Serverless High-Performance Data Engineering Using Python

Qualification Exam Proposal

Mills Wellons Staylor, III
qad5gv@virginia.edu
University of Virginia
USA

Abstract

Data can be found everywhere, from health to human infrastructure to the surge of sensors to the proliferation of internet-linked devices. To meet this challenge, the data engineering domain has expanded monumentally in recent years in both research and industry. Additionally, In recent years, the data engineering discipline has been dramatically impacted by Artificial Intelligence (AI) and Machine Learning (ML), which has resulted in research on the speed, performance, and optimization of such processes. Traditionally, data engineering, Machine Learning, and Artificial Intelligence workloads have been executed on large clusters in a data center environment. This requires considerable investment in terms of both hardware and maintenance. With the advent of the public cloud, it is now possible to run large applications across nodes without maintaining or owning hardware. Serverless computing has emerged in cloud and open-source varieties to meet such needs. This allows users of such systems to focus on the application code and take advantage of bundled CPUs and memory configuration without focusing primarily on the semantics of horizontal scalability and resource allocation.

Serverless functions like AWS Lambda offer horizontal scaling and fine-grained billing without requiring traditional cloud infrastructure management. However, when executing jobs or tasks on large datasets, users frequently depend on external storage options significantly slower than direct communication used by HPC clusters. The FMI library has been developed to address this limitation, which employs NAT traversal to enable direct communication between pairs of Lambda functions. It includes support for distributed collective operations such as direct, reduce, allreduce, scan and other collective operations using point-to-point communication. We describe Cylon as a high-performance distributed data frame solution that has shown promising results for data processing with Python. We present distributed join scaling outcomes deployed in Docker containers on AWS ECS and compare these results to Singularity on UVA's Rivanna supercomputer using Cylon with high-performance communication and collectives provided by UCX and UCC, respectively. We propose integrating the FMI library into Cylon as a communicator to address the communication and performance challenges associated with serverless functions. We aim to demonstrate comparable performance to HPCs.

CCS Concepts

• **Networks → Cloud Computing, Serverless Computing**; • **FMI → Fast Messaging Interface**; • **UCC → Unified Collective Communication**; • **UCX → Unified Communication X**; • **BSP → Bulk Synchronize Parallel**.

1 Introduction and Motivation

The data engineering domain has expanded monumentally over the past decade, thanks to the emergence of Machine Learning (ML) and Artificial Intelligence (AI). Data is no longer categorized as Gigabytes (GB) or Megabytes (MB), files, or databases but as terabytes or abstract data stores. It takes a considerable amount of developer time for pre-processing when a better use of time would be designing and implementing deep learning or machine learning models. The increasing amount of connected internet devices and social media results in data growing at parabolic or exponential rates. Therefore, improving performance is critical to continuing to build pipelines that support the development of optimized AI/ML training and inference.

The exponential growth in data volume and complexity poses significant challenges across various scientific domains, particularly in genomics, climate modeling, astronomy, and neuroscience. For instance, the 1000 Genomes project and the Cancer Genome Atlas alone utilize nearly five terabases of data[15]. The TSE search terabase alone consumes a substantial 50.4 TB in size. This can be understood by considering that a single genome sequence alone can generate approximately 200 GB of data[8]. Consequently, by 2025, it would be necessary to store the genome sequences of the entire world, amounting to 40 exabytes of storage.

Decomposing complex problems represented by large datasets can be challenging and demands both scalable and efficient solutions. Machine Learning and Artificial Intelligence have revolutionized data analysis, allowing researchers to extract valuable insights that were previously unattainable. This leads to more robust and reliable results than possible with the previously available tools.

A crucial aspect of these solutions is the use of analytical engines that run on High-Performance Computing (HPC) clusters for training Large Language Models (LLMs) [2]. However, a significant challenge with this approach is that frameworks like Pandas, NumPy, and PyTorch are not directly compatible with technologies such as MPI, UCC/UCX, or Open Fabric Interfaces (OFI).

