Radical-Cylon: A Heterogeneous Data Pipeline for Scientific Computing

Arup Kumar Sarker[1,3], Aymen Alsaadi[2], Niranda Perera[5], Mills Staylor[1], Gregor von Laszewski[3], Matteo Turilli[2,4], Ozgur Ozan Kilic[4], Mikhail Titov[4], Andre Merzky[2], Shantenu Jha[2,4], and Geoffrey Fox[1,3]

[1] University of Virginia, Charlottesville, VA 22904 {djy8hg, qad5g, vxj6mb}@virginia.edu
[2] Rutgers University, 57 US Highway 1. New Brunswick, NJ 08901-8554 {aymen.alsaadi, matteo.turilli, shantenu.jha}@rutgers.edu, andremerzky@gmail.com
[3] Biocomplexity Institute and Initiative, Town Center Four, 994 Research Park Boulevard Charlottesville, VA 22911 laszewski@gmail.com
[4] Brookhaven National Laboratory, 98 Rochester St, Upton, NY 11973 {okilic, mtitov}@bnl.gov
[5] Voltron Data, Mountain View, 650 Castro St, CA niranda.perera@gmail.com

Abstract. Managing and preparing complex data for deep learning, a prevalent approach in large-scale data science can be challenging. Data transfer for model training also presents difficulties, impacting scientific fields like genomics, climate modeling, and astronomy. A large-scale solution like Google Pathways with a distributed execution environment for deep learning models exists but is proprietary. Integrating existing open-source, scalable runtime tools and data frameworks on high-performance computing (HPC) platforms is crucial to address these challenges. Our objective is to establish a smooth and unified method of combining data engineering and deep learning frameworks with diverse execution capabilities that can be deployed on various high-performance computing platforms, including cloud and supercomputers. We aim to support heterogeneous systems with accelerators, where Cylon and other data engineering and deep learning frameworks can utilize heterogeneous execution. To achieve this, we propose Radical-Cylon, a heterogeneous runtime system with a parallel and distributed data framework to execute Cylon as a task of Radical Pilot. We thoroughly explain Radical-Cylon's design and development and the execution process of Cylon tasks using Radical Pilot. This approach enables the use of heterogeneous `MPI-Communicators` across multiple nodes. Radical-Cylon achieves better performance than Bare-Metal Cylon with minimal and constant overhead. Radical-Cylon achieves (4∼15)% faster execution time than batch execution while performing similar join and sort operations with 35 million and 3.5 billion rows with the same resources. The approach aims to excel in both scientific and engineering research HPC systems while demonstrating robust performance on cloud infrastructures. This dual capability fosters collaboration and innovation within the open-source scientific research community.

Keywords: HPC · BSP · Cylon · ETL · MPI · UCX · RP · BM · SPMD · MPMD

1 Introduction

The exponential growth of data volume and complexity creates unprecedented challenges for the scientific community. With the increasing prevalence of sensors, internet-connected devices, and social media, the amount of data being generated is growing at an unprecedented rate. In addition, the complexity of scientific data is also increasing, with data coming from multiple sources often characterized by heterogeneity, high dimensionality, and complex relationships between variables, making it challenging to analyze data using traditional analysis tools. Even with the most advanced computing systems, processing massive datasets can be a significant bottleneck. Moreover, the sheer volume ofx data can make storing and transferring data across systems challenging.

These challenges have far-reaching implications across various scientific domains, including but not limited to genomics, climate modeling, physics simulations, and neuroscience. In genomics, for example, the amount of data generated from a single genome sequencing has grown exponentially, with individual genome sequencing now generating over 200GB of data [14]. Processing and analyzing such data using traditional methods can take months or even years. Similar to genomics, a single climate modeling simulation can generate vast amounts of data, with some simulations producing up to 10PB of data. Apart from huge growth, a data science survey conducted by Anaconda, indicates that a considerable amount of developer time (45%) is dedicated to tasks like data exploration, preprocessing, and prototyping along with 33% on deep learning tasks [3].

Analyzing and extracting insights from such massive data sets can be difficult, requiring a paradigm shift in how data analysis is performed, with the need for more efficient and scalable data analysis tools. Finding scalable solutions for processing data, such as large-scale simulations, modeling, and machine learning, is crucial for domain science, as it enables researchers to extract insights and knowledge from vast amounts of data efficiently. These solutions can directly impact reducing the time to solution and cost associated with data analysis by order of magnitudes, allowing scientists to focus on understanding complex systems that are not possible with smaller datasets and conduct more comprehensive and accurate analyses, leading to more robust and reliable results.

Such solutions can be achieved by integrating existing scalable HPC runtime tools with data frameworks. Nevertheless, these solutions impose many research questions, such as performance optimization, programming models, efficient scalability, resource management, and utilization. Cylon [2] provides underlying frameworks for data engineering and deep learning applications to run on scalable HPC machines. However, it is not optimized for the efficient use of system resources and does not support a heterogeneous data pipeline. So, we first focus on a Python runtime engine that efficiently executes heterogeneous workloads of both executables/Python functions (non)MPI on a set of HPC machines. A task-based architecture, RADICAL-Pilot enable Cylon to interact and operate with different HPC platforms seamlessly, shielding Cylon from heterogeneous configurations of different HPC platforms. RADICAL-Pilot separates the resource management from the application layer, this would allow Cylon to run on any HPC resources without the need to refactor or rewrite the code, which reduces the development efforts, resulting in a loosely coupled integration.

Furthermore, Cylon requires a heterogeneous runtime environment that constructs a private MPI/UCX/GLOO communicator for every Cylon task on HPC machines, which can be delivered using RADICAL-Pilot. Adding heterogeneous capabilities with RADICAL-Pilot and Cylon framework, which we call Radical-Cylon, can provide a powerful solution to these challenges as it enables the development of a unified system that can handle both compute and data-intensive workloads in an efficient and scalable manner. The experimental outcomes demonstrate that Radical-Cylon performs comparable to Bare-Metal (BM)-Cylon in strong and weak scaling scenarios involving join and sort operations. Radical-Cylon outperforms BM-Cylon in certain instances, particularly when the parallelism level reaches 512 or higher. For heterogeneous execution with strong and weak scaling of multiple tasks (e.g. combination of sort and join), Radical-Cylon showcases a performance improvement of (4∼15)% faster compared to batch execution with the same amount of resource utilization across all configurations with datasets of 35 million and 3.5 billion rows.

2 Related Works

Recently, researchers in the field of distributed computing systems have been exploring various ways to improve the runtime of machine learning models. One such approach is the Ray framework [17], which proposes a new way of thinking about distributed computing for future AI applications. Although Ray is primarily intended for reinforcement learning scenarios, it has not yet been adopted for large-scale reinforcement learning systems due to some unresolved issues. However, Ray has had a significant impact and is considered to be positioned between K8S and deep learning frameworks, although it cannot replace them in these areas. Therefore, there is a need for an end-to-end framework that can optimize performance at every layer of the system, as the multiple components in each layer are tightly coupled, and the performance of each distributed model operation is affected if a single node or communication is not optimized.

