Deep RC: A Scalable Data Engineering and Deep Learning Pipeline

Arup Kumar Sarker[1,3], Aymen Alsaadi[2], Alexander James Halpern[1], Prabhath Tangella[1], Mikhail Titov[4], Niranda Perera[7], Mills Staylor[1], Gregor von Laszewski[3], Shantenu Jha[2,5,6], and Geoffrey Fox[1,3]

[1] Department of Computer Science, University of Virginia, Charlottesville, VA 22904 {djy8hg, upy9gr, nww7sm, vxj6mb}@virginia.edu
[2] Rutgers-New Brunswick, NJ, 08901-8554 {aymen.alsaadi, shantenu.jha}@rutgers.edu
[3] Biocomplexity Institute and Initiative, Town Center Four, 994 Research Park Boulevard Charlottesville, VA 22911 laszewski@gmail.com
[4] Brookhaven National Laboratory, Upton, NY mtitov@bnl.gov
[5] Princeton Plasma Physics Laboratory, Princeton, NJ
[6] Princeton University, Princeton, NJ
[7] Nvidia Corporation, Santa Clara, CA niranda@niranda.dev

arXiv:2502.20724v2 [cs.DC] 22 Apr 2025

Abstract. Significant obstacles exist in scientific domains including genetics, climate modeling, and astronomy due to the management, preprocess, and training on complicated data for deep learning. Even while several large-scale solutions offer distributed execution environments, open-source alternatives that integrate scalable runtime tools, deep learning and data frameworks on high-performance computing platforms remain crucial for accessibility and flexibility. In this paper, we introduce Deep Radical-Cylon(RC), a heterogeneous runtime system that combines data engineering, deep learning frameworks, and workflow engines across several HPC environments, including cloud and supercomputing infrastructures. Deep RC supports heterogeneous systems with accelerators, allows the usage of communication libraries like `MPI`, `GLOO` and `NCCL` across multi-node setups, and facilitates parallel and distributed deep learning pipelines by utilizing Radical Pilot as a task execution framework. By attaining an end-to-end pipeline including preprocessing, model training, and postprocessing with 11 neural forecasting models (PyTorch) and hydrology models (TensorFlow) under identical resource conditions, the system reduces 3.28 and 75.9 seconds, respectively. The design of Deep RC guarantees the smooth integration of scalable data frameworks, such as Cylon, with deep learning processes, exhibiting strong performance on cloud platforms and scientific HPC systems. By offering a flexible, high-performance solution for resource-intensive applications, this method closes the gap between data preprocessing, model training, and postprocessing.

1 Introduction

It is a significantly complex and challenging task to manage and prepare massive data for deep learning in big data science which introduces additional difficulty in model training causing an impact on fields such as hydrology, genomics, climate modeling and astronomy [13]. Furthermore, the integration of data from diverse sources not only introduces significant heterogeneity and multifaceted structure but also manifests complex interdependencies among parameters, thereby exacerbating the challenges associated with the bare metal process. The sheer magnitude and complexity of these enormous datasets place significant processing limitations on even the most advanced computing infrastructures, frequently resulting in computational bottlenecks that impede effective data analysis and interpretation. [23]. Google Pathways [5, 7] and OneFlow [28] address some aspects of these challenges. But the design and distributed runtime of those systems are black-box and there is no way to measure the comparative results of a heterogeneous data pipeline. We aim to develop a unified approach combining data engineering and deep learning frameworks with diverse execution capabilities, which can be deployed on various HPC platforms, including cloud systems and HPCs.

To address this, Cylon [27] offers foundational frameworks for data engineering on scalable HPC machines. DASK [21] and SPARK [29] Dataframe can be alternative to Cylon. However, Cylon outperforms both in multiple scaling operations [2, 19, 27] although Cylon lacks optimization for efficient resource use and does not support a heterogeneous data pipeline. On the other hand, our previous work Radical-Cylon [23], a task-based architecture, enables Cylon to interact and operate with different HPC platforms seamlessly, shielding Cylon from heterogeneous configurations of different HPC platforms that support data engineering pipelines. However, it lacked support for deep learning execution and the use of heterogeneous resources. To execute the deep learning job by using the cylon dataframe, we had to preprocess the dataframe separately, and then use it as a dataset for the deep learning model. Moreover, It has limitations supporting GPUs and a combination of CPU-GPU execution based on the platform. On the other hand, RAY [15] provides a distributed runtime that can be an alternative to RADICAL-Pilot [14] as a workflow engine. However, Ray's GPU support is restricted to scheduling and reservations, whereas Radical pilot has the flexibility to control underlying hardware resources for scheduled tasks. We aim to enhance support by introducing Deep Radical-Cylon(RC), a heterogeneous runtime system with accelerators, enabling frameworks like Cylon with Pytorch and Tensorflow to leverage heterogeneous execution

through RADICAL-Pilot and handle data pipelines from data engineering tasks with cylon-distributed data frame to downstream deep learning training and inferencing jobs. By separating resource management from the application layer, deep RC makes it possible for a job to execute on any HPC platform without the need for code rewrites or changes. This makes development easier and produces a system that is more adaptable and loosely integrated frameworks. In comparison to bare-metal deep Learning execution, this method achieves competitive performance with little overhead by enabling the usage of heterogeneous communicators across numerous nodes. It offers enhanced functionalities such as distributed data pretreatment and post-processing inside a unified pipeline, accommodating heterogeneous data and model executions. Deep RC has been evaluated with 11 NeuralForecast [16] models for PyTorch and hydrology models [12] for TensorFlow with inferencing jobs and reduces 3.28 and 75.9 seconds respectively.