Over the past decade, Python has seen increased adoption and usage, surpassing Java usage, according to Google Trends. Figure 1 shows Java versus Pythons based on reported Google trends from 2019 to 2025.

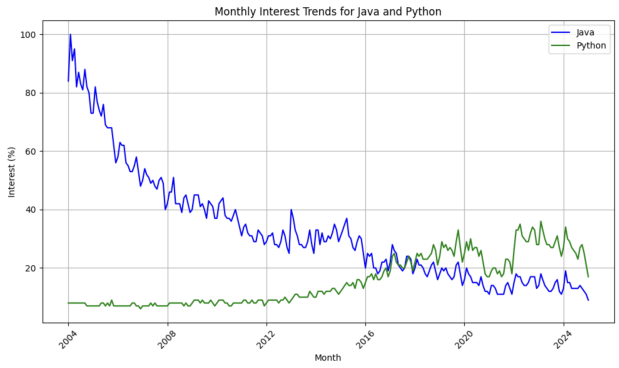

Figure 1: Java/Python Dominance

Python's dominance can be attributed to functional APIs such as Pandas, Modin, and Dask, a shallow learning curve, and user experience[21]. A key benefit of Python is it can interface directly with C/C++ or other native runtimes. Frameworks such as Pandas are not optimized with HPC-compute kernels. Therefore, bindings such as Cython, which allow access to C++, can be used to create high-performance APIs for high-performance kernels available in HPC clusters such as UVA's Rivanna Supercomputer[26].

The rise of Big Data and cloud computing, which began in the early 2000s, played a pivotal role in the emergence of modern artificial intelligence and machine learning. Machine learning and deep learning applications rely heavily on large, predetermined, and preprocessed datasets[2]. The exponential growth of data has posed significant challenges, particularly in terms of traditional storage and the need for efficient data exchange between distributed storage locations. Cloud computing services, such as Amazon Web Services (AWS), offer innovative solutions to these challenges by providing shared resources, including computing, storage, and analytics[13].

Function as a Service (FaaS) is a rapidly evolving paradigm in cloud computing applications. In FaaS, users focus on writing code divided into individual functions. This approach allows users to concentrate on the application code itself, rather than on deploying and managing the underlying compute and storage infrastructure. Serverless functions also provide elasticity and fine-grained billing capabilities. However, serverless currently lacks efficient communication through message passing for large-scale parallel data processing applications, such as Cylon[7]. To address this limitation, Network Address Translation (NAT) TCP Hole Punching offers a high-performance and cost-effective alternative to other methods, such as object or in-memory storage used on AWS Lambda to circumvent communication challenges[17]. Copik et al. detail performance experiments of the FMI library in the paper "FMI: Fast and Cheap Message Passing for Serverless Functions." Performance improvements for FMI usage are observed to be between 105 and 1024 times better than that of AWS DynamoDB. Additionally, the authors detail experiments demonstrating performance converging comparably to FMI and MPI on EC2 around 8 nodes and note improvements in performance for distributed machine learning FaaS applications 162 times and cost reductions up to 397x times[7].

There are documented serverless applications for Bioinformatics analysis, leveraging AWS Lambda functions' embarrassingly high parallelism and scalability. Grzesik et al. describe a serverless application that mines single nucleotide polymorphism (SNP) data. The application pushes data to an Amazon S3 bucket, which is later processed asynchronously. [9] Similarly, Robert Aboukhalil describes serverless executions using *Cloudflare*, a serverless execution service that connects to providers like AWS to simulate DNA sequencing data via streaming to AWS S3. [3] Hung et al. describe a serverless approach for RNA Sequencing Data analysis. Like other AWS Lambda applications, an S3 bucket is used as an intermediate layer for data transfer. The authors acknowledge design challenges related to the impact of data transfers from the AWS Lambda function to the AWS S3 bucket. [12] This is the gap our proposed research aims to fill. We propose the use of Nat Hole Punching to provide a high-performance alternative to intermediate storage solutions like AWS S3. This will enable function-to-function bidirectional communication, a capability not natively supported by AWS Lambda. This is also an ideal choice for AWS Fargate based on the difficulties implementing high-performance communication on serverless docker for libraries such as UCX, MPI and Gloo.