The question of why we need a uniform distributed architecture arises, along with the bottlenecks of the current system. The answer is that we need an optimized system to reduce latency, but we also need to prove that the architecture is incremental and adaptive. To address this, Jeff Dean proposed a new concept of program execution patterns in his blog "Pathways: Next Generation AI Architectures." [9]. The framework is straightforward when considering Single Program Multiple Data (SPMD) criteria, but Multiple Program Multiple Data (MPMD) is composed of multiple SPMDs. Pathways [6] departs from the traditional deep learning framework and design at a higher level to consider the best architecture for MPMD which aggregates all state-of-the-art Big Data frameworks. Hadoop [4] was the first generation of Big Data analytics and introduced the MapReduce programming model [10]. However, Apache Spark [27] and Apache Flink [7] surpassed Hadoop by providing faster and more user-friendly APIs. These advancements were made possible by hardware improvements that allowed for in-memory Big Data processing. Python Pandas Dataframes [15] have emerged as the preferred data analytics tool among the data science and engineering community, despite being limited in performance and scalability. Dask Distributed and Modin are built on top of Pandas, providing distributed and generalized DataFrame abstractions, respectively. Later, CuDF emerged as a DataFrame abstraction that can be used for ETL pipelines on top of GPU hardware.

OneFlow [26] is a pioneering effort to revolutionize distributed data processing specifically within the realm of deep learning. The authors of OneFlow assert that their framework has the potential to replace Plaque, a component within the Pathways system. ZeroMQ [28] is a high-performance asynchronous messaging library that provides communication between different applications or parts of an application over transport protocols. It can be a potential alternative to MPI. Arkouda [13] supports ZeroMQ-based communication protocol but has limitations in supporting heterogeneous data pipelines. Parsl [5] is a parallel scripting library designed to enhance Python with straightforward, scalable, and adaptable elements for representing parallelism. RP employs specifically to create a dynamic dependency graph of components.

3 Design and Implementation

A unique integration approach of Cylon with the RADICAL-Pilot runtime system via their native Application Programming Interface (API) is proposed on Radical-Cylon. In the core system, RADICAL-Pilot is used as the distributed runtime for managing the execution of Cylon tasks. Importantly, we consider both systems 'as they are' in a loosely coupled design without the need to implement an integration plugin while exposing both system capabilities via RADICAL-Pilot API. This approach allows Cylon to benefit from RADICAL-Pilot heterogeneous runtime capabilities, specifically the capabilities to construct and deliver MPI communicators without modifying Cylon tasks. Additionally, the proposed design offers a flexible and adaptable framework for developing and deploying data-intensive applications on various HPC platforms.

3.1 Radical-Pilot (RP)

RADICAL-Pilot is a flexible and scalable runtime system designed to support the execution of large-scale applications on HPC leadership-class platforms. RADICAL-Pilot enables the execution of concurrent and heterogeneous workloads on various HPC resources. Further, RADICAL-Pilot offers the capabilities to manage the execution of (non)MPI single/multi-thread/core/node executables and functions efficiently.

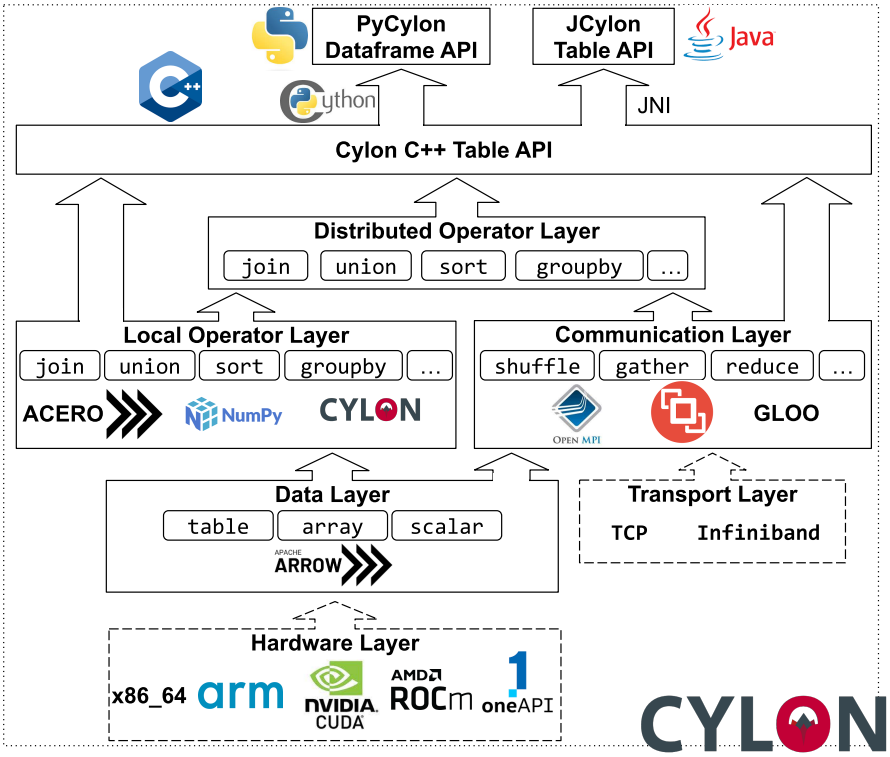

Fig. 1. Cylon Layered Architecture. From the bottom-up view, the Hardware layer is compatible with vendor-based or open-sourced transport layer [20]

RADICAL-Pilot consists of three main components: The *PilotManager*, *TaskManager*, and *RemoteAgent*. Both *PilotManager* and *TaskManager* run on user resources such as local computer or login/compute node of a cluster or HPC platforms, while the *RemoteAgent* resides on the compute resources [16].

The *PilotManager* is responsible for managing the lifecycle of the pilot, which is a placeholder for resources on HPC systems. The *TaskManager* is responsible for managing the lifecycle of tasks, which represent an application such as a function or executable that runs on the pilot's available resources. The *RemoteAgent* is responsible for preparing the execution environment and starting the pilot to execute the tasks on the remote resources.

RADICAL-Pilot enables efficient scheduling, placement, and launching of independent tasks across multiple compute nodes. Leveraging the pilot abstraction model, RADICAL-Pilot has demonstrated the ability to concurrently execute up to one million tasks across one thousand nodes with minimal overheads [16].

3.2 Cylon

Cylon represents a profound evolution in the realm of data engineering, offering a comprehensive toolkit that seamlessly connects AI/ML(with PyTorch [19] and TensorFlow [1]) systems with data processing pipelines [25]. Cylon's overarching vision is rooted in the fusion of data engineering and AI/ML, as exemplified by its ability to effortlessly interact with a spectrum of data structures and systems and optimize ETL performance.

At the heart of Cylon's architecture, there is a core framework, wielding a sophisticated table abstraction to represent structured data in Fig. 1. This abstraction empowers individual ranks or processes to collectively handle partitions of data, fostering a sense of unity and collaboration despite distributed computing challenges. Cylon's arsenal of "local operators" execute operations solely on locally accessible data, while "distributed operators" harness network capabilities to execute complex tasks that necessitate inter-process communication.

To mitigate the complexities of distributed programming, Cylon orchestrates network-level operations that transpire atop communication protocols (Fig. 2) like TCP or Infiniband. This allows multiple communication abstraction frameworks, e.g., MPI, UCX [23], and GLOO [12] for heterogenous data transmission [21, 24]. This strategic approach of channel abstraction elevates the efficiency of Cylon's operations(e.g. shuffle, gather, reduce, etc.), enabling seamless communication between processes while harmonizing performance across diverse hardware environments.

