover, our framework easily enables researchers to scale their resources to meet the computational demands of their DL model.

V. DISCUSSION

Our proposed framework allows researchers and practitioners to easily scale their resources to quickly and efficiently train machine learning models. Our empirical study using the MNIST dataset suggests that our methods scales well to multiple CPU nodes.

Future studies can be conducted using larger datasets and larger models to better approximate real-world workloads. Additionally, researchers and practitioners often have access to heterogeneous types of resources: CPUs, GPUs, TPUs, IPUs, etc. Future directions also include supporting additional hardware. While it is noteworthy that larger models require increasingly specialized hardware, allowing freely available and cheaper hardware such as CPUs enable more equal access to machine learning.

VI. CONCLUSION

We propose a unified framework for users and researchers to automatically scale deep learning training efforts across multiple virtual machine instances in Chameleon cloud. Users are able to easily provision and orchestrate their efforts using templates. Our empirical study on the MNIST dataset observes decreases in overall training time as additional nodes are added. Future directions include using larger models, larger datasets, and expanding support to heterogenous hardware.

ACKNOWLEDGMENT

This report was completed as part of a graduate course in distributed systems. We would like to thank Dr. Aniruddha Gokhale for his insightful lectures on a variety of core topics in the field.

REFERENCES

[1] H. A. Pierson and M. S. Gashler, "Deep learning in robotics: a review of recent research," *Advanced Robotics*, vol. 31, no. 16, pp. 821–835, 2017.
[2] S. Webb *et al.*, "Deep learning for biology," *Nature*, vol. 554, no. 7693, pp. 555–557, 2018.
[3] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, 2020.
[4] J. Roach, "How microsoft's bet on azure unlocked an AI revolution," 2023.
[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[6] E. Liberty, Z. Karnin, B. Xiang, L. Rouesnel, B. Coskun, R. Nallapati, J. Delgado, A. Sadoughi, Y. Astashonok, P. Das, *et al.*, "Elastic machine learning algorithms in amazon sagemaker," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 731–737, 2020.
[7] "Distributed training," 2023.
[8] "What is distributed training?," 2023.
[9] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.
[10] J. Geng, D. Li, and S. Wang, "Horizontal or vertical? a hybrid approach to large-scale distributed machine learning," in *Proceedings of the 10th Workshop on Scientific Cloud Computing*, pp. 1–4, 2019.
[11] S. Subramanian, "Distributed data parallel."
[12] A. Wälchli, "Distributed deep learning with pytorch lightning (part 1)."
[13] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.
[14] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.
[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.