Hydrology, earthquake prediction, and astronomical image data processing are additional examples of time series foundational models that could be applied and executed on serverless cloud systems like AWS Lambda. For instance, in the paper "Science Time Series: Deep Learning in Hydrology," the authors describe a Python Jupyter notebook that runs on Google Collab to analyze hydrology time series using the CAMELS and Caravan global datasets for rainfall and runoff[11]. Another paper, "Time Series Foundation Models and Deep Learning Architectures for Earthquake Temporal and Spatial Nowcasting," describes earthquake time series forecasting using MultiFoundationQuake and GNNCoder for the next 14 days using Southern California earthquake data from 1986 to 2024[14]. There is existing work to apply and integrate this research in the form of inference and training in Cylon, which could be integrated with Serverless via our proposed communicator integration. Lastly, Cloud-base Astronomy Inference (CAI), a serverless application of astronomical image data processing, is detailed in the paper "Scalable Cosmic AI Inference using Cloud Serverless Computing with FMI." This paper utilizes the FMI library for collectives and aligns with the proposed AWS Lambda architecture described later. The authors plan to integrate Cylon into CAI once the FMI communicator integration is completed[27].

This work presents a proposed subset of research exploring parallelism and vectorization innovatively across Serverless AWS and HPCs using Apache Arrow and Apache Parquet which is detailed below. We suggest employing NAT Traversal TCP Hole Punching as a novel approach to enable Serverless AWS Lambda functions and AWS Fargate for embarrassingly parallel execution on AWS.

In this proposal, we'll explore Cylon, a runtime engine designed to optimize the execution of deep learning applications in high-performance computing (HPC) and cloud environments. We introduce and incorporate Cylon to achieve an existing goal of parallel computing, making it implementable without user intervention. This is accomplished by utilizing libraries of previously parallelized operators (such as join and all the other Pandas operators, as well

as deep learning). We'll explain the advantages of executing workloads within Docker containers on AWS using ECS tasks and discuss how this approach can be extended to HPC environments utilizing Singularity or Apptainer so that we can execute derivative experiments.

Next, we'll address the challenges of large-scale parallel execution on AWS and propose integrating the FMI library into Cylon for preprocessing and inference in serverless environments. By demonstrating the library's effectiveness, we'll highlight its applicability in scientific fields like Bioinformatics, Hydrology, Earthquake Prediction, Astronomy, and Genome research, where it can address high-performance communication issues observed in current state-of-the-art approaches, such as distributed storage latency and data parallelism. Furthermore, we'll explore the cost savings associated with FaaS execution to showcase the relevance and innovation of this approach compared to serverful execution in traditional HPC environments. Based on results published in Copik and al.'s paper, "FMI: Fast and Cheap Message Passing for Serverless Functions," we believe we can improve performance compared to object storage by two orders of magnitude for the *allreduce* collective operation and achieve comparable performance on AWS Lambda and AWS Fargate to performance on ECS with a CPU bound EC2 cluster[7].

2 Related Works

One of the fundamental concepts in data science is the dataframe. In the Python ecosystem, Pandas[16] was created as a Python derivative of the R *data.frame* class. However, one significant limitation of Pandas is that it can only execute on a single core. In contrast, frameworks like Apache Spark, a Java Virtual Machine (JVM) framework, offer similar capabilities and improved performance and support the dataframe abstraction. One notable drawback of Spark is that it operates as a framework rather than a standalone Python library. While there is support for Python through PySpark, its usage necessitates the configuration, setup, and deployment of a Spark cluster. The JVM-to-Python translations introduced by Spark add a substantial performance bottleneck compared to the C++-to-Python translation implemented by Cylon and other high-performance Python libraries, which are more lightweight[2].