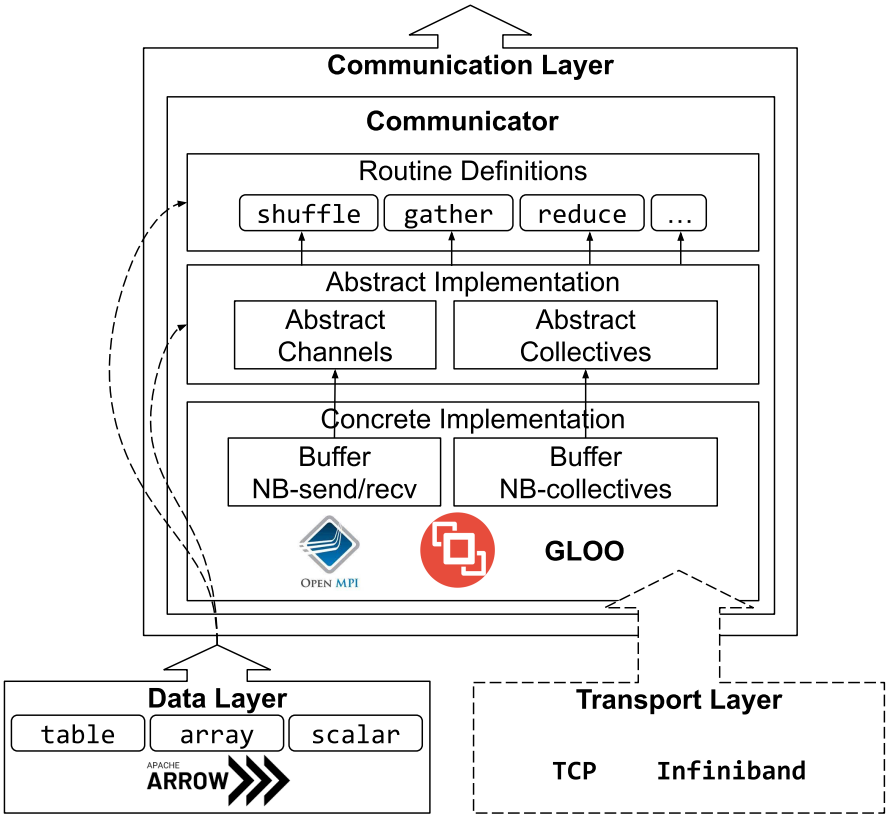

Fig. 2. Cylon Communicator Model. It has cross-platform support of Open-MPI, GLOO and UCX [20]

With Apache Arrow's Columnar Format as its foundation, Cylon's data model aligns seamlessly with a myriad of open-source frameworks and libraries. This interoperability ensures a harmonious coexistence within the larger data ecosystem, facilitating the exchange of data and insights between different platforms. In this paper, we embark on an exploration of Cylon's rich tapestry, dissecting its core components and intricate layers as an abstraction of RADICAL-Pilot. From data models and operators to communication and transport layers, Cylon's architecture emerges as a core component of distributed high-performance frameworks to reshape the future of data engineering and AI/ML integration.

3.3 Design

Cylon and RADICAL-Pilot are two isolated systems offering different functionalities and capabilities. The integration design advocates the loosely coupled approach where both systems work independently of each other, with minimal dependencies and interactions, while benefiting from each other's capabilities.

The integrated design of RADICAL-Pilot and Cylon is shown in Fig. 3 where Cylon is plugged as a top-level component to send different types of Cylon tasks (functions or executables) to RADICAL-Pilot to execute on HPC resources. The main communication point between both systems is their native APIs, as both systems offer flexible and simple Python-based interfaces.

Cylon and RADICAL-Pilot loosely coupled integration can be easily scaled out, expanded, or contracted to meet changing demands [11]. Flexibility-wise, both systems are developing rapidly and might introduce new fundamental changes in the design or implementation, such as adding or removing new system components. Further, any changes in both systems do not necessarily require changes to the other system's components and would not affect the existing integration as there are no direct dependencies between the integrated systems. From a fault tolerance perspective, the integration approach of Cylon and RADICAL-Pilot is more resilient, as failures in one system or component do not affect the entire system. Failure of any component can be isolated and contained, allowing the rest of the system to continue receiving and executing tasks.

3.4 Implementation

We implemented Radical-Cylon as a single system enabling communication between the two systems via their Python APIs, as shown in Fig. 3. The implementations expose RADICAL-Pilot API as a main interface to specify, interact, and execute Cylon tasks on multiple HPC platforms. Further, each Cylon task is represented as a `RadicalPilot.TaskDescription` class with their resource requirements, such as the number of CPUs, GPUs, and memory.

Once the Radical-Cylon starts (Fig.3-1), RADICAL-Pilot instructs the `PilotManager` to create the `Pilot` object with the required number of resources (Fig.3-2). Further, RADICAL-Pilot creates the `TaskManager` and submits Cylon tasks to the `TaskManager` to be executed on the remote resources (Fig.3-3). Synchronously, once the pilot resources are acquired from the HPC resource manager, RADICAL-Pilot starts the `RemoteAgent` on the acquired resources (Fig.3-4). Once the `RemoteAgent` is bootstrapped and ready, it starts the RAPTOR subsystem (Fig.3-5), which is an abstraction of the Master-Worker MPI paradigm. RAPTOR implementation is based on `mpi4py` [8] and can concurrently execute heterogeneous MPI/non-MPI functions across multiple nodes. Unlike other pilot systems, RADICAL-Pilot and via RAPTOR offer the capabilities of constructing private MPI communicators of different sizes during the runtime, which Cylon tasks require.

Once all RAPTOR master(s) and worker(s) start, the master(s) receives the Cylon tasks from the `RemoteAgent` scheduler and distributes them across the workers to be executed. When the worker receives Cylon tasks, it isolates a set of `MPI-Ranks` based on the resource requirements of the Cylon task and groups them to construct a private `MPI-Communincator` and deliver it to the task during runtime (Fig.3-6). Finally, once all of Cylon's tasks finish execution, the master collects the results of the tasks and sends them back to the `TaskManager`.

A bird's-eye view of the Radical-Cylon system is shown in Fig. 4, with in-depth components and data flow. In the initial step (Step 1), when a user intends to execute a traced program (MPMD) comprising multiple computations (SPMD), they employ the Radical-Cylon system by invoking the RP-Client. Moving to Step 2, the Pilot Manager assigns virtual devices for computations not previously executed and registers these computations with the Resource Manager.

Subsequently, in Step 3, the client activates the background server to execute instructions for the pilot manager, incorporating considerations for network connections between devices and data routing operations among various computations. If the virtual device for a program remains unchanged, the generated representation can be swiftly reused; however, if the Resource Manager alters a program's virtual device, recompilation is necessary. These three steps collectively form the front end of Radical-Cylon. Remote Agent creates multiple execution pipelines with two persistent daemons – a scheduler and an executor – capable of communicating (Steps 5, 6, 7) to achieve distributed coordination, constituting the control plane communication.