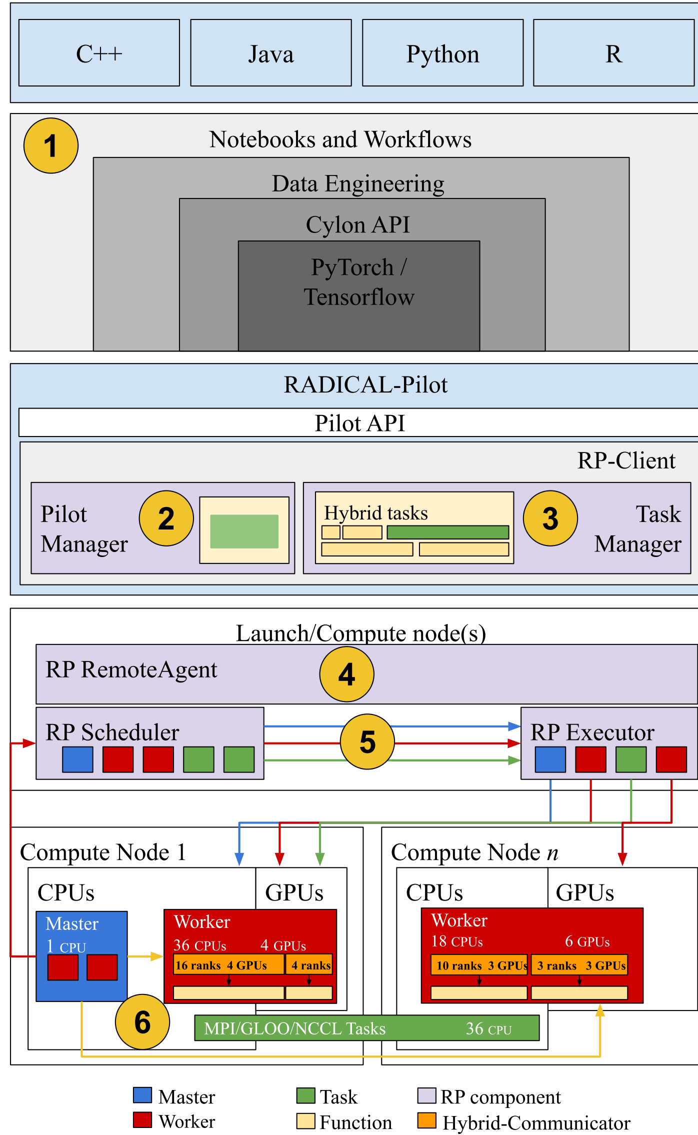

Fig. 1. Deep RC design incorporating Radical-Cylon [23] Architecture with GLOO and NCCL as communication framework. A modular design with dependent components. Segregated independent module with top-down flow from cross-platform to hardware resources. Application pipeline is incorporated with Cylon APIs and PyTorch and TensorFlow with proposed Deep RC Bridge in Fig.-2

2 Design and Implementation

2.1 Cylon

To improve performance and lower the complexity of distributed data operations, Cylon is designed to optimize Extract, Transform, and Load (ETL) procedures [27]. At the heart of Cylon's design lies a sophisticated core

framework centered on a table abstraction that represents structured data. This abstraction ensures a cooperative solution to distributed computing difficulties by allowing individual ranks or processes to handle data partitions jointly. Local and distributed operations are intended to be balanced by the Python and C++-based interface. Tasks involving only locally accessible data are handled by local operators, guaranteeing prompt and effective completion. Distributed operators(e.g. shuffle, gather, reduce, etc.), on the other hand, use sophisticated network capabilities to carry out intricate operations that call for inter-process communication, like data collecting, reduction, and shuffling.

Cylon incorporates network-level operations based on communication protocols such as TCP and Infiniband to handle the inherent challenges of distributed programming. Heterogeneous data transmission is made possible by these protocols' support for several communication abstraction frameworks, including MPI, UCX [25], and GLOO [10]. This channel abstraction technique is essential for improving performance in a variety of hardware settings and simplifying communication across dispersed processes. High-efficiency operations like shuffle and reduce that depend on these abstractions allow for scalable and smooth distributed workflows [18, 26].

Cylon's data model is made compatible and interoperable by using Apache Arrow's Columnar Format as its foundation. Because of this, Cylon may easily interact with a wide range of open-source frameworks and libraries, guaranteeing compatibility and seamless data flow across the broader data ecosystem. This foundation encourages innovation in the creation of high-performance applications while supporting a consistent and effective approach to data processing. Cylon's architecture combines sophisticated communication techniques, local and distributed operators, and strong data models to create a high-performance distributed framework [19]. Additionally, it is versatility and useful in coordinating intricate workflows across disparate computer resources are highlighted by its abstraction under RADICAL-Pilot.

2.2 Radical-Cylon

We published Radical-Cylon [23] for heterogeneous data engineering jobs seamlessly integrating Cylon and RADICAL-Pilot (RP). RP is a flexible runtime system that effectively manages the concurrent execution of highly heterogeneous workloads on diverse heterogeneous resources. RP makes deploying complicated workloads across various HPC resources easier by abstracting resource management through a flexible pilot-based approach. The PilotManager, TaskManager, and RemoteAgent are the three primary parts of the system that provide a simplified execution environment on HPC resources. The PilotManager is responsible for managing the lifecycle of the pilot, a placeholder that acquires and manages the computes resources. It ensures that resource requests and allocations are managed effectively by operating on user-accessible resources, such as a local computer or a cluster's compute nodes. Tasks represent the user applications, which can be executables or functions managed by TaskManager [14]. The TaskManager works closely with the PilotManager to ensure efficient and effective task management and scheduling to maximize resource utilization.

When installed on an HPC system's computing nodes, the RemoteAgent sets up the execution environment and controls task execution. It guarantees smooth communication between the workload and the resources allotted, enabling RADICAL-Pilot to effectively carry out activities across dispersed nodes. Even in the most taxing computational situations, RADICAL-Pilot can provide high throughput and scalability thanks to its decentralized yet integrated design. It supports a diverse range of workloads, including MPI/GLOO/NCCL-based tasks, as well as single-threaded, multi-threaded, and multi-core applications. By offering such flexibility, RADICAL-Pilot caters to a variety of use cases, from large-scale simulations to data-intensive workflows, making it a vital tool for researchers and engineers working on HPC platforms [14].By using a loosely coupled approach, the integration makes full use of both systems' capabilities via the RADICAL-Pilot API and eliminates the need for further integration plugins. Without needing any changes to Cylon activities, this design enables Cylon to leverage the heterogeneous runtime characteristics of RADICAL-Pilot, specifically, its capacity to generate and control MPI communicators [23].

2.3 Design

Deep learning model execution, Cylon and RADICAL-Pilot are three distinct systems that serve complementary purposes, each offering specialized functionalities and capabilities. The loosely linked architecture that underpins the end-to-end pipeline enables them to function independently while utilizing one another's advantages. This strategy minimizes dependencies and maximizes modularity by ensuring that neither system is firmly tied to the other. When workloads or computing demands change, the system can be scaled out, enlarged, or reduced. Furthermore, the absence of direct dependencies guarantees that any changes made to either system's existing components or the addition of new ones won't interfere with the integrated framework. Without requiring changes to the other system, each system can implement updates, add or remove components, or rethink essential features.

Cylon is a high-level component in this pipeline that is in charge of developing and overseeing tasks like executables or data engineering functions. RADICAL-Pilot is utilized as a workflow engine for execution on

HPC resources after these tasks are sent to a deep learning framework as an input dataframe for training and inferencing jobs [8]. The native APIs of the three systems, which both offer straightforward and adaptable Python-based interfaces, enable communication between them. Task handoff and execution are made possible by this architecture, which guarantees a smooth connection between the two without the need for intricate intermediary layers. For instance, RADICAL-Pilot schedules these operations to optimize the flow of data to CPUs or GPUs, while Cylon can preprocess big datasets into forms compatible with PyTorch's `DataLoader` [17] or TensorFlow's `tf.data.Dataset` [1].

In terms of fault-tolerance, this integration is very robust. The overall stability of the framework is maintained since failures in one system or its components do not ripple into the other system. For example, RADICAL-Pilot is unaffected and can carry on with other tasks even if a task fails during the model training phase. Similarly, the deep learning framework's task management capabilities are unaffected by any errors or resource limitations in RADICAL-Pilot. The system can recover gracefully and carry on even in the face of adversity thanks to this isolation of failures. The design philosophy behind the pipeline exemplifies a robust and forward-looking approach to creating modular, scalable, and fault-tolerant frameworks. Through this interface, data engineering and deep learning workflows on HPC platforms are executed with flexibility and resilience, utilizing the benefits of both systems while preserving their independence. This architecture guarantees flexibility for upcoming advancements in both systems in addition to meeting present computing demands.

