Simplifying HPC and ML Application Deployment across the Computing Continuum

Jack Brassil
Dept. of Computer Science
Princeton University
Princeton, New Jersey, USA 08544
Email: jbrassil@cs.princeton.edu

***Abstract*—To accelerate computational science on campus and beyond we investigate the *Ray* software framework across a range of different computing environments including 1) desktop systems; 2) campus compute clusters; 3) NSF-supported mid-scale computing and networking research infrastructures; and 4) hybrid campus and commercial compute clouds such as the Google Cloud Platform (GCP). Ray offers a single, unifying, open-source, distributed computing framework that promises to allow users to code once – using the Python language in Jupyter notebooks familiar to many researchers across disciplines – and easily deploy applications spanning diverse computing platforms. In this paper we specifically focus on the benefits of training researchers and students on using Ray on the FABRIC and CloudLab mid-scale research infrastructures, highlighting the ease of use of multiprocessing and hardware accelerators (e.g., GPUs) across heterogeneous hardware systems.**

1. Introduction

A principal challenge in advancing scientific computing on campuses today is the efficient computation of a mix of conventional *High Performance Computing* (HPC) workloads with fast-growing *Machine Learning* (ML) workloads. Both benefit from acceleration through hardware technology support including co-processors such as GPUs and TPUs, as well as interconnects including NVLink, Infiniband, and Slingshot.

Over the past two decades efficient campus computation has been realized by deploying collections of big data processing frameworks (e.g., Apache *Spark*), data analysis tools (e.g., *Pandas*), GPU programming environments (e.g., *CUDA*), and Python compilation (e.g., *Numba*), often cleverly combined to address a specific application need. Yet this approach of integrating disjoint systems can be complicated. It is also poorly suited for higher education settings, where training investigators and students on so many tools reduces research productivity, strains campus research computing expertise, and taxes already thinly stretched IT finances. In addition, new user demand is growing rapidly as less sophisticated data science and ML users from outside STEM disciplines need to learn and perform computational research in their fields of study.

Ray was created at UC Berkeley as an open-source project to improve the efficiency of both scientific computing *and* machine learning workloads by distributing and parallelizing them [1]. Ray has been rapidly and broadly adopted by large commercial service platform operators. *Uber*, for example, is one of several major platform providers that integrate Ray in its service architecture for matching vehicles and passengers. Ray is currently supported by its original developers at *AnyScale, Inc*, providing the commercial support needed to ensure reliable, high-performance distributed systems for both experimental and production computational science.

We see a timely opportunity for educators and campus Research Computing (RC) organizations to rationalize the software frameworks they offer as core campus computing services. To realize this opportunity we consider Ray for scaling Python ML and conventional HPC applications [2]–[4]. Ray provides a single, common environment that supports both types of workload. Ray's ease of use, multi-programming abstractions, and integrated scheduling and cluster management capabilities offload many of the burdens of programming large-scale systems. Ray represents a more modern alternative to comparable established frameworks including Spark and Dask. In this paper we specifically do not seek to compare these frameworks side-by-side as already done elsewhere [5]. Rather, we consider Ray as an promising choice to satisfy our predicted future workload mix, our heterogeneous computing systems, and our growing cohort of campus ML users with limited distributed computing expertise.

Ray parallelizes suitable Python applications easily. It also offers integrations with common scientific computing and data science tools, including RayDP ("Spark on Ray"), Scikit-learn, and Hugging Face Transformers. A single, spanning distributed computing environment eliminates the complexity of using different tools in different settings, reducing the need to learn a variety of frameworks and integrations, each of which might introduce a steep learning curve. Ray helps us take a step towards a longer-term vision of a single computing framework that integrates computational resources across the distributed computing continuum.

In this paper we show that Ray can help accelerate research across a wide variety of campus compute settings – from small edge computers sensing data in the field to massive shared compute clusters in campus datacenters. We demonstrate how Ray can help extend campus resources by facilitating computing on shared computing research infrastructures including FABRIC [6], and CloudLab [7]). In addition, FABRIC's *Facility Ports* enable attaching higher performing campus clusters

*This project is supported in part by the National Science Foundation under grant OAC-2429485.