Similar to the Cylon library, the Twister2 toolkit developed by Kamburugamuwe et al. is implemented using the Bulk Synchronous Parallel (BSP) architecture, based on the observed advantages of scalability and performance. It also incorporates a dataframe API implemented in Java[22]. A crucial abstraction introduced by Twister2 is TSets, a concept similar to dataframes or equivalently RDDs in Apache Spark or datasets in Apache Flink. Twister2 is regarded as a foundational step towards high-performance data engineering research and serves as a direct precursor to Cylon[21].

The Dask, Modin, and Ray Python distributed dataframe packages are built on the Pandas API and share architectural goals similar to those of the Cylon architecture. A related project, Dask-Cudf, provides distributed dataframe capabilities on Nvidia GPUs, leveraging both Dask and Pandas to facilitate integration. In Perera et al.'s paper, "Supercharging Distributed Computing Environments for High Performance Data Engineering," the authors describe the potential to "supercharge" Dask/Ray performance by integrating Cylon using actors to create a highly efficient HP-DDF runtime.

To address the need for a uniform distributed architecture, Jeff Dean describes Google's Pathway architecture, which aims to address the future of AI/ML systems as researchers migrate from Single Program Multiple Data (SPMD) to Multiple Program Multiple Data (MPMD). However, this system is closed-source. In response, the Cylon Radical Pilot has been developed to support the execution of heterogenous workloads on HPCs. [24]

Nvidia describes Rapids as "...an open-source suite of GPU accelerated data science and AI libraries with APIs that match the most popular open-source data tools..." One interesting open-source integration of Rapids is with cuDF dataframes. This integration enhances the performance of Pandas and offers a 100% compatible API with Pandas, a shared goal with the Cylon project[23].

Another related project, Wukong, offers a similar capability, including support for AI/ML inference, but it was built on serverless FaaS where AWS manages provisioning, scaling, and other undifferentiated tasks. This DAG execution framework has demonstrated near-ideal scalability by executing jobs approximately 68 times faster while achieving nearly 92 percent cost savings compared to NumPyWren. Wukong uses Redis as a key-value store for intermediate and final storage to address data parallel and communication performance described by related works in the scientific domain.

A significant challenge in serverless execution, particularly for parallel data processing applications, is data transfer between running functions. Moyer proposes a solution that leverages Nat Traversal via TCP Hole Punching through an external rendezvous server to enable direct TCP connections between a pair of functions. For an experiment involving over a hundred functions, this approach resulted in a performance improvement of 4.7 times compared to using object storage. [17]

A notable difference between Moyer's work and the FMI library lies in Moyer's implementation, which utilizes web sockets for communication. In contrast, FMI is developed as a C++ library making it highly applicable to executing parallel processing tasks on serverless architectures using Cylon.

3 Design and Implementation

This section outlines the overarching design of Cylon. The design is shaped by insights gained from developing the Twister2 toolset. Specifically, Cylon aims to achieve the following goals: 1) High performance and scalability, 2) An extensible architecture, 3) User-friendly APIs, and 4) Integration across multiple platforms. Following this overview, we will shift to an analysis of our AWS stack's semantics, which builds on this groundwork. We will explore the challenges and architecture of serverless data engineering at scale, emphasizing the novelty and importance of our proposed work[21].

3.1 Cylon Design

Cylon was developed to address shortcomings, interact directly with high-performance kernels, and perform better than libraries such as Pandas. Cylon represents an architecture where performance-critical operations are moved to a highly optimized library. Moreover, the architecture can leverage the performance associated with in-memory data and distributed operations and data across

processes, an essential requirement for processing large data engineering workloads at scale. Such benefits are realized, for example, in the conversion from tabular or table format to tensor format required for Machine Learning/Deep Learning or via relational algebraic expressions such as joins, select, project, etc. A key aspect of the Cylon architecture is the intersection of data engineering and AI/ML, allowing it to interact seamlessly with frameworks such as Pytorch[20] and TensorFlow[1].