The executor invokes Cylon data engineering frameworks (Step 8) to perform local sorting or joining, or data plane communication as indicated in Step 9 (primarily cluster communication involving shuffle or gather operations). Notably, the communication between the data plane employs the same communication framework, with the former depicted by a blue arrow indicating higher bandwidth, and the latter represented by a green arrow indicating lower bandwidth.

4 Experiments

Table 1 shows the setup of our experiments. We use UVA Rivanna HPC [22] and ORNL-Summit [18] to set up weak and strong scalability experiments. On Rivanna, we use the `parallel` queue with 37 cores per node and a maximum of 14 nodes, and on ORNL-Summit, we use a maximum of 64 nodes with 42 cores per node. We evaluate the efficiency of Radical-Cylon and compare it to Bare-Metal Cylon (BM-Cylon) while executing Cylon join and sort operations with single pipeline execution. We measure two metrics: Total Execution Time and Radical-Cylon overheads. The Total Execution Time represents the total time Radical-Cylon spent executing

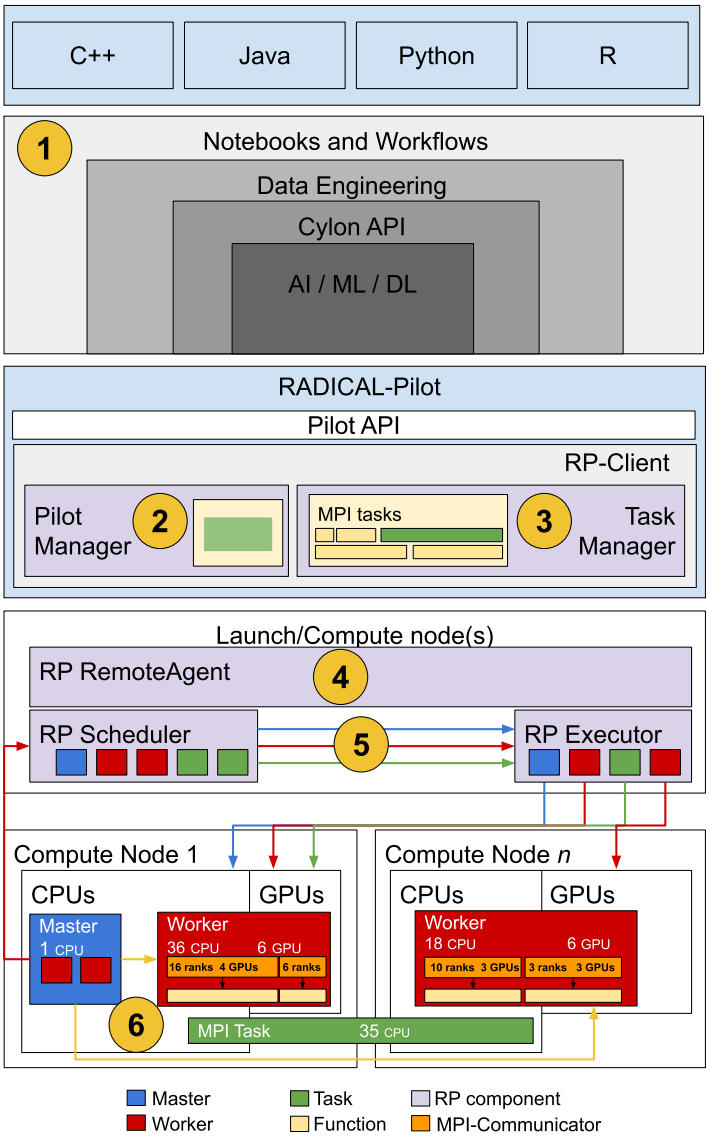

Fig. 3. Radical-Cylon Architecture. A modular design with dependent components. Segregated independent module with top-down flow from cross-platform to hardware resources.

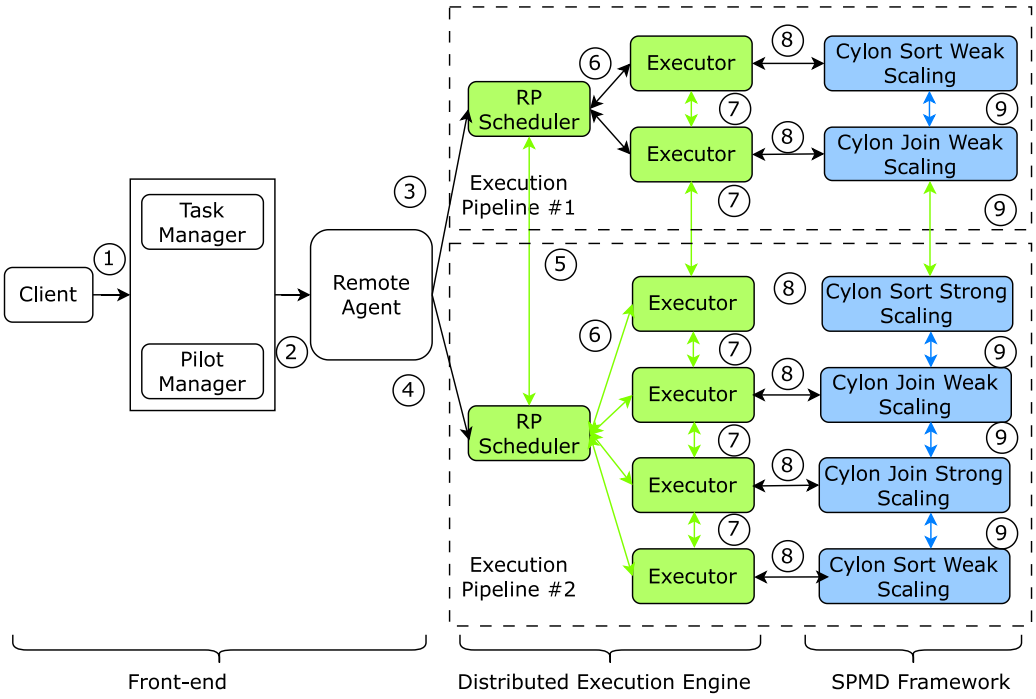

Fig. 4. Heterogeneous Execution with Control and Data Flow. The execution pipeline uses a separate SPMD framework for underlying tasks.

the join and sort tasks on the computing resources with N ranks. The Overheads represent the time taken by Radical-Cylon (mainly RP) to (i) deserialize the task object and (ii) construct the `MPI-Communicator` with N ranks and deliver it to the tasks. Each join and sort task takes N ranks with a maximum of 35 million rows per rank for weak scaling, and 3.5 billion rows are divided into N ranks for strong scaling. Collectively, experiments 4.1, 4.2 and 4.3 allow us to study and evaluate the scalability performance of Radical-Cylon while comparing it to BM-Cylon on Rivanna and batch execution of BM-Cylon on ORNL-Summit with the setup of multiple configurations.