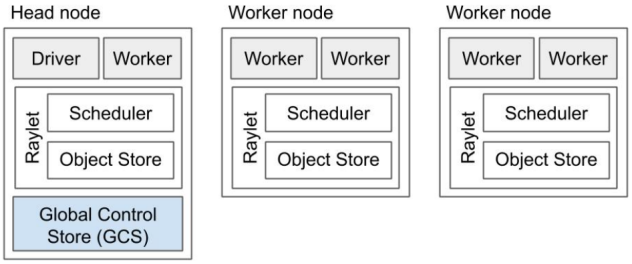

Fig. 1: Illustration of cluster components in head and worker nodes.

to create hybrid cloud computing systems. Our preliminary experiments suggest that Ray can transform our view of mid-scale research infrastructures into integrative hubs for regional, multi-institution computational research.

2. Ray Backgrounder

The emergence of Big Data applications led to the development of early, successful software frameworks for distributed data analysis (e.g., MapReduce [8], Hadoop Distributed File System [9]). This was later followed by frameworks for training Deep Neural Networks (DNNs) (e.g., TensorFlow [10], PyTorch [11], companion hardware accelerators (e.g., GPUs) with associated software (e.g., Nvidia CUDA DNN), and public datasets (e.g., MNIST [12], ImageNet [13]).

In 2018 the UC Berkeley *RISELab* team introduced Ray, a distributed computing framework that makes it easy to horizontally and vertically scale both ML and Python science workloads. Ray runs on infrastructures ranging from laptops to large compute clusters. Computational scientists can easily parallelize and test code on their desktops, and then deploy at scale on campus clusters – or commercial compute clouds or hybrid clouds – without the need to specify or construct complex compute infrastructures or even modify their code. Ray handles all aspects of distributed execution — from scheduling tasks to auto-scaling to fault tolerance [14]. Unlike earlier *bulk-synchronous parallel* (Spark) and *task parallel* (Dask) systems, Ray was designed to support simulation, training, and model serving for RL applications. Hence, the deployment of Ray can help move campus computing from a collection of discrete compute platforms to a single unified continuum of resources on-premises and beyond.

From a computational scientist's perspective, Ray handles the work of managing a cluster, scheduling tasks, transferring network data, checking system health, overcoming faults, etc, allowing the investigator to focus on the underlying data science problem and not the complexities of distributed systems.

A. Ray Clusters & Jobs

Fig.1 depicts how Ray *cluster* is a set of logical worker nodes sharing a head node. Nodes may be heterogeneous, and may be located anywhere. A single physical machine with multiple cores (e.g., desktop) can be configured as a cluster. Ray clusters can be configured with a fixed set of nodes, or may autoscale up and down on-demand. Ray provides native support for clusters within commercial cloud systems including GCP and AWS. Ray also supports Kubernetes-orchestrated container environments (e.g., Docker), where a node is organized as a pod. In this setting Ray benefits from the many advantages of application containerization including portability, security, reliability and manageability.

A Ray *job* is a single application; it is the collection of Ray tasks, objects, and actors that originate from the same (Python) script. The worker that runs the script is called the *driver* of the job.

B. API & Programming Model

Many Ray use cases fortunately require understanding only a handful of basic Ray API calls (see Table I). Functions are implemented as follows:

- A *task* represents the execution of a remote function on a stateless worker. When a remote function is invoked, a *future* representing the result of the task is returned immediately. Futures can be retrieved using `ray.get()` and passed as arguments into other remote functions without waiting for their result.
- An *actor* represents a stateful computation. Each actor (Class) exposes methods that can be invoked remotely and are executed serially. A method execution also returns a future.

Ray employs a dynamic task graph computation, where the execution of both remote functions and actor methods is automatically triggered when their inputs become available.