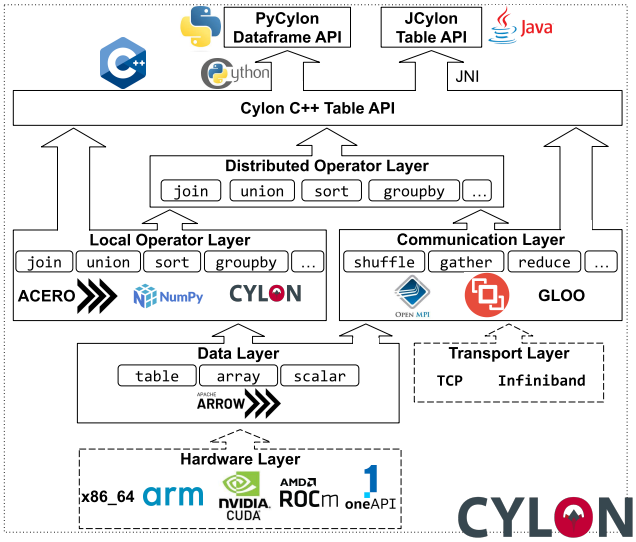

Figure 2: Cylon Architecture

Figure 2 provides a high-level overview of the Cylon Architecture. Solid boxes represent features and capabilities implemented within the Cylon runtime. The table layer encompasses tables, arrays, and scalars managed by the Apache Arrow runtime. Cylon supports Parquet integration, a file format that optimizes data storage by using a columnar format instead of a row format. Arrow facilitates loading Parquet files into Arrow for rapid, in-memory processing and then saving the results back into Parquet for long-term storage.

The communication layer represents the interface between Cylon and supported communication libraries. Currently, Cylon supports OpenMPI, UCX, and Gloo. Both distributed and local operators are supported. Cython bindings facilitate Python access to C++ data structures, functions, and classes.

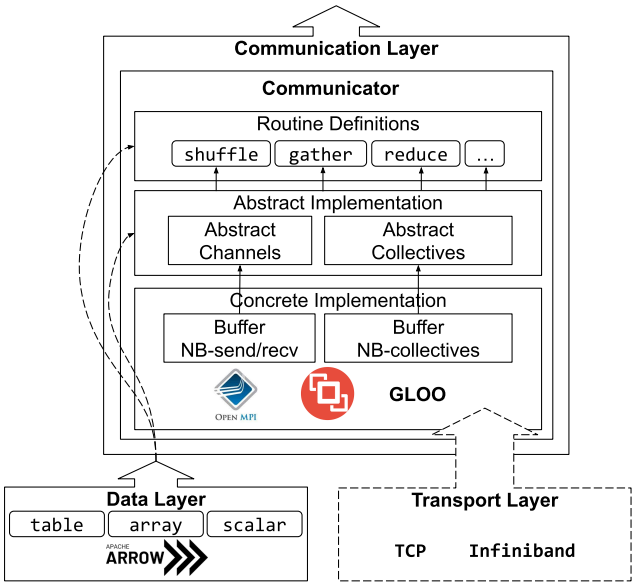

Figure 3: Cylon Communication Model

Figure 3 illustrates the Cylon communication interface. Two key features of this model are its modular architecture and extensibility, as discussed in Perera's thesis, "Towards Scalable High Performance Data Engneering Systems." To simplify the complexities of distributed programming, frameworks like TCP and Infiniband facilitate plug-and-play configuration, supporting high-performance communication libraries such as OpenMPI, UCX, and Gloo. The Cylon architecture is based on the Apache Arrow Columnar format. This library offers serialization-free copying across various runtimes, including Pandas and Numpy, as mentioned in "Towards Scalable High Performance Data Engneering Systems."[21]. We anticipate that Cylon will profoundly impact the development of high-performance data engineering frameworks and ultimately contribute and help shape the integration of ML/AI pipelines.