2.4 Implementation

Deep Radical-Cylon(RC) is a unified system that facilitates seamless communication between Cylon, RADICAL-Pilot, and deep learning frameworks through their Python APIs. To define, organize, and carry out deep learning jobs across many HPC systems, the architecture makes use of RADICAL-Pilot as the main interface. A `RadicalPilot.TaskDescription` object, which outlines resource requirements such CPUs, GPUs, and memory allocations, is used to represent each deep learning task. When specifying the computational requirements of each activity, this structured representation guarantees accuracy and clarity.

In order to instruct the PilotManager to generate a Pilot object with the necessary resources at initialization, Deep RC uses RADICAL-Pilot. On HPC systems, this Pilot acts as a stand-in for resource management. At the same time, RADICAL-Pilot creates a TaskManager that is in charge of sending deep learning tasks, like model training or inference, as well as Cylon-specific tasks to be carried out on distant computing resources. The system works in tandem with the HPC resource manager, guaranteeing efficient coordination between task execution and resource acquisition. The RemoteAgent is deployed on the compute nodes by RADICAL-Pilot after the necessary resources have been assigned. RADICAL-Pilot deploys a multi-node master-worker execution environment, which offers the capabilities to dynamically construct MPI-Communicators to run heterogeneous NCCL/GLOO/MPI operations on several nodes at once (Fig.-1). Such an approach allows the efficient execution of concurrent AI and HPC tasks, which frequently call for unique communication patterns among a subset of resources, to benefit greatly from this characteristic.

The RemoteAgent's scheduler sends a queue of Cylon and deep learning tasks to the master(s), and ultimately, each master distributes the tasks to the worker(s). Based on the particular resource requirements of the task, a worker isolates a set of parallelism upon receiving it. The task's communicator is then built and delivered at runtime, allowing for effective execution and task-specific communication. The master process gathers the outcomes of tasks as they are finished and sends them back to the TaskManager. Thanks to this organized feedback loop, all job results are centralized and made accessible for additional processing or analysis. RP's dynamic and adaptable features enable the integration of Cylon and deep learning frameworks within RADICAL-Pilot, resulting in a highly scalable and effective solution for carrying out intricate processes on HPC platforms. This architecture is an effective tool for large-scale data engineering and deep learning execution since it not only improves resource usage but also simplifies the execution of heterogeneous activities.

The ability to use GLOO [10] and NCCL [9] as the communication backend for distributed operations is one of the enhanced advantages of the suggested architecture on Deep RC. The design encourages a loosely connected approach where both systems benefit from one another's advantages while functioning separately with few interactions and dependencies. We provide a distributed data loader for the Cylon data frame so that the preprocessed Cylon dataframe can be used as input for the deep learning system. We introduce two kinds of bridges: Data Bridge, which connects Cylon to the deep learning framework, and System Bridge, which manages the resources and flow from Cylon to RADICAL-Pilot(RP). Deep RC is the name given to the complete system.

In Fig.- 2, we demonstrated the Deep RC bridge, which allows data to be preprocessed using Cylon distributed dataframes that operate on top of MPI/UCX/GLOO and produce a Cylon Global Table (GT). The global table, which may be zero-copied and translated to pandas and other data frame formats, is used to create the distributed Cylon dataframe. Before batches are loaded into memory, data transformations and augmentation are frequently implemented. This guarantees that the training process is not slowed down by preprocessing operations like shrinking photos, standardizing values, or applying random augmentations. Our custom data loader was designed to inherit the functionality of the default PyTorch/TensorFlow data loader, enabling it to

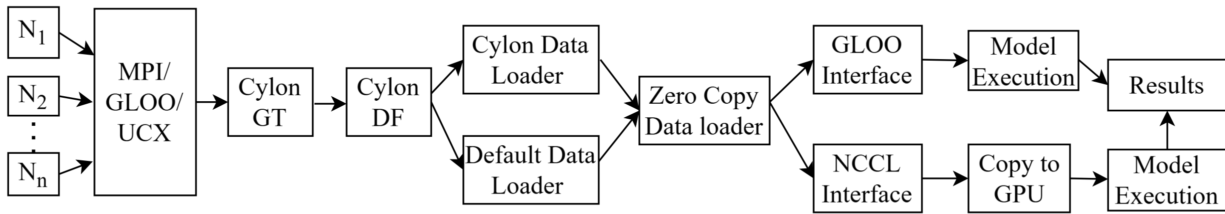

Fig. 2. Deep RC Bridge. From distributed data pre-processing to deep learning model execution. Cylon distributed Tasks are scheduled and executed on CPUs with MPI/GLOO/UCX communication frameworks.

efficiently traverse the dataset as required. A batch of `train_features` and `train_labels` are returned at the end of each iteration. Since it requires no extra memory to build training and verification datasets, we dubbed it a zero-copy data loader. The zero-copy data loader uses several workers to retrieve and preprocess data in simultaneously, effectively managing data loading in the Deep RC pipeline. It divides the workload among several subprocesses, each of which is in charge of loading a section of the dataset, rather than depending on a single process to load data sequentially. By preventing training bottlenecks, this parallelism makes sure the model gets data as soon as possible. The main objective is to minimize waiting time for data preparation while optimizing execution and memory use.

When data is loaded, it is frequently saved in memory before being transferred to the GPU for training via the NCCL interface. Pinged memory, a unique kind of memory allocation that permits DMA transfers between the CPU and GPU, is used by zero-copy data loaders to increase efficiency. By doing this, the overhead of transporting data is decreased, guaranteeing that the GPU gets it fast. Additionally, a queue of batches is kept ready before the model requires them by using data prefetching. Training can continue uninterrupted and resource usage is maximized by overlapping data loading and calculation. Each worker in the pipeline processes a fraction of the data while working under the radical-pilot process management. Each worker operates as a distinct process when multiprocessing is enabled, therefore memory sharing is not the default setting.

Effective memory management is a key feature of Deep RC when working with big datasets. Frameworks such as PyTorch's `DistributedSampler` make sure that each GPU receives a distinct slice of the dataset when many GPUs are being used, avoiding redundancy. Batch sizes also need to be carefully considered; although lower batch sizes may use less memory but result in more updates per epoch, bigger batch sizes speed up training but use more memory. Training performance can be greatly impacted by properly adjusting the batch size, memory settings, and number of workers.

In Fig.- 3, Cylon is connected as a top-level client program to submit different Deep RC tasks (services, functions, or executables) to RADICAL-Pilot for multi-node (CPUs and GPUs) execution. It follows master-workers architectures, where a number of worker nodes are managed by one or more masters, each of whom may have one or more central processing units.