Table 1. Experiments Setup on UVA.Rivanna and ORNL.summit. WS/SS = weak/strong scaling; M=Million; B=Billion; rank=1 physical core; RN=Rivanna Nodes; SN=Summit Nodes; RC=Rivanna CPU; SC=Summit CPU

ID	Experiment Type	RN	SN	Rows Size	RCs(ranks)	SCs(ranks)
A	Join Operation	WS/SS $4-14$	$2-64$	[35M | 3.5B]	#nodes × 37	#nodes × 42
B	Sort Operation	WS/SS $4-14$	$2-64$	[35M | 3.5B]	#nodes × 37	#nodes × 42
C	Heterogeneous	WS/SS	$2-64$	[35M | 3.5B]		#nodes × 42

4.1 Join Operation Scalability

The join weak scaling experiment is depicted in Fig.-5 (right), 6 (right). Across all tests are performed 10 times with multiple parallelisms (a single rank is used for each parallel execution), and the total execution time ranges from 215 to 250 seconds for both bare-metal and RADICAL-Pilot Cylon executions on Rivanna. We got an overlapping error bar by increasing the number of workers in ***ORNL-Summit*** with different configurations because of constant RP overheads (Fig. 6 (right)).

As the number of ranks increases, a minor addition of execution time for allgather from all ranks becomes evident in the performance in the join weak scaling experiment. It's noteworthy that Radical-Cylon exhibits better performance with a lower number of ranks, particularly when it's below 200. Starting from 222 ranks and onwards, in join weak scaling, an average of 10 seconds of increasing is observed, which is deemed acceptable considering the benefit of achieving heterogeneity among multiple nodes in the HPC system. However, the error bar graph in both Cylon overlaps indicates we achieved similar execution times in a single pipeline. Similar trends can be observed in the join strong scaling operation. For strong scaling, where 3.5 billion rows are distributed

Table 2. Radical-Cylon((RP-Cylon)) Execution Time and Overheads of Strong and Weak Scaling from Experiment A (Join Operations) and B (Sort Operation) on Rivanna.

Operation	Scaling	Parallelism	Execution Time time (seconds)	Overheads (tasks/second)
Join	Weak	148	215.64 ±4.35	2.9 ±0.1
		222	226.12 ±2.59	2.3 ±0.4
		296	237.01 ±2.96	2.8 ±0.8
		370	239.87 ±3.41	2.5 ±0.8
		444	241.59 ±2.76	2.9 ±0.4
		518	253.66 ±1.53	3.2 ±0.6
	Strong	148	144.80 ±3.21	2.79 ±0.05
		222	98.03 ±3.32	2.51 ±0.2
		296	78.14 ±3.02	2.45 ±0.1
		370	61.80 ±3.35	2.81 ±0.3
		444	52.72 ±2.32	3 ±0.8
		518	47.10 ±3.54	3.5 ±0.8
Sort	Weak	148	192.74 ±3.21	3.87 ±0.9
		222	204.44 ±3.32	3.4 ±1.2
		296	207.20 ±4.02	3.85 ±0.9
		370	212.81 ±3.35	2.59 ±0.39
		444	215.05 ±3.32	2.61 ±0.88
		518	223.88 ±4.54	3.23 ±1.3
	Strong	148	125.53 ±2.64	2.42 ±0.8
		222	84.20 ±2.64	2.37 ±0.61
		296	63.76 ±2.80	2.42 ±0.5
		370	51.31 ±3.18	2.65 ±0.92
		444	44.46 ±0.96	2.91 ±0.8
		518	39.52 ±3.98	3.5 ±1.05

among all ranks, the same 10 iterations are employed. The results, depicted in Fig.-5 (left), 6 (left) demonstrate a significant reduction in execution time as the number of ranks increases for both BM-Cylon and Radical-Cylon implementations on Rivanna and ORNL-Summit.

With the increasing rank count, Radical-Cylon gradually closes the latency gap with BM-Cylon, showing only a marginal difference in latencies of total execution time. The error bar shows an identical performance with both experiments set up. This leveling of latencies can be attributed to the efficient scheduling and task distribution mechanisms employed by Radical-Cylon. This efficiency arises from the fixed allocation of rows among ranks for join operations and the utilization of a consistent table index for merging in distributed join operations. Consequently, the communication and aggregation overheads are constant in Radical-Cylon (in Table 2).

4.2 Sort Operation Scalability

The identical scaling configurations are applied to sorting operations, and a single rank is used in each parallel execution on Rivanna and ORNL-Summit (Fig.-7 (right), 8 (right)). In Fig.-7 (right), for Rivanna, an average latency discrepancy of around 15 seconds is observed between the minimum rank count (148) and the maximum rank count (518) in the weak scaling experiment. This latency increase with higher rank numbers is anticipated, as it influences the data shuffling and merging stages within the distributed sorting process, thereby introducing additional overhead. Effective utilization of resources for communication and data partition is pivotal in influencing execution time. Remarkably, as the rank count increases, Radical-Cylon demonstrates enhanced performance and consistently narrows the gap with BM-Cylon. But with multiple iterations, we are getting an overlapping error bar that indicates similar performance with both metrics.

Partitioning a massive dataset across numerous nodes leads to a reduction in execution time. In the strong scaling sort operation, showcased in Fig.-7 (left) and 8 (left), a tabular dataset containing 3.5 billion rows is partitioned among hundreds of ranks across various test runs. Each test run encompasses 10 iterations, and the execution time is utilized for graph plotting. The results unequivocally highlight that with 148 ranks, the total execution time amounts to 125 seconds, which diminishes to a mere 39.5 seconds with 518 ranks on the Rivanna cluster. Due to constant overheads in both scaling of the sort operation, the same behavior is observed in the ORNL-Summit.

Both RADICAL-Pilot and BM-Cylon approaches achieve closely comparable performance, differing by mere milliseconds in their total execution times, although, with multiple iterations, we are getting an overlapping

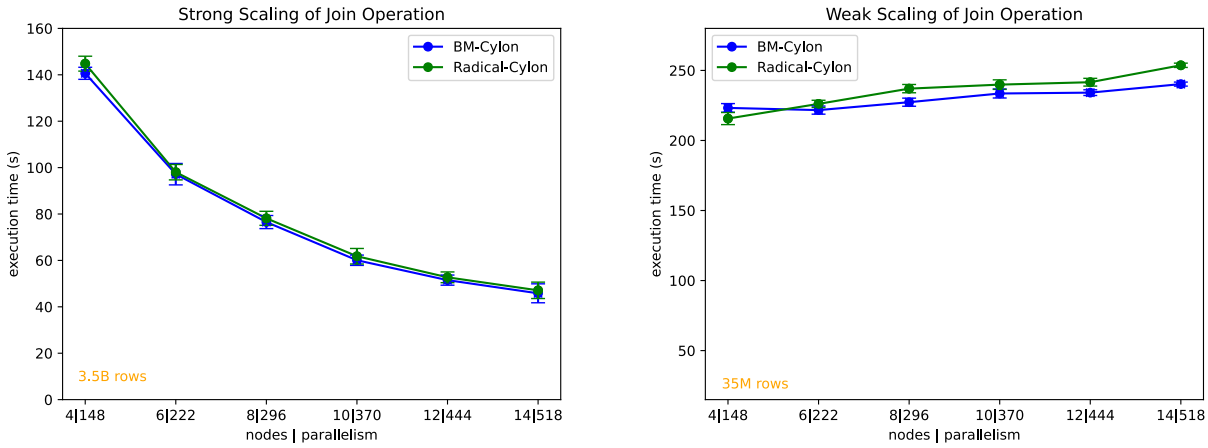

Fig. 5. Comparison of strong scaling(left) and weak scaling(right) performance of Bare-Metal and Radical-Cylon with join operation on ***Rivanna***. `execution time(s)` is calculated by running task for 10 iterations. The number of parallelism is calculated by nodes multiple by 37 cores per node