3.2 Pillars of Cloud Computing and Well-Architected Cloud Systems

Cloud computing is widely regarded as the cornerstone of the fourth industrial revolution. As companies and organizations increasingly rely on Big Data and Artificial Intelligence, cloud providers like AWS are pivotal in facilitating integration and innovation to support these endeavors. [10] AWS offers four fundamental pillars for successful cloud architectures. First, cloud architectures eliminate the uncertainty surrounding capacity requirements. Second, they provide virtually unlimited usage, enabling architectures to scale up and down autonomously. Third, cloud architectures facilitate testing systems at production scale. In other words, production-like environments can be created on demand, simulating production environment conditions. Cloud architectures also lower the risk associated with architectural change by removing the test serialization problem with on-premise infrastructure. This refers to the situation that can occur with software testing on-premise. In this case, tests are often executed in parallel due to limitations with

shared resources or hardware constraints. This can slow testing and create other bottlenecks that complicate testing and evaluation. To address this problem, cloud systems support automation as an enabler of architectural experimentation. This allows for the creation and replication of systems at low costs and enables tracking and auditing of impact with low effort. Lastly, cloud architectures allow for evolutionary architectures. This includes automating and testing on-demand to lower the impact of design change. With evolutionary architectures, systems can evolve so that it is possible to take advantage of innovation(s) as standard practice[4].

Well-architected cloud systems are built on four key pillars: security, reliability, performance efficiency, and cost optimization[4].

Security protects information systems and assets while delivering value through risk assessment and mitigation strategies. It involves applying security at all layers, enabling traceability, automating responses to security events, and treating automation as a best practice[4].

Reliability refers to a system's ability to recover from infrastructure or service failures, acquire resources dynamically to meet demand, and mitigate disruptions such as misconfiguration or transient network issues. Design principles of reliability include testing recovery procedures, automating recovery from failures, using horizontal scalability to increase aggregate system availability, and eliminating guesswork about capacity[4].

Performance Efficiency denotes the ability to use computing resources efficiently to meet system requirements and maintain that efficiency as demand changes and technology evolves. Key principles related to performance efficiency include democratizing advanced technologies, going global in minutes, and experimenting often[4].

Cost Optimization involves avoiding unnecessary costs and suboptimal resource use. Effective cost optimization includes using managed services to reduce the overall cost of ownership, trading capital expenses for operating expenses, taking benefits from economies of scale, and reducing spending on data center operations[4].

This proposal's proposed and related work aligns with the best practices outlined in the discussed pillars of successful cloud architectures and the key characteristics of well-architected cloud architectures.

3.3 Implementation

As part of this work, we propose implementing high-performance data engineering in serverless environments using Cylon and FMI to support the direct communication necessary for Bulk Synchronous Parallel architectures. We will discuss a reference serverful implementation for running data-parallel applications on HPCs and traditional cloud services such as EC2 using Cylon, and outline our proposal to integrate the FMI library into Cylon as a communicator for serverless infrastructure such as AWS Fargate and AWS Lambda.

3.4 Cylon High Performance Communication

Data parallelism in Cylon is built around Bulk Synchronous Parallel (BSP), a model that uses Single Program Multiple Data (SPMD) across compute nodes. The Message Passing Interface (MPI) is implemented by frameworks such as OpenMPI, MPICH, MSMPI, IBM Spectrum MPI, etc.