Let's now execute a program that calls the RP-Client(1) and does many computations using the Deep RC system. The Pilot Manager then assigns virtual devices for calculations that have not yet been performed and registers these calculations with the Resource Manager (2). After that, the RP-Client gives the background server instructions to execute the pilot manager's commands, accounting for various calculations, including device network connections and data routing (3,4). If a program's virtual device remains constant, the generated representation can be reused fast. However, if the Resource Manager changes the virtual device of a program, recompilation is necessary. Together, these three stages comprise Deep RC's front end. Remote Agent creates several execution pipelines with two persistent daemons, an executor and a scheduler, that can communicate in order to enable distributed coordination, which is the control plane communication (5, 6, 7). The executor invokes the Deep RC bridge to perform local sorting, joining, or deep learning inferencing operations and to construct a distributed data loader(DDLoader) (8). In this scenario, most data plane connections consist of cluster communications involving shuffle, gather, or gradient-sharing activities (9) within the deep learning framework(DL FW). It is noteworthy that the data plane uses the same communication structure; a green arrow indicates lower bandwidth and a blue arrow indicates higher bandwidth.

The main channel of communication between the two systems is through their native APIs because they both provide simple, flexible Python-based interfaces [8]. Both systems are rapidly changing in terms of flexibility, which could result in new, substantial changes to the design or implementation, including the addition or removal of new system components. It is possible to isolate and contain any malfunctioning component, preventing the rest of the system from receiving or carrying out duties.

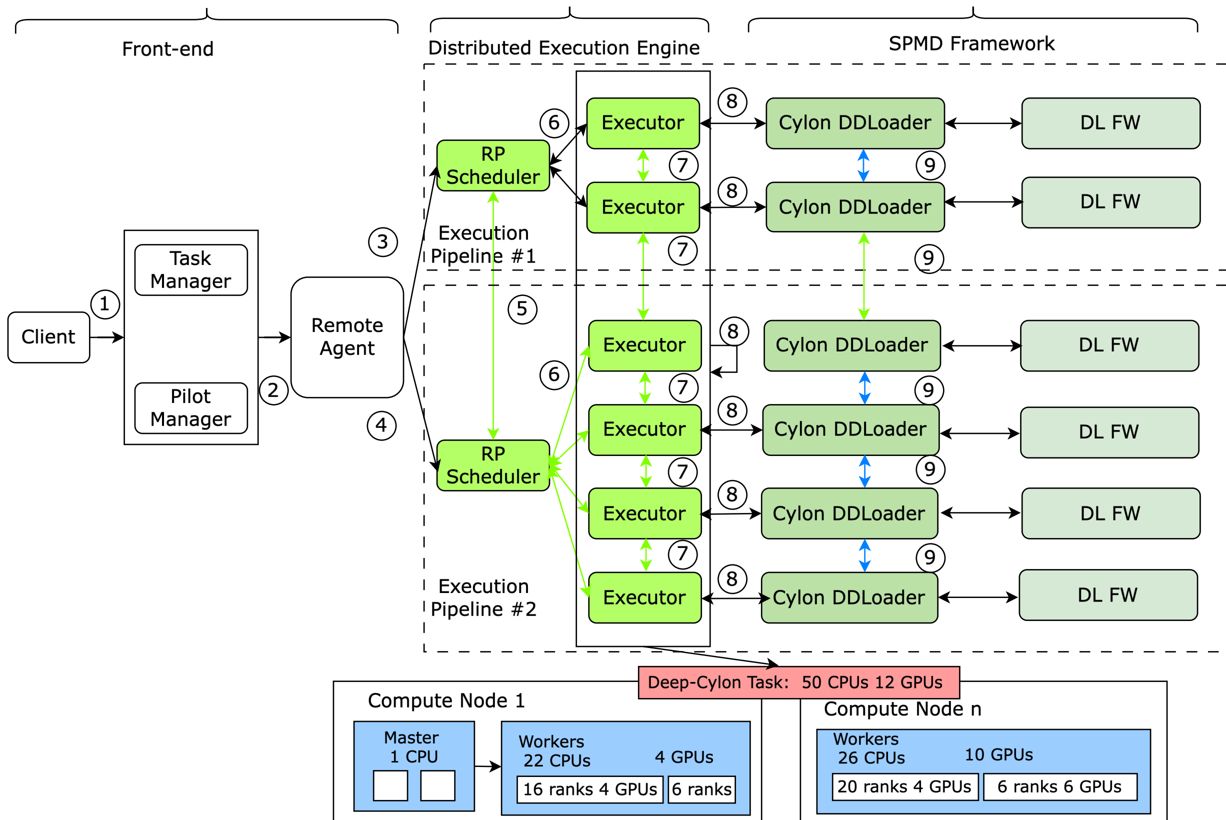

Fig. 3. Deep RC Workflow Architecture. From the bottom-up view, the compute node is a Hardware layer that is compatible with vendor-based CPUs and GPUs. Deep Radical-Cylon(RC) tasks consist of data engineering and deep learning jobs executed on multiple execution pipelines.

3 Experiments

3.1 Experimental Setup

We set up pipeline and scalability studies using UVA Rivanna HPC [20]. We utilize the `parallel` queue on Rivanna, which has a maximum of 16 nodes and 40 cores per node. While performing model training and prediction operations with many pipeline executions, we assess Deep RC's efficiency and contrast it with Bare Metal deep learning. Since Deep RC smoothly supports both PyTorch and Tensorflow, we test the Tensorflow pipeline using the Hydrology Model [12] and the PyTorch pipeline using 11 models from Neuralforecast [16]. In addition to Total Execution Time and Deep RC overheads from the systems, we measure a number of other parameters. The Total Execution Time shows how long Deep RC took to complete the training or prediction tasks using the N-rank computational resources. Time spent deserializing the task object, building the `NCCL/GLOO-Communicator` with N ranks, and delivering it to the tasks is represented by the overheads for Deep RC (mostly RP). Together, the experiments enable us to compare Deep RC's scalability performance to that of Bare Metal(BM) deep learning on Rivanna using a variety of configurations.

3.2 Multiple Data Pipeline scaling operations

We carried out four distinct scaling procedures as part of the data pretreatment in Fig-4 in order to demonstrate a heterogeneous data pipeline. We have shown that both strong and weak scaling operations exhibit the predicted tendencies. When number of worker increase, total execution time should decrease despite additional communication overheads across multiple workers in strong scaling. For weak scaling, the same amount of data is allocated to all workers, who are expected to finish the job with additional communication overheads. It is clearly evident that the system can perform all scaling operations on multiple pipeline with expected behavior. The output of all operations will create a Cylon Global Table(GT) and Dataframe which will be used as input for deep learning model training and inferencing in the downstream task.