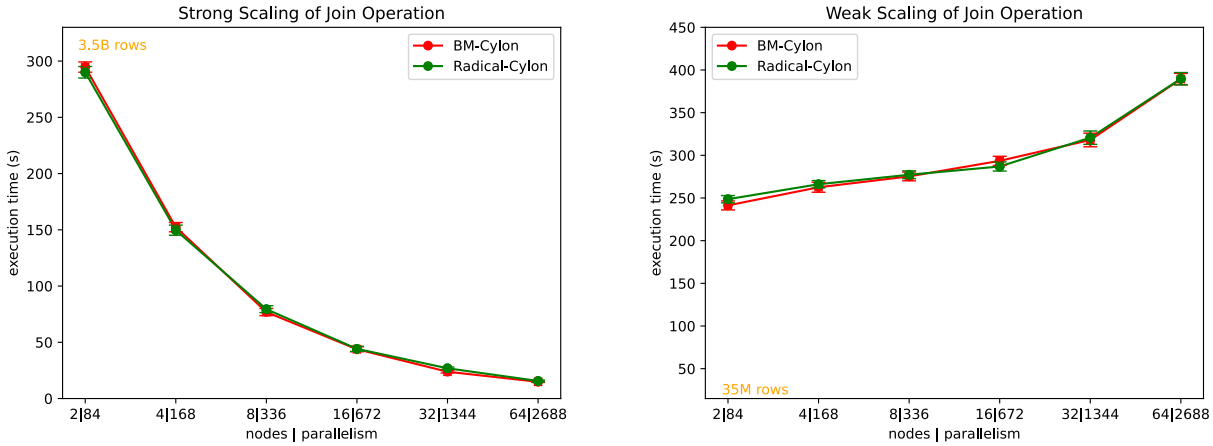

Fig. 6. Comparison of strong scaling(left) and weak scaling(right) performance of Bare-Metal and Radical-Cylon with join operation on ***ORNL-Summit***. `execution time(s)` is calculated by running the task for 10 iterations and it's used for a higher scalability test. The number of parallelism is calculated by nodes multiple by 42 cores per node.

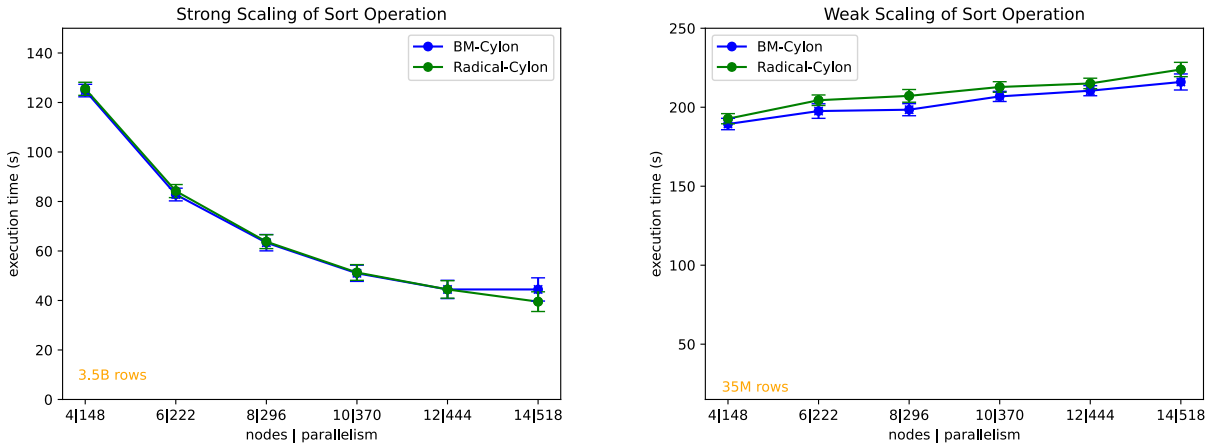

Fig. 7. Comparison of strong scaling(left) and weak scaling(right) performance of Bare-Metal and Radical-Cylon with sort operation on ***Rivanna***. `execution time(s)` is calculated by running task for 10 iterations. The number of parallelism is calculated by nodes multiple by 37 cores per node

error bar. However, distributed execution introduces a set of challenges encompassing the management of data distribution, navigation of communication overhead between nodes, and mitigation of potential node failures. These complexities are magnified with an increased number of nodes. That might happen in both BM and Radical Cylon. Apart from the comparable performance, we see a constant overhead when using Radical-Cylon in strong and weak scaling operations (in Table 2) despite of increasing parallelism.

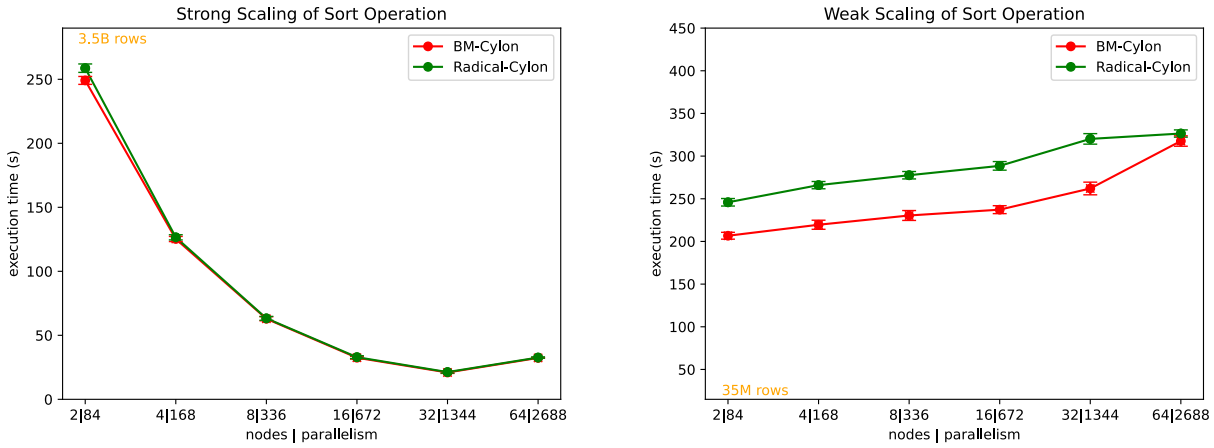

Fig. 8. Comparison of strong scaling(left) and weak scaling(right) performance of Bare-Metal and Radical-Cylon with sort operation on ***ORNL-Summit***. The number of parallelism is calculated by nodes multiple by 42 cores per node. Strong scaling with 2688 nodes takes a bit more time than with 1344 due to the lack of rows available for each worker and some workers go idle.