For most use cases, parallel data processing with OpenMPI Cylon is standard, based on the OpenMPI library's availability on HPCs like Rivanna and Summit. However, MPI isn't compatible with all frameworks, such as Dask and Ray. These distributed libraries are better suited for cloud technologies like Kubernetes and Amazon Elastic Container Service (ECS). OpenMPI includes process bootstrapping capabilities that facilitate the initialization of communication primitives and configuration. Cloud environments like AWS don't require this process management capability; therefore, communication libraries such as UCX/UCC are ideal. UCX is an open-source, production-grade communication framework for data-centric and high-performance applications[19]. We use UCX for point-to-point communication. UCC is a collective communication operations API and library that is flexible, complete, and feature-rich for current and emerging programming models and runtimes[18]. We use UCC for collective operations. A key benefit of UCX is the library is designed to be decoupled from network hardware. This provides increased portability while ensuring high performance and scalability. Another key advantage of UCX is that it allows access to GPU memory and bidirectional GPU communication. This library also supports Remote Direct Memory Access (RDMA) over Infiniband and Converged Ethernet (ROcE). Zero-copy GPU memory copy over RDMA is also supported, leading to exceptional performance. OpenSHEM is also included to support parallel programming[26].

For Cylon and as the predecessor of our proposed serverless work, we implemented a configuration mechanism that is modularized in such a way as to enable the implementation of new sources as they become available. We used Redis as a key-value store for our experiments to facilitate the bootstrapping process and provide barrier conditions. Another interesting detail about this implementation concerns resource usage, specifically with Unified Communication Protocols (UCP) or endpoints for a network connection. UCC also uses endpoints for collective operations. Unfortunately, reusing endpoints is impossible; therefore, we implemented separate endpoints for both UCC and UCP. This feature is not included with UCC/UCX and is necessary for parallel pre-processing tasks and BSP style execution using Cylon. The integration of UCX as a Cylon communicator involves the following: 1) Serializer, 2) Communication Operators, 3) Channels and AllToAll, and 4) Integration with dataframe operators[26].

Within Cylon, the UCX communicator serializes tables, columns, and scalars into buffers before being passed to the network communication layer. As was mentioned earlier, Cylon implements a Distributed Memory Dataframe (DDMF) via Apache Arrow. This is represented as a collection of P dataframes or partitions of lengths $\{N_0, ...N_{P-1}\}$ and a schema denoted as S_m. The total length of the Distributed Memory Dataframe can be represented as $\sum N_i$ and the concatenation of row labels as $R_n = \{R_0, R_1, ...R_{P-1}\}$. Figure 4 illustrates this process.

The Cylon channels API implements the AllToAll operation as point-to-point communication and uses the UCC library for collective operations such as AllGather. Distributed operators are implemented generically within Cylon. The experiments detailed in the next section use the Distributed Join DataFrame operator.

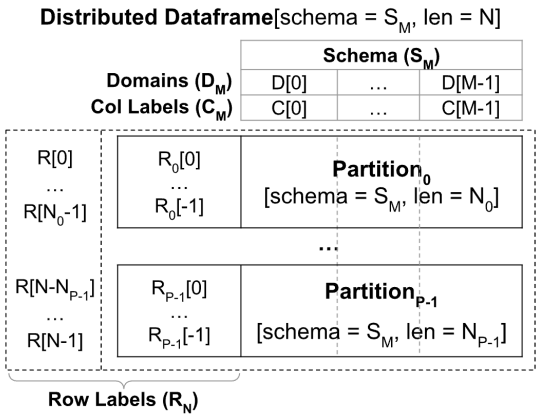

Figure 4: Distributed Memory Dataframe (DDMF)[21]

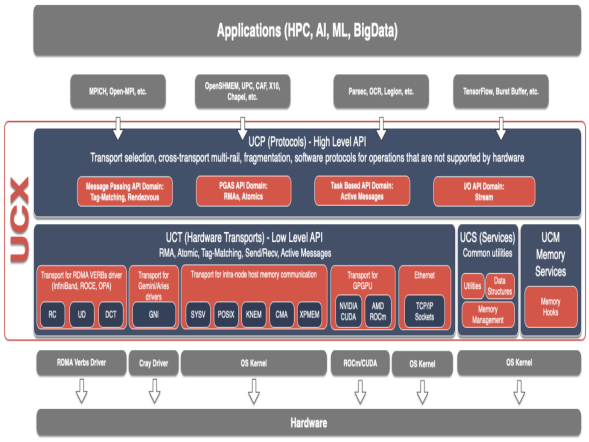

Figure 5: Unified Communication X (UCX) Library Architecture[25]

For this case, the process follows: 1) Hash applicable columns into partitioned tables, 2) Use AllToAll to send tables to the intended destination, and 3) Execute a local join on the received tables[26].