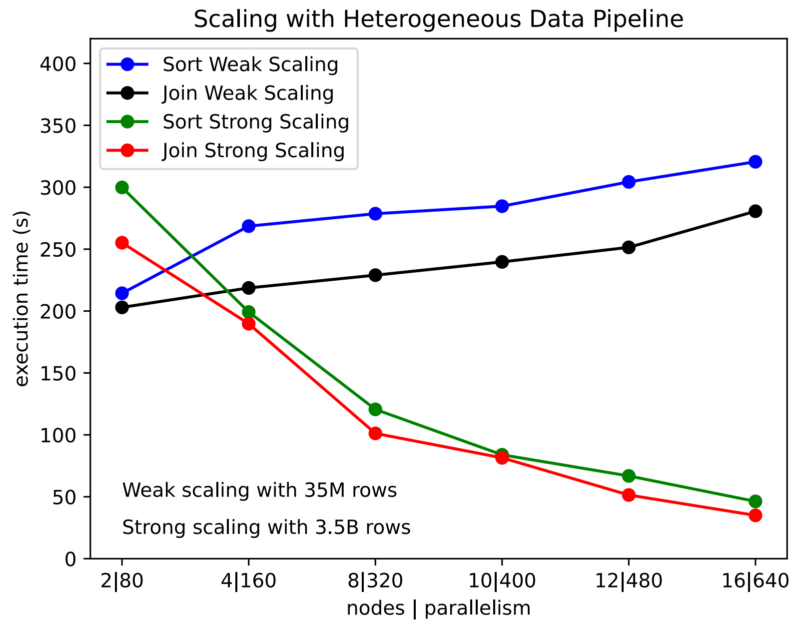

Fig. 4. Heterogeneous Executions with sort and join strong and weak scaling(4) operations on ***rivanna***. Strong scaling with 640 parallelism takes a bit more time due to the lack of rows available for each worker and some workers go idle.

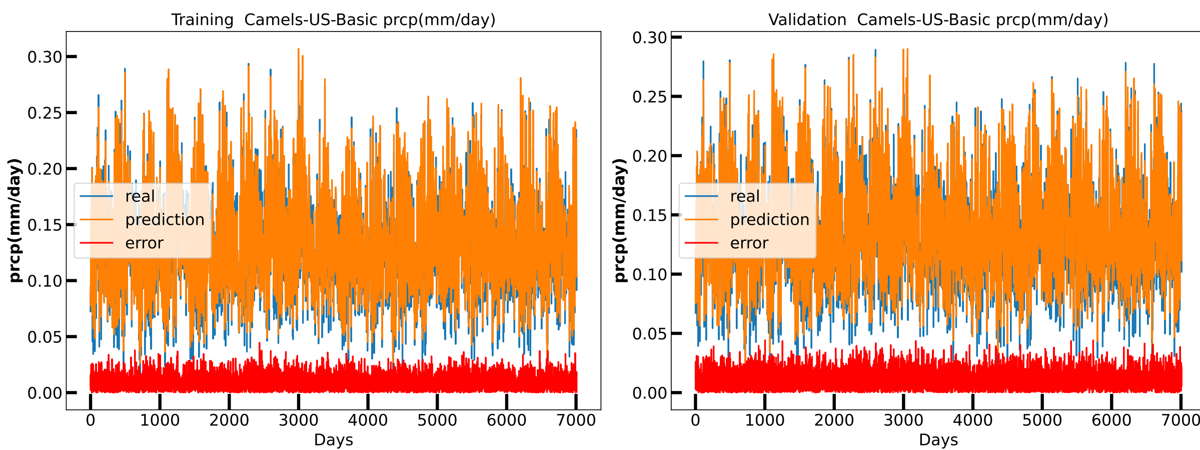

Fig. 5. Training(left) and Prediction(right) accuracy of Precipitation with Camels-US datasets [3] in LSTM Hydrology model.

3.3 Pipeline Testing with TensorFlow based Hydrology Model

Applying a heterogeneous pipeline to inferencing and model training, however, adds complexity to data loading and gradient gathering over several nodes. We have got our input data from the section 3.2. Due to ongoing RP overheads, the overall execution time for both bare-metal and Deep RC executions on Rivanna varies between 4 and 6 hours. All tests are run four times with different parallelisms (a single rank is used for each parallel execution). The experimental findings demonstrate how well the LSTM-based hydrology model captures and accurately predicts important factors. The potential of using deep learning pipelines to advance hydrological research and management is demonstrated by its performance.

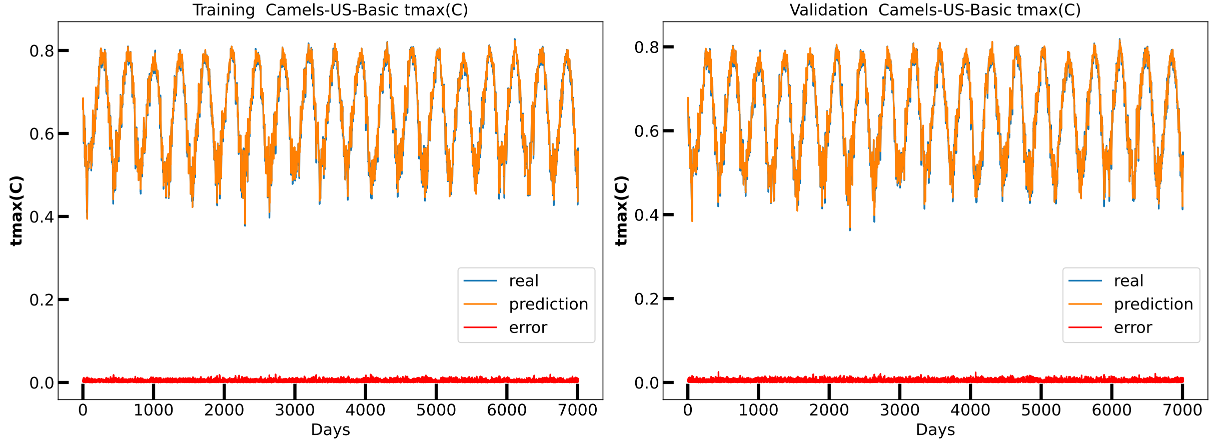

Fig. 6. Training(left) and Prediction(right) accuracy of Maximum Temperature with Camels-US datasets in LSTM Hydrology model.

Table 1. Performance summary with CAMELS US data. Summed time series MSE: CAMELS-US training MSE = 0.004111, validation MSE = 0.004203

Properties	Execution	MSE	NNSE	Execution Time(s)
Precipitation	Train	0.003508	0.820	
	Val	0.003585	0.819	
Mean Temperature	Train	0.000276	0.961	6482.24(Train)
	Val	0.000283	0.960	1927.70(Val)
Streamflow	Train	0.000287	0.806	
	Val	0.000296	0.812	

Key hydrological parameters were successfully predicted by the LSTM-based hydrology model, which achieved a training MSE range of 0.000276 - 0.003508 and a prediction error range of 0.000283 - 0.003585 in Mean Temperature, Streamflow and Precipitation shown in Table- 1. The model's capacity to efficiently learn and generalize temporal patterns in the dataset is demonstrated by these low error values. The model's exceptional accuracy in capturing the relationships between temperature (Fig.- 6), precipitation(Fig.-5), and streamflow(Fig.-8) during training suggests that the LSTM architecture is a good fit for hydrological forecasting.

The prediction performance was particularly outstanding, with errors remaining continuously low over the test dataset. Temperature forecasts showed minimal variance from observed values, showing the model's effectiveness in capturing reasonably smooth temporal trends with minimal execution cost. In Table-2, we have shown model training time with multiple configurations to prove the performance of the system. We see constant overheads between 6 to 8 seconds compared to a total time of 6482.24 to 14456.64 seconds which is very negligible. We performed 11 concurrent validations and shown them in Table-1, column Execution Time(s), and got 1927.70 seconds which is very reasonable, and got similar results(Fig.-7) compared to the Hydrology baseline model [12].