4.3 Benchmarking with multiple data pipeline

The heterogeneous data pipeline is used on the ORNL-Summit clusters, involving multiple scaling benchmarks. Four distinct scaling operations, namely Sort and Join weak scaling (WS), are configured with 35 million rows in each worker, while strong scaling is executed with 3.5 billion rows. Six different experiments are conducted with CPU counts ranging from 84 to 2688. Each experiment is iterated 10 times in a single run. We gauge the total execution time (in seconds) against the number of nodes or parallelism and illustrate the results in Fig.-9.

In the case of weak scaling for the sort and join operations, there is a gradual increase in execution time to compile results for generating a global table. as the number of CPUs rises. Similarly, with strong scaling operations, where 3.5 billion rows are distributed among multiple workers, performance improves as the number of workers increases, resulting in a significant reduction in execution time. This experiment validates the achievement of a scalable model using the proposed task-based execution framework.

However, the core premise of our argument faces a potential challenge if the proposed design cannot surpass the performance of Batch execution while ensuring minimal resource utilization. In the Batch execution model, join and sort operations are configured through an LSF-based script on the ORNL-Summit cluster, running in parallel. Each operation lacks control over the hardware resources of the other operation, even if some workers finish their tasks, introducing a potential inefficiency in resource usage.

In the context of the heterogeneous scaling operation, the join and sort processes are treated as distinct tasks within a single execution. Consequently, when any worker completes their task, the released resources become available to others. For weak scaling join and sort operations (depicted in Fig.-10 (right)), 84 CPUs are efficiently allocated, and resource release is effectively managed, enabling both tasks to conclude in 417.33 seconds. In contrast, under the batch execution model, the same amount of CPUs are allocated separately for the join and sort processes, consuming a total of 488.33 seconds to execute both tasks, despite resource allocation considerations.

The same efficiency is observed in strong scaling join and sort operations for both heterogeneous and batch execution, as illustrated in Fig.-10 (left). Radical-Cylon achieves comparable or improved execution times while utilizing the same resources for the two tasks, thanks to additional optimizations in separate resource utilization and constant RP overheads. To provide a comprehensive overview of the performance evaluation between heterogeneous and batch execution, we have plotted a bar graph (Fig-11) depicting performance improvement with multiple configurations. Radical-Cylon consistently outperforms batch execution by 4-15% in various configurations of scaling operations. This underscores the effectiveness of Radical-Cylon in achieving superior performance with optimized resource utilization.

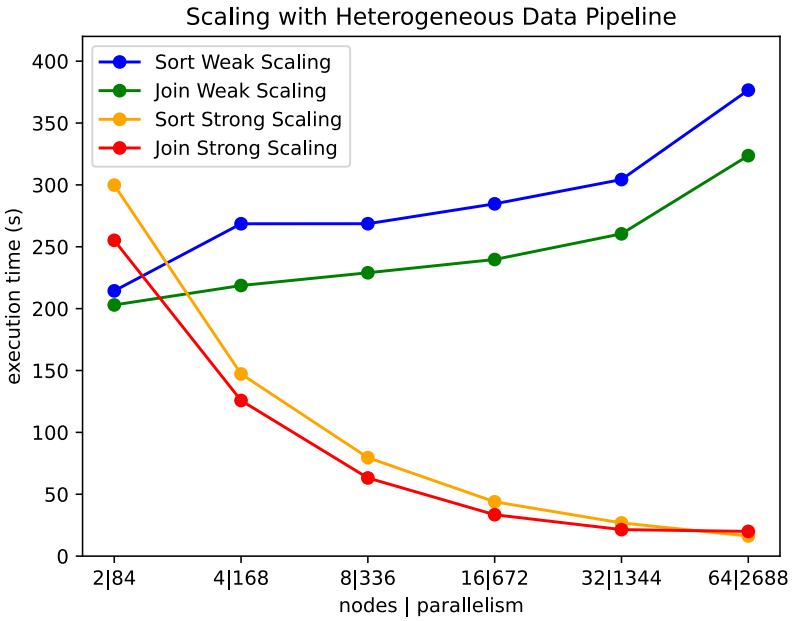

Fig. 9. Heterogeneous Executions with sort and join strong and weak scaling(4) operations on ***ORNL-Summit***. Strong scaling with 2688 nodes takes a bit more time than with 1344 due to the lack of rows available for each worker and some workers go idle.

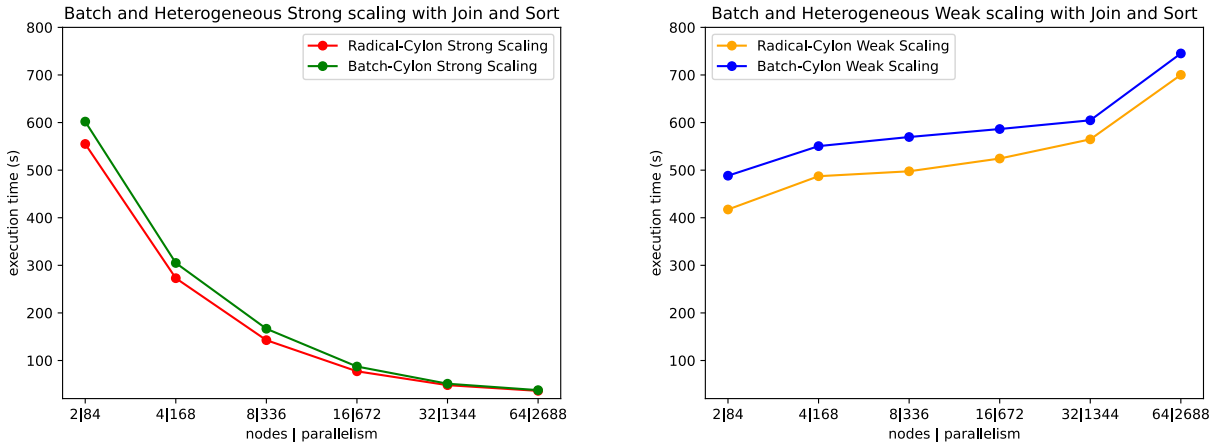

Fig. 10. Comparison of strong scaling(left) and weak scaling(right) performance of Heterogeneous and Batch executions on ***ORNL-Summit***. `execution time(s)` is calculated by running task for 10 iterations. Batch execution time for join and sort is calculated separately from two batch outputs

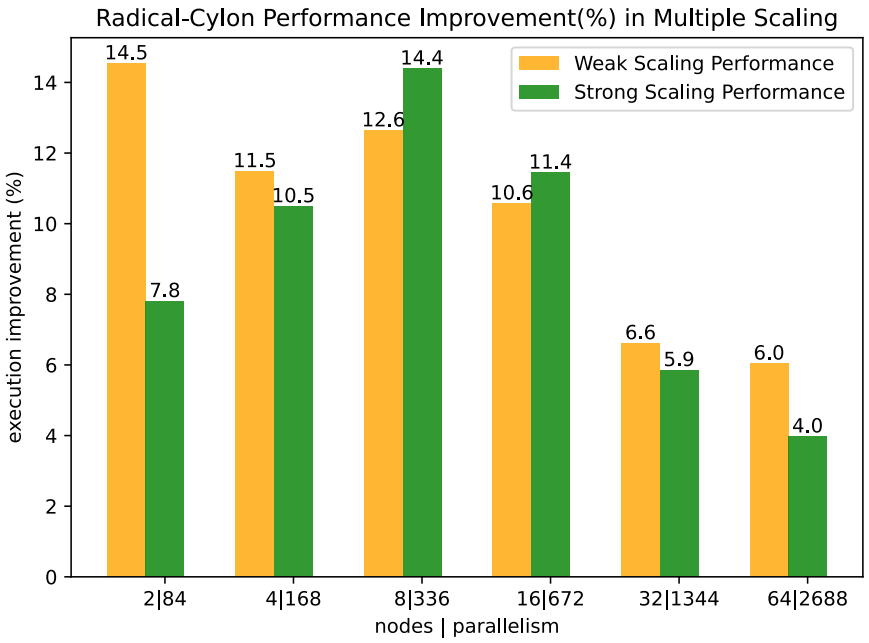

Fig. 11. Radical-Cylon performance improvement with scaling operation on ***ORNL-Summit***. The number of nodes for batch execution is generated by the requested resource as the system allocates resources separately