The initialization or bootstrapping of the Cylon UCX/UCC communicator necessitates the provision of communication metadata during startup. This metadata includes each process's world size, rank, and endpoint address. We utilize the *Redis* key-value store to facilitate this exchange. Our design increments an atomic value to represent the rank. Before incrementing the value, the rank is set. Redis is launched externally to the Cylon processes. The only requirement is that the *Redis* host be provided before initializing the Cylon runtime. Our experiments employed a combination of AWS ElastiCache, Redis, and ECS tasks, including the dockerized *Redis* runtime. Depending on the specific experiment's communication requirements, we selected a combination of *Redis* deployments. One implementation limitation is that the stored metadata on *Redis* must be cleared between subsequent experiments. Otherwise, the experiment executes non-deterministically and ultimately fails. In the case of executing N experiments of a given class (e.g., world size of 8 for 9.1 million rows), we need to configure N *Redis* hosts when running experiments in parallel.

Our experiment design, which utilized UCX/UCC for our initial experiments, proved highly effective for most AWS and HPC experiments. One aspect of our design involved using Docker. Docker uses OS-level virtualization to deploy software in packages or containers to create a runtime environment that could be seamlessly deployed across both HPC and AWS[6]. We pushed our Dockerized environments to Dockerhub and employed Apptainer on Rivanna to construct a Singularity container encapsulating the Cylon runtime. A significant advantage of this approach is that modifications to the environment (or Dockerfile) were not required. Consequently, all our experiments utilized the same runtime, resulting in a consistent and derivative approach across infrastructure. We believe our architecture and the inclusion of UCX/UCC to be a novel contribution on its own, and based on our research, it is the only case we could find where UCC/UCX was being used for ML/AI pipelines such as Cylon.

A crucial aspect of our proposed research involves executing data engineering pre-processing on serverless compute. For our initial exploratory work, we modified the UCX Unified Communication Transport (UCT) lower-level API. This change was necessary due to UCX's initialization behavior. UCX employs low-level system calls to select the most suitable and optimal communication protocol based on the available resources. However, only TCP is supported for serverless computing. We observed that the UCX device query incorrectly retrieved the wrong IP address from the hardware during initialization. To ensure successful initialization, we had to override the IP address discovered by UCX. This modification enabled us to conduct experiments on AWS Fargate, a serverless Docker running on ECS, a fully managed container orchestration service. Unfortunately, this capability was deprecated due to changes in the underlying serverless architecture implemented by AWS. This result led us to investigate other potential communication options for BSP systems that operate on serverless systems and represents our proposal for this research.

A similar issue was encountered with AWS Lambda functions: we could not leverage the IP address override capability mentioned above to execute Cylon on AWS Lambda due to the unavailability of system calls, a similar issue observed on AWS Fargate. One solution presented by Copik et al. was to use NAT traversal or TCP hole punching to enable direct communication between a pair of AWS Lambda functions via the FMI library. Unlike storage-based solutions, direct communication decreases communication time by multiple orders of magnitude. Figure 6 illustrates this process. NAT hole punching requires functions to be behind a NAT gateway. The gateway hides the addresses of each function by rewriting the internal addresses with an external address. To support NAT Hole Punching, a publicly accessible server must create entries in the translation table and relay addresses between two functions.

Our initial plan was to integrate the FMI library into Cylon by modifying the existing TCP UCT UCX source code. However, this