Table 2. Deep RC Execution Time and RP Overheads of different Neural Forecast and Hydrology models with Training Task on Rivanna.

Forecasting Domain	Model	GPUs a100 80GB	Execution Time time (seconds)	Overheads (tasks/second)
Hydrology	LSTM	1	14456.64 ±4.97	4.13 ±1.1
		2	10216.52 ±4.26	4.13 ±1.6
		4	6482.24 ±5.13	5.01 ±1.82
Neural Forecast	Autoformer	2	189.15 ±3.23	3.56 ±0.8
	AutoNHITS	2	500.8 ±3.13	3.51 ±0.41
	TFT	2	298.13 ±3.23	3.12 ±0.9
	TimesNet	2	806.09 ±4.84	4.69 ±0.21
	DeepAR	2	72.32 ±3.35	3.45 ±0.51
	VanillaTransformer	2	269.18 ±4.67	4.56 ±0.33

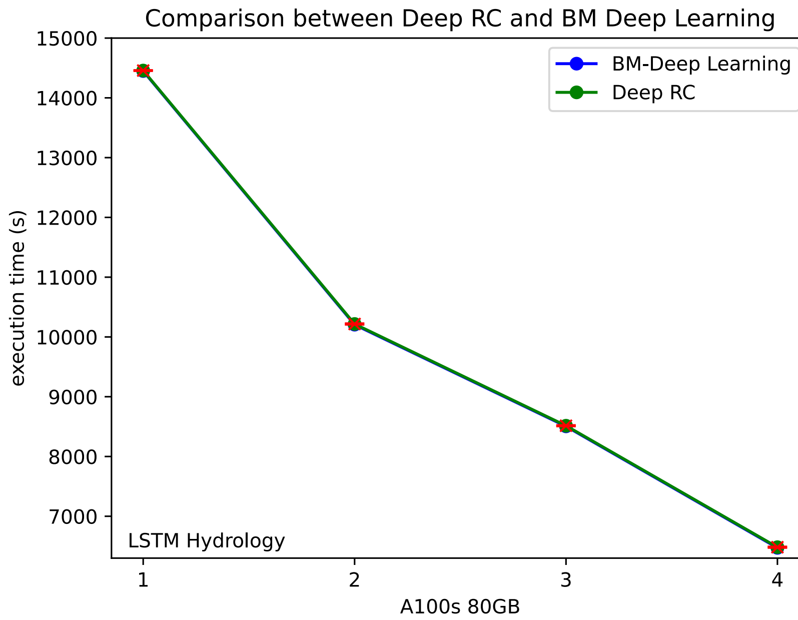

Fig. 7. Execution time comparison. Radical-pilot overhead is so negligible that BM Deep Learning and Deep RC graphs overlap. We used 4 different execution configurations to train the LSTM Hydrology model. BM Deep Learning execution are 14448.81, 10205.37, 8504.53, 6471.71 where Deep RC execution time is shown in Table- 1

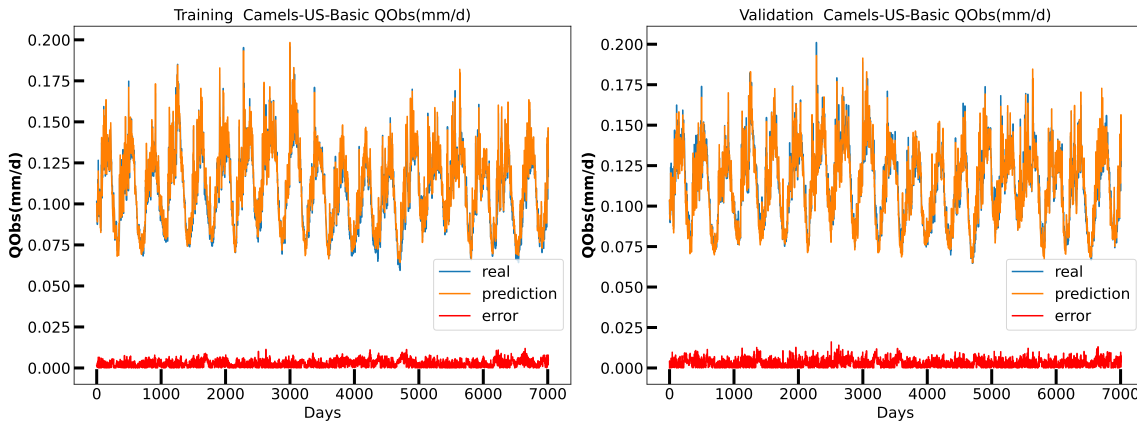

Fig. 8. Training(left) and Prediction(right) accuracy of Observed Streamflow(QObs) with Camels-US datasets in LSTM Hydrology model.

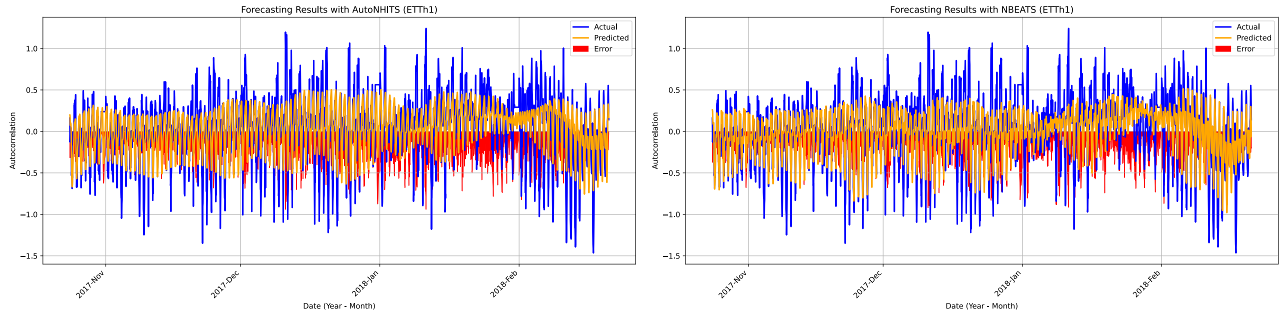

Fig. 9. Training and Prediction accuracy of electrical transformers' oil temperature for AutoNHITS and NBEATS model. The dataset used is the Electricity Transformer Dataset (ETDataset) [30]

3.4 Pipeline Testing with PyTorch based Multiple forecasting Model

Deep RC has the capability to perform deep learning jobs with PyTorch-based models. We choose 11 Neuralforecast models for training and prediction jobs. After training 11 models, the prediction job is run 10 times for each model(total of 110 tasks) as a parallel function. Each parallel execution on Rivanna uses a single rank, and the same scaling configurations are used for the training and inference processes of neural forecast models. We got the expected precision in all models and generated the results with 3 metrics(MAPE, MAE, MSE) shown in Table-3 which are similar to the baseline execution shown in the Neuralforecast experiment [16]. We have shown training and prediction graphs(Fig.-9) for 2 models(AutoNHITS and NBEATS) due to limited space. All results are archived to the github repository.