Ray's architecture [14] consists of two parts: an application layer and a system layer. The application layer implements the API and the computation model, and the system layer implements task scheduling and data management to satisfy the job's performance and fault-tolerance requirements. Ray's *system* layer consists of three major components: a Global Control Store (GCS), a distributed scheduler, and a distributed object store. Each component is horizontally scalable and fault-tolerant.

C. Machine Learning with Ray

Ray has been extensively extended to support a complete set of machine learning requirements. Associated Ray libraries address each RL loop component including distributed model training (Ray *Train*), hyperparameter tuning (Ray *Tune*), model serving (Ray *Serve*) and overall application development (Ray *RLlib*), making it exceptionally capable of helping to meet the growing campus demand to handle these applications.

D. Simple Ray Programming Examples

First, let's see how invoking Ray is comfortably familiar to Python programmers. We begin with parallelizing a set of 96 *empty* tasks each consisting of nothing other than a half-second duration `sleep()` call. The left side of Fig. 2 shows the code used to execute the code serially (i.e., on a single core). The right side shows the minor modifications needed to parallelize the same code to run on *any* campus Ray cluster, including a single node. The use of the decorator `@ray.remote` to wrap a

Call	Description
futures = f.remote()	The non-blocking execution of function f remotely. f.remote() can take objects or futures as inputs and returns futures.
objects = ray.get(futures)	Returns the values associated with one or more futures. This call is blocking.
ready_futures = ray.wait(futures, k, timeout)	Return the futures whose corresponding tasks have completed as soon as either k have finished or the timeout expires.
actor = Class.remote()	Instantiate *Class* as a remote actor, and return a handle to it.
futures = actor.method.remote()	Non-blocking call on the actor method to return future(s).

TABLE I: Basic Ray Core API calls.

```
import time


def f(x):
   time.sleep(0.50)
   return x

futures = [f(i) for
    i in range(96)]
print(futures)
```

```
import time, ray
ray.init()

@ray.remote
def f(x):
   time.sleep(0.50)
   return x

futures = [f.remote(i)
    for i in range(96)]
print(ray.get(futures))
```

Fig. 2: The minor changes to serial Python code (left) for parallelization (right) of 96 empty tasks on a Ray cluster.

function is familiar to, say, those using *Numba* JIT compilation to accelerate their code.

Fig. 3 informs a more revealing demonstration of Ray's built-in portability and cluster management capabilities. We modified the RHS of Fig. 2 code to invert 96 4000x4000 matrices with Numpy.

```
import ray, numpy as np
N = 4000
ray.init()

@ray.remote
def f(x):
   arr = np.random.randn(N, N)
   inv_arr = np.linalg.inv(arr)
   return x

futures = [f.remote(i) for i in range(96)]
print(ray.get(futures))
```

Fig. 3: The Ray Python code to invert 96 4000x4000 matrices.

We will use the code of Figs. 2 and 3 in the next section, where we will show how Ray was deployed and performed efficiently across our various campus systems and shared community research infrastructure settings.

3. Experiment Methodologies and Results

In this section we report on experiments performed on several desktop and laboratory systems, CloudLab, and FABRIC. An important learning is that Ray makes moving between these diverse systems relatively straightforward through programming abstractions that effectively hide complexities of exploiting cluster management, multiprocessing and the underlying diverse hardware (e.g., GPUs).

We deploy a collection of tools to monitor the execution of Ray-accelerated programs. Even in a simple single node cluster monitoring distributed execution can become complicated, partly because Ray is offloading cluster management decisions (e.g., task relocation) from the programmer. The *Ray Dashboard* can help monitor and understand execution behavior by presenting live hardware resource consumption, performance metrics, job status, and internals such as task and actor distribution. The dashboard also offers additional visualization panels via integrations with *Grafana* and *Prometheus*. Using these tools to track performance metrics becomes crucial as additional physical nodes are added to the cluster, as we will see later in this section. We also continue to rely on Linux system tools such *nvtop* and *btop* to monitor runtime behavior. Ray also provides extensive logging mechanisms to analyze experiment behavior offline.

A. Desktops and local clusters

1) Methodology: Let's begin with an example of Ray on a cluster comprising a single physical machine. This example can represent a common use case on campus – either a modest compute user or one debugging or validating code before uploading to a larger cluster for execution.