4.4 Discussions

The results in 4.1, 4.2 show that Radical-Cylon scales efficiently with very small overheads and achieves similar performance in single-task execution. It also outperforms batch processing of BM-Cylon in 4.3 with heterogeneous execution. Radical-Cylon overheads presented in Table 2 show that Radical-Cylon takes an average of 3.4 seconds to construct an `MPI-Communincator` with 518 ranks which are marginal compared to the total execution time and the size of the experiments. It shows impeccable scalability, particularly when the number of CPUs is less than or equal to 2688 on ORNL-Summit. The Raptor module, responsible for resource allocation and scheduling, unfortunately, encountered challenges in allocating resources for Cylon. We are working consistently with OLCF support team to fix the issue.

Cylon performance is measured with a data frame execution runtime. A collection of data frame operators can be arranged in a directed acyclic graph (DAG). Execution of this DAG can further be improved by identifying independent branches of execution and executing such independent tasks parallelly. Additionally, each of these tasks themselves is Bulk Synchronize Parallel (BSP). Radical-Cylon allows us to control the parallelism of these BSP tasks. In the future, Cylon is planning to add an optimizer based on the data frame DAG. One aspect is traditional query optimization, similar to SQL query optimization, which is orthogonal to the scheduling mechanism. Another important metric would be scheduling overhead distribution of the underlying scheduling environment. This forward-looking initiative aims to enhance the efficiency and performance of data processing within the context of machine/deep learning tasks.

Radical-Cylon has been meticulously crafted to address a unified execution setting encompassing both CPUs and GPUs, catering to the needs of multiple data pipelines. While integrating CPUs and GPUs within a single task does introduce computational intricacies owing to the foundational structure of Cylon, the concept of heterogeneous execution remains viable. This involves employing distinct groups of ranks equipped with specialized memory allocated either on CPUs or GPUs, enabling the concurrent utilization of these processing units. However, we limit our experiments to CPU clusters only due to dependencies on CUDA-aware MPI along with supported frameworks of ORNL-Summit and Rivanna cluster, which is in the process of being addressed.

The design of RADICAL-Pilot aims to support an extensive spectrum of meticulous resource management policies. Our initial focus has revolved around establishing multiple data pipelines and executing distributed operations in the form of functions. Here Cylon plays an important part by providing distributed execution. Looking ahead to more complex multi-tenancy scenarios, RADICAL-Pilot must proficiently manage a diverse range of resource types, including not only device and host memory but also network bandwidth. The Master-Worker raptor model employed by RADICAL-Pilot provides the system with a robust capability to monitor available resources and allocate them on a large scale. Our future plans involve exploring common multi-tenancy requirements such as prioritization, performance segregation, and resource tracking. Importantly, the timeframe

for these endeavors is considerably shorter than prior efforts yet will encompass significantly larger pools of resources that will help to implement a seamless ML/DL pipeline.

5 Conclusions

RP achieves parity with the cutting-edge multi-execution design in today's data engineering execution landscape, which predominantly employs an SPMD approach. This compatibility extends to multi-execution setups using SLURM-SRUN on top of Cylon, as evidenced in our evaluation section. RP effectively tackles the intricacies of resource management and the execution of diverse data pipelines. Notably, RADICAL-Pilot attains performance levels comparable to Bare-Metal Cylon across various distributed operations.

Concurrently, RP revolutionizes the execution model of Cylon programs, consolidating user code under a single execution framework. This transformation introduces a centralized resource management and scheduling framework that interfaces between the client and cluster nodes. The outcome of this unified execution model is enhanced user access to more comprehensive computation patterns. Our micro-benchmarks substantiate the effective interleaving of client workloads and streamlined pipelined execution, firmly establishing the efficiency and adaptability of the system with minimal overhead. Furthermore, the resource management and scheduling layer facilitates the reimplementation of cluster management policies, such as multi-execution sharing and virtualization, tailored specifically to the demands of ML and BigData workloads.

Acknowledgments

We gratefully acknowledge the support from the Department of Energy and National Science Foundation through DE-SC0023452, NSF 1931512, and NSF 2103986 grants.

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. arXiv preprint arXiv:2010.06312 (2020)
3. Anaconda: The state of data science 2020 moving from hype toward maturity. `https://www.anaconda.com/resources/whitepapers/state-of-data-science-2020"` (December 2020), (Accessed on 04/05/2023)
4. Apache: Apache hadoop. `https://hadoop.apache.org/` (May 2022), (Accessed on 04/18/2023)
5. 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)
6. 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)
7. 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)
8. Dalcin, L., Paz, R., Storti, M.: Mpi for python. Journal of Parallel and Distributed Computing **65**(9), 1108–1115 (Sep 2005). https://doi.org/10.1016/j.jpdc.2005.03.010
9. Dean, J.: Introducing pathways: A next-generation ai architecture. `https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/` (October 2021), (Accessed on 04/17/2023)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
11. 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`
12. Facebookincubator: Gloo: Collective communications library with various primitives for multi-machine training. `https://github.com/facebookincubator/gloo"` (March 2023), (Accessed on 04/01/2023)
13. Government, U.: Arkouda: Numpy-like arrays at massive scale backed by chapel. `https://pypi.org/project/arkouda/#description"` (March 2019), (Accessed on 04/05/2023)
14. 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
15. McKinney, W., et al.: pandas: a foundational python library for data analysis and statistics. Python for high performance and scientific computing **14**(9), 1–9 (2011)

16. 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
17. 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)
18. ORNL-Summit: Summit (2019), `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`
19. 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`
20. 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)
21. Perera, N., Shan, K., Kamburugamuwe, S., Kanewela, T.A., Widanage, C., Sarker, A., Staylor, M., Zhong, T., Abeykoon, V., Fox, G.: Supercharging distributed computing environments for high performance data engineering. arXiv preprint arXiv:2301.07896 (2023)
22. Rivanna: University of virginia's high-performance computing (hpc) system (2019), `https://www.rc.virginia.edu/userinfo/rivanna/overview/`
23. 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)
24. 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)
25. 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)
26. 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)
27. 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)
28. ZMQ: High-level messaging patterns. `https://zguide.zeromq.org/docs/chapter2/#High-Level-Messaging-Patterns"` (October 2021), (Accessed on 04/05/2023)