Although the total execution times of the Deep RC and BM Deep Learning approaches differ by 1 to 5 seconds, training time is less and does not impact on prediction process. Aside from the comparable performance, we observe a constant overhead when using Deep RC in strong scaling operations despite increasing parallelism. Distributed execution presents a set of challenges that include managing data distribution, navigating communication overhead between nodes, and mitigating potential node failures, which are magnified with an increased number of nodes.

Table 3. Single pipeline testing by measuring Mean Absolute Error(MAE), Mean Squared Error(MSE), and Mean Absolute Percentage Error(MAPE) to ensure model training works as expected. Comparison of training time with Deep RC and Bare Metal Deep Learning(BM DL) with 400 epochs. We observe constant overheads(approximately 4.15 seconds on average) in a single pipeline.

Model	Train(s) BM DL	MAE	MSE	MAPE(%)	Train(s) Deep RC
Autoformer	185.51	0.51	0.57	2.65	189.15
DeepAR	72.14	0.50	0.59	2.48	76.32
NLinear	20.98	0.42	0.39	2.45	24.34
GRU	157.15	0.52	0.59	2.18	163.2
NBEATS	15.21	0.39	0.36	2.01	19.03
AutoNHITS	495.78	0.39	0.33	2.22	500.8
PatchTST	47.07	0.37	0.32	2.20	51.15
TFT	293.79	0.42	0.47	1.28	298.13
TimesNet	801.13	0.39	0.36	2.51	806.09
VanillaTransformer	264.57	0.46	0.51	1.43	269.18
TiDE	19.39	0.36	0.34	1.84	23.61

Table 4. Performance Comparison of multiple pipelines with Deep RC and Bare Metal Deep Learning(BM DL) with PyTorch and TensorFlow based models, runs on 2 a100s 80GB GPUs. For cylon job, it uses 8 nodes with 40 cores/node

Pipeline Type	Number of pipelines	Cylon Task	DL Task	BM-DL Execution(s)	Deep RC Execution(s)
Hydrology	11	join	inferencing	21381.73135	21305.83772
NeuralForecast	11	join	inferencing	167.8454124	164.5677196

3.5 Discussions

The results show that Deep RC maintains comparable speed in multi-task execution while achieving effective scaling with low overhead. Furthermore, during heterogeneous execution, it outperforms the BM-Deep Learning model's batch processing capabilities. We have developed 11 pipelines with one Cylon join and 11 deep learning inferencing jobs using an LSTM-based Hydrology and NeuralForecast model in Table- 4. It significantly decreased 75.9 and 3.28 seconds in both experiments, which is important for inferencing tasks. Because any commercial cloud platform that manages thousands of requests would be significantly impacted. In comparison to the overall execution time and the scale of trials, Radical-Cylon consistently generates a `MPI/NCCL/GLOO-Communicator` with numerous ranks in constant time, according to the overheads associated with Deep RC. However, when we attempted to infer LLMs, the RP scheduler and resource allocation module—had trouble assigning resources for Deep RC. To manage such jobs, a design modification incorporating multi-level parallelism is necessary. This significant design modification will be presented separately because we view it as a future work.

Deep RC was created to accommodate a variety of data pipelines and enable unified execution on both CPUs and GPUs. Despite the computational complexity that comes with combining CPUs and GPUs into a single job, heterogeneous execution is still possible by using different rank groups with memory dedicated to either CPUs or GPUs. The initial focus of Deep RC, has been on setting up several data pipelines and using functions to carry out distributed operations; Cylon and deep learning frameworks are essential to making distributed execution possible. As multi-tenancy scenarios get more complicated, RADICAL-Pilot will need to handle a variety of resource types, including CPUs/GPUs allocation and host or device memory. Large-scale resource allocation and reliable monitoring are features of the master-worker paradigm. We plan to support all multi-tenancy requirements, e.g. prioritization, performance isolation, and resource tracking in the future. Compared to earlier efforts in Radical-Cylon, Deep RC seeks to produce scalable and effective deep learning pipelines on much bigger resource pools in a shorter amount of time.

4 Related Works

In order to increase the scalability, effectiveness, and performance of large-scale AI systems, recent developments in distributed computing and deep learning research have placed an increasing emphasis on novel frameworks and architectures. The Ray framework, for instance, has recently been used by researchers to streamline workflows for distributed reinforcement learning [15]. Distributed policy assessment in high-dimensional contexts was implemented using Ray. Compared to conventional MPI-based configurations, the framework's capacity to handle diverse workloads across clusters allowed for a reduction in training time. Another example is the use of Google Research's Pathways architecture into state-of-the-art multimodal AI models. Complex models that needed to analyze text, pictures, and temporal input simultaneously were trained using Pathways. By combining data from many SPMD units, researchers were able to reduce hardware usage and accomplish smooth task coordination by utilizing Pathways' MPMD design [5].

Similarly, current AI research has relied heavily on Apache Spark [29] and Apache Flink [6] to process large datasets. Terabytes of genomic data are preprocessed using Spark's in-memory processing capabilities to enable deep learning-based analysis later on. Flink's ability to handle continuous data streams was demonstrated when its stream-processing capabilities were used in real-time anomaly detection systems for industrial Internet of Things applications. Frameworks like Dask [21] have been widely used in the data analytics field. A gradient-boosted decision tree model's hyperparameter tweaking computational load is divided among GPU clusters using Dask.

CuDF has emerged as a key technique for GPU-accelerated ETL pipelines. CuDF was utilized by researchers to integrate high-resolution video feeds with preprocessed 2D/3D data. Because of this configuration, preprocessing latency was decreased, allowing multi-modal and other decision-making models to infer in real time [22, 24]. CuDF facilitates effective handoff between data loaders and model training pipelines through its interface with frameworks such as TensorFlow, PyTorch, and RAPIDS.ai. There is no need for extra transformations or data copying processes when converting CuDF-prepared data straight into GPU tensors. Even in high-throughput situations, this close integration makes sure that the data flow doesn't become a bottleneck. Researchers used CuDF to preprocess textual input in a recent work on training big language models, which included batching, tokenization, and normalization. The preprocessing speed rose by up to 15x when these jobs were distributed across a GPU cluster using Dask-CuDF, and the training pipeline maintained near-linear scalability as the dataset size increased [11]. Even though CuDF has many benefits, there are still issues in integrating it smoothly in environments that use both CPU and GPU resources. CuDF's compatibility with these hybrid systems is being improved through continuing research, opening the door to even more reliable distributed data loading pipelines.

Frameworks designed specifically for deep learning, such as OneFlow, have also drawn interest. The potential for OneFlow to take the place of conventional workflows based on TensorFlow in large-scale transformer model training [28]. In order to achieve greater scalability, the researchers emphasized OneFlow's simultaneous processing and efficient memory management. In studies on hybrid distributed systems, ZeroMQ has been investigated at

the communication level. ZeroMQ was utilized in a 2024 study to lower communication overhead and latency among participating nodes by enabling asynchronous communication in federated learning configurations [31]. Similar to this, a recent effort that trained generative models on decentralized datasets made use of technologies like Parsl, which enabled better resource allocation and dynamic task scheduling [4].