2) Results: We executed the code on the RHS of Fig. 2 on a bare metal Dell PowerEdge R6515 1U single-socket rack server with a 2.9 GHz AMD EPYC 7542 32-Core Processor (64 NUMA nodes) with 64 GB of memory, and *Ray* v2.40.0 on Ubuntu 24.04. The speedup obtained by Ray in this example reduced execution time from 24.02 to 5.7 secs, which would be unsurprisingly similar to what could be achieved using multiple other possible approaches including the standard Python multiprocessing library. But as we will see, using Ray allows the programmer to enjoy many of Ray's accompanying benefits.

Next, we executed the matrix inversion code of Fig. 3 on a desktop (HP Z6 Workstation, Intel Xeon Silver 4208 32-core CPU @2.1GHz). A top-based monitor showed that all cores were compute-bound during the execution interval of 65.7 secs.

Let's now consider an example of Ray's use on a single node with a GPU. For simplicity we selected a small yet

classic learning application, training a basic neural network image classifier on the MNIST dataset with distributed data parallelism. The open-source code [15] uses *Ray Train* to train a Pytorch Lightning module.

Testing a single, isolated application gives us an initial sense of whether to deploy a job on either a single core with GPU or across workers on many CPU cores (with no GPU). The examples we present used the Nvidia CUDA 12.4 toolkit and driver version 550.54.15. We first tested on 19 (of 20) available cores on an otherwise idle bare metal HP Z6 desktop equipped with a Intel Xeon W-2255 CPU @3.70GHz, and compared the performance against using only a single core with an Nvidia Quadro P2200 GPU. In this one example, execution on the CPU cores averaged 40.2 seconds with little variability, whereas executing on a single core and GPU took 47.8 seconds but suffered additional variability per run. This highlights how Ray performs best with longer-duration jobs, as the overhead of constructing a cluster is high for shorter jobs.

While only an anecdotal example, the result reinforces the need for wisely assigning jobs when executing across heterogeneous hardware systems. We seek to better understand optimizing this CPU/GPU tradeoff when presented with large workloads of hundreds of mixed HPC and ML applications across heterogeneous campus clusters and beyond. To do so, we next examine Ray application deployment on both CloudLab and FABRIC. The code we use can be found on the github site associated with our project web site [16], while other artifacts including images, profiles and notebooks are available on the testbeds themselves.

B. Using Ray on CloudLab

1) Methodology: Ray's native cluster management capabilities, autoscaling, and ease of exploiting hardware accelerators and multiprocessing make it an ideal candidate for use on computing clusters and research infrastructures unencumbered by job schedulers (e.g., slurm). CloudLab is one such widely used experiment testbed for research and education that supports research on cloud architectures, distributed systems, and applications.

2) Results: We studied the performance of Ray on a cluster of 5 Cisco UCS c240g5 nodes on the CloudLab Wisconsin site. A public CloudLab profile to deploy the topology is available as *ray5node*. Each bare metal server had 2 Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, and a Tesla P100 GPU. We created a public disk image *ray-cuda-pytorch22.04* with Ubuntu 22.04, Nvidia Driver v.550.144.03, and the CUDA Toolkit 12.4.

We first examined Ray performance on a cluster of 1 physical node executing one or more instances of the Pytorch Lightning Image Classifier training run. Here an examination with *btop* indicated that the cores were lightly used. The job running time was 85 sec. Increasing the workload to 5 simultaneous runs extended running time to only 89 sec., however monitoring with *nvtop* reveals that the added work consumed up to 50% of the GPU.

We next used Ray to manually construct the 5 node cluster. Specifying the head and each worker on multi-homed servers is simply a matter of running these commands:

```
head: ray start --head --node-ip-address=
            '128.105.144.55:6379'
workers: ray start --address='128.105.144.55:6379'
     --node-ip-address= 128.105.144.[44,45,57,59]
```

We next examined application behavior on the 5 node cluster using 10 concurrent instances of both the matrix inversion tasks and the image training task. The training task helped highlight the value of extensive Ray logging; Fig. 4 shows an offline TensorBoard plot of the converging training accuracy over the 10 execution epochs for one of the training runs.

We elected to use the ray *job* command to submit jobs to the cluster:

```
RAY_ADDRESS='http://127.0.0.1:8265' ray job submit
    --working-dir . -- python ./invertMatrix.py &
```

This job submission entrypoint serves to supplement the Ray Dashboard by allowing us to examine the status of job execution. This permits us, for example, to see a report (not shown) on the execution of the 10 matrix inversion tasks (960 inversions), clearly showing that all 200 cores were being used for much of the job duration.

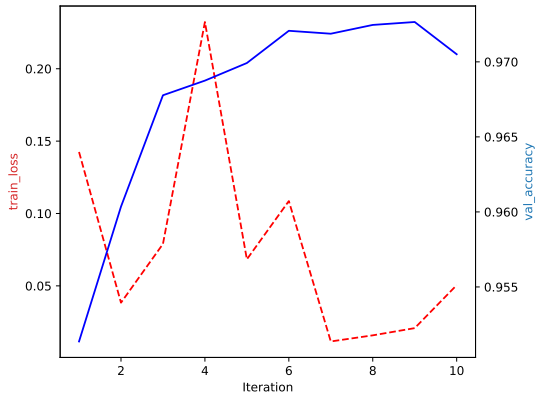

Fig. 4: Visualizing the performance of a Pytorch Lightning Image Classifier training run on a CloudLab c240g5 GPU node using ray[train] with TensorBoard. Convergence of the training to the target threshold is shown on the right axis (solid, blue), and training loss on the left axis (dashed, red).

C. FABRIC as a distributed computing substrate

1) Methodology: With its extensive global networking footprint, the FABRIC research infrastructure serves as an important resource for computer *networking* research. It is less commonly thought of as an integral component of a globally decentralized community ***computing*** facility. Yet as a 'testbed of testbeds,' FABRIC hosts native computing resources, and also interconnects external site-based, shared community testbed, and cloud computing resources. Hence, FABRIC can be tapped to assemble Ray clusters using computing resources that are

exclusively native, host site-based, testbed-based, or cloud-based. Even more interestingly, FABRIC permits Ray clusters using resources that cleverly span these platforms to form either 'hybrid Ray clouds' or 'Ray multiclouds.'[1]

FABRIC offers key capabilities to support this model, including horizontal scaling, in-network computing, edge computing, and high-performance networking. In addition, *Facility Ports* can expose shareable campus-owned compute resources via a dedicated FABRIC attachment networking link [17], permitting the rapid assembly of virtual community infrastructure via a 'Bring Your Own Equipment' (BYOE) model.

Ray offers the potential to investigate wide-area distributed computation across FABRIC's attached heterogeneous computing resources. Ray stripes bulk transfers across multiple TCP connections, so even lightly network-intensive computing applications can be studied across high-speed WAN connections. Of course, FABRIC in-network resources are limited and don't represent a significant general-purpose computing resource. However, their vantage points represent a potentially valuable means for monitoring, learning, and dynamically optimizing either 1) self-driving FABRIC operation [18]; or 2) the control of highly geographically distributed computing execution (e.g., balancing workloads over sites).

To better understand how FABRIC can serve as a platform for distributed computing and community resource sharing, we have experimented with Ray clusters on and across multiple FABRIC sites. Constructing these topologies and methods for deploying the Ray framework will be immediately familiar to FABRIC experimenters. We first examined a local Ray cluster spanning private Princeton University site resources and local native FABRIC VMs (i.e., PRIN site resources) connected via a Facility Port. Here we used the notebook *facility_port_2local-PRIN.ipynb* available at the Ray project site github. The purpose of this activity was to understand Ray behavior on FABRIC resources which were essentially isolated, under our direct control, and easily instrumented with network protocol analyzers. That is, the topology served to establish performance baselines.