These papers highlight how important distributed frameworks and architectures are to the advancement of AI and deep learning research. Researchers are pushing the limits of what is feasible in scaled AI systems by combining technologies such as Ray, Pathways, Spark, Flink, and OneFlow. Using Deep RC, we shade a portion of the deep learning execution.

5 Conclusions

In conclusion, Deep RC successfully attains performance parity with state-of-the-art multi-execution designs, confirming its efficacy in the rapidly changing fields of deep learning execution pipelines and data engineering. It offers a unified framework that simplifies model training, prediction, and data processing under a centralized execution paradigm by tackling the challenges of resource management and varied pipeline execution. Through this change, task management becomes more organized and scalable while also improving computing efficiency. Our analysis demonstrates Radical Cylon's versatility across a range of distributed tasks, including the capacity to smoothly interleave client workloads, optimize deep learning execution pipelines, and reduce computational overhead. Several advanced cluster management rules, such as virtualization and multi-execution sharing, can be reimplemented to meet the particular requirements of machine learning and big data workloads in the future. These developments establish Deep RC as a strong and adaptable solution that can spur innovation in deep learning and large-scale data processing ecosystems.

Acknowledgments

We gratefully acknowledge the support from the Department of Energy and National Science Foundation through DE-SC0023452, NSF 1931512,NSF 2103986, and OAC-2411009 grants. The RADICAL team thanks Andre Merzky and Matteo Turilli for their support and development of RADICAL-Cybertools.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), https://www.tensorflow.org/, software available from tensorflow.org
2. Abeykoon, V., Perera, N., Widanage, C., Kamburugamuve, S., Kanewala, T.A., Maithree, H., Wickramasinghe, P., Uyar, A., Fox, G.: Data engineering for hpc with python. In: 2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC). pp. 13–21. IEEE (2020)
3. Addor, N., Newman, A.J., Mizukami, N., Clark, M.P.: The camels data set: catchment attributes and meteorology for large-sample studies. Hydrology and Earth System Sciences **21**(10), 5293–5313 (2017)
4. Babuji, Y., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J.M., Foster, I., et al.: Parsl: Pervasive parallel programming in python. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. pp. 25–36 (2019)
5. Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., et al.: Pathways: Asynchronous distributed dataflow for ml. Proceedings of Machine Learning and Systems **4**, 430–449 (2022)
6. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. The Bulletin of the Technical Committee on Data Engineering **38**(4) (2015)
7. Dean, J.: Introducing pathways: A next-generation ai architecture. https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/ (October 2021), (Accessed on 04/17/2024)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. p. 205–220. SOSP '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1294261.1294281, https://doi.org/10.1145/1294261.1294281
9. Developer, N.: Nvidia collective communications library (nccl)s. https://developer.nvidia.com/nccl" (April 2022), (Accessed on 08/10/2024)
10. Facebookincubator: Gloo: Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo" (March 2023), (Accessed on 04/01/2023)
11. Goel, P.: Accelerated data analytics: Speed up data exploration with rapids cudf. https://developer.nvidia.com/blog/accelerated-data-analytics-speed-up-data-exploration-with-rapids-cudf/" (March 2023), (Accessed on 10/10/2024)

12. He, J., Chen, Y.J., Idamekorala, A., Fox, G.: Science time series: Deep learning in hydrology. arXiv preprint arXiv:2410.15218 (2024)
13. McKenna, A.: The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. Genome research **20 9**, 1297–303 (2010). https://doi.org/10.1101/gr.107524.110
14. Merzky, A., Turilli, M., Titov, M., Al-Saadi, A., Jha, S.: Design and performance characterization of radical-pilot on leadership-class platforms. IEEE Transactions on Parallel and amp; Distributed Systems **33**(04), 818–829 (apr 2022). https://doi.org/10.1109/TPDS.2021.3105994
15. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., et al.: Ray: A distributed framework for emerging {AI} applications. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 561–577 (2018)
16. Olivares, K.G., Challú, C., Garza, A., Canseco, M.M., Dubrawski, A.: NeuralForecast: User friendly state-of-the-art neural forecasting models. PyCon Salt Lake City, Utah, US 2022 (2022), `https://github.com/Nixtla/neuralforecast`
17. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`
18. Perera, N., Sarker, A.K., Shan, K., Fetea, A., Kamburugamuve, S., Kanewala, T.A., Widanage, C., Staylor, M., Zhong, T., Abeykoon, V., et al.: Supercharging distributed computing environments for high-performance data engineering. Frontiers in High Performance Computing **2**, 1384619 (2024)
19. Perera, N., Sarker, A.K., Staylor, M., von Laszewski, G., Shan, K., Kamburugamuve, S., Widanage, C., Abeykoon, V., Kanewela, T.A., Fox, G.: In-depth analysis on parallel processing patterns for high-performance dataframes. Future Generation Computer Systems (2023)
20. Rivanna: University of virginia's high-performance computing (hpc) system (2019), `https://www.rc.virginia.edu/userinfo/rivanna/overview/`
21. Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th python in science conference. vol. 130, p. 136. Citeseer (2015)
22. Sakib, R., Sarker, A.K., Sourave, R.H.: Electronic device for image processing and operating method of the same (Apr 23 2024), uS Patent 11,967,048
23. Sarker, A.K., Alsaadi, A., Perera, N., Staylor, M., von Laszewski, G., Turilli, M., Kilic, O.O., Titov, M., Merzky, A., Jha, S., et al.: Radical-cylon: A heterogeneous data pipeline for scientific computing. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 84–102. Springer (2024)
24. Sarker, A.K., Lin, F.X.: Incremental perception on real time 3d data. In: Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications. pp. 68–73 (2022)
25. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., et al.: Ucx: an open source framework for hpc network apis and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. pp. 40–43. IEEE (2015)
26. Shan, K., Perera, N., Lenadora, D., Zhong, T., Sarker, A.K., Kamburugamuve, S., Kanewela, T.A., Widanage, C., Fox, G.: Hybrid cloud and hpc approach to high-performance dataframes. In: 2022 IEEE International Conference on Big Data (Big Data). pp. 2728–2736. IEEE (2022)
27. Widanage, C., Perera, N., Abeykoon, V., Kamburugamuve, S., Kanewala, T.A., Maithree, H., Wickramasinghe, P., Uyar, A., Gunduz, G., Fox, G.: High performance data engineering everywhere. In: 2020 IEEE International Conference on Smart Data Services (SMDS). pp. 122–132. IEEE (2020)
28. Yuan, J., Li, X., Cheng, C., Liu, J., Guo, R., Cai, S., Yao, C., Yang, F., Yi, X., Wu, C., et al.: Oneflow: Redesign the distributed deep learning framework from scratch. arXiv preprint arXiv:2110.15032 (2021)
29. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. Communications of the ACM **59**(11), 56–65 (2016)
30. Zhou, H., Zhang, S., Peng, J., Zhang, S., Li, J., Xiong, H., Zhang, W.: Informer: Beyond efficient transformer for long sequence time-series forecasting. In: The Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Conference. vol. 35, pp. 11106–11115. AAAI Press (2021)
31. ZMQ: High-level messaging patterns. `https://zguide.zeromq.org/docs/chapter2/#High-Level-Messaging-Patterns"` (October 2021), (Accessed on 04/05/2024)