2) Results: Exploring Ray performance on heterogeneous GPU nodes at various sites was of particular interest. We began testing Ray on individual VMs, again turning to the MNIST Image Classifier example. This included experimenting with an AMD EPYC 7532 32-core CPU with Nvidia TeslaT4 at the Utah site, and an EPYC 7542 32-core dual socket CPU with a more capable Nvidia A40 at the CERN site. Nonetheless, we frequently observed comparable overall execution times, though this is unsurprising for short job durations (e.g., 75 secs.) and small-sized VM instances that are CPU-bound.

The CERN site's A40 GPU node(s) allowed us to more deeply examine the benefits of a Datacenter GPU. Each VM was assigned 2 cores, and we ran Ubuntu 24.04 with Nvidia driver v.565.57 and CUDA toolkit v.12.7. To exercise the GPU we computed the *Mandelbrot set* various ways [19]; Table II presents those execution times.

TABLE II: Mandelbrot set computation.

Execution environment	**time (sec.)**
CPU-only with numpy	4.11
CPU-only with Numba JIT	0.35
CPU + GPU with cuda.jit	
(with CPU array copy overhead)	0.61
(w/o CPU array copy overhead)	0.17

We did find that our understanding of job execution was improved by using Ray's native machine microbenchmarking facilities as shown in Fig. 5. While the performance tests are highly specific to Ray runtime execution, they were nonetheless valuable to understand the tradeoffs to make between selecting nodes with more compute cores, or more capable GPUs. This provides an experimenter with insights into preferred task and job placements for their application workload.

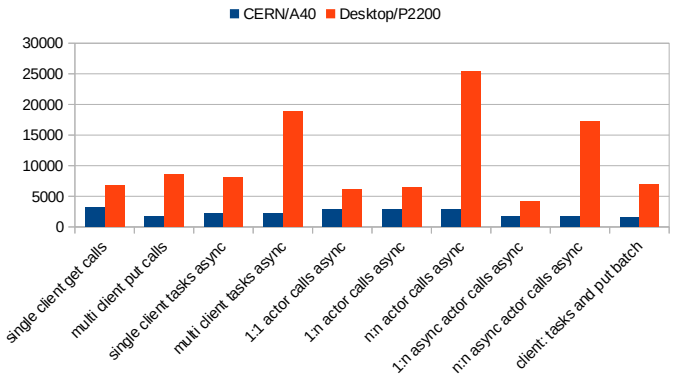

Fig. 5: Ray microbenchmarks provided a quick indication that a bare metal 20-core desktop with a modest workstation-class P2200 GPU would outperform a single FABRIC VM at CERN with a Data Center-class A40 GPU.

Finally, we examined a 2-node (4 core) cluster at CERN (using both available A40 GPUs). An illustrative notebook *ray_A40_2nodes.ipynb* for building the topology is available at the Ray project github site. For comparison with earlier results for Ray on a single node and on a 5 bare metal node (200 core) CloudLab cluster, the 2 node CERN cluster completed the matrix inversion task in a slow 1 min. and 54.0 secs, as the few cores were CPU bound. The same situation was revealed for the MNIST image classifier tasks, with a single task taking 1 min. and 29 sec and 10 concurrent tasks consuming a long running time of 4 min. and 55 secs. Such results could help argue the case for an assignment of more cores per VM on FABRIC's GPU nodes.

4. Conclusion

From desktops and local clusters to shared mid-scale infrastructures including CloudLab and FABRIC, we have demon-

[1]While a *cluster* is the Ray framework term of art, in certain cases a 'federation of loosely-coupled Ray clusters' might be a more appropriate label.

strated that Ray is a valuable tool to support distributed systems research across the computing continuum.

Ray offers experimenters simple and powerful programming abstractions that are less cumbersome than lower-level alternatives such as message passing interfaces. Ray conveniently offloads cluster management from experimenters, handles node heterogeneity, and eases multiprocessing. Ray takes a step toward achieving a "code once, run anywhere" model, allowing students and domain scientists to focus on their science. While one distributed framework will never adequately satisfy the needs of campus computational scientists, Ray can help use campus' heterogeneous resources efficiently, and ease scaling of computational models. This is particularly valuable in settings where many domain scientists and students have limited expertise in distributed systems programming techniques.

Much more research needs to be done to better understand the many tradeoffs in recommending and supporting a framework. In our future work we will study larger and more representative mixes of anticipated HPC and ML workloads. This testing will be performed at scale on conventional slurm-managed HPC clusters, and other large GPU clusters specifically tailored for studying Large Language Models. Finally, we will also examine Ray's performability when used to implement emerging Reinforcement Learning applications on increasingly capable edge computing networks.

REFERENCES

[1] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, 'Ray: A Distributed Framework for Emerging AI Applications', *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, October, 2018.

[2] 'Overview of Ray Python Notebook', https://github.com/ray-project/ray/educationalmaterials/blob/main/Introductory_modules/Overview_of_Ray.ipynb.

[3] Tom Reid, 'An introduction to distributed computing using the Ray library and AWS EC2 clusters', March 28, 2022, https://www.linkedin.com/pulse/introduction-distributed-computing-using-ray-library-aws-tom-reid/.

[4] Tingkai Liu et al, 'Cloud-Bursting and Autoscaling for Python-Native Scientific Workflows Using Ray,' *Lecture Notes in Computer Science*, Vol. 13999, August 2023.

[5] N. Machev, 'Spark, Dask, and Ray: Choosing the Right Framework', Domino Data Lab, Sep. 2021, https://domino.ai/blog/spark-dask-ray-choosing-the-right-framework.

[6] Ilya Baldin, et al, "FABRIC: A national-scale programmable experimental network infrastructure", *IEEE Internet Computing*, Vol. 23, No. 6 pp. 38–47.

[7] Dmitry Duplyakin et al, "The Design and Operation of CloudLab", *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1-14, July 2019.

[8] J. Dean, S. Ghemawat, 'MapReduce: Simplified data processing on large clusters,' *Communications of the ACM*, Vol. 51, Iss. 1; Jan. 2008, 107–113.

[9] K. Shvachko, H. Kuang, S. Radia and R. Chansler, 'The Hadoop Distributed File System,' *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[10] Martín Abadi, et al, 'TensorFlow: A System for Large-Scale Machine Learning,' *OSDI'16*, 2016.

[11] Adam Paszke, et al, 'PyTorch: An Imperative Style, High-Performance Deep Learning Library', *33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[12] 'Modified National Institute of Standards and Technology (MNIST),' https://paperswithcode.com/dataset/mnist.

[13] J. Deng, W. Dong, R. Socher, L.J. Li, Kai Li, Fei-Fei Li, 'ImageNet: A large-scale hierarchical image database,' *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[14] Anyscale Ray Team, 'Ray v2 Architecture,' October 2022, https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI/preview?tab=t.0#heading=h.iyrm5j2gcdoq.

[15] Anyscale Inc, 'Train a PyTorch Lightning Image Classifer,' https://docs.ray.io/en/latest/train/examples/lightning/lightning_mnist_example.html.

[16] Jack Brassil, 'Ray project page,' https://www.cs.princeton.edu/∼jbrassil/public/projects/ray/.

[17] FABRIC Facility Ports, https://learn.fabric-testbed.net/knowledge-base/fabric-facility-ports/.

[18] Nick Feamster, Jennifer Rexford, *Report on the Workshop on Self-Driving Networks*, https://www.cs.princeton.edu/∼jrex/papers/self-driving-networks18.pdf, Princeton NJ, Feb. 15-16, 2018.

[19] 'A Numba Mandelbrot Example,' https://github.com/harrism/numba_examples/blob/master/mandelbrot_numba.ipynb

[20] 'TensorBoard: TensorFlow's visualization toolkit', https://www.tensorflow.org/tensorboard.

[21] Max Pumperla, Edward Oakes, Richard Liaw, *Learning Ray* February 2023, O'Reilly Media, Inc., ISBN: 9781098117221, https://github.com/maxpumperla/learning_ray/?tab=readme-ov-file#learning-ray—flexible-distributed-python-for-machine-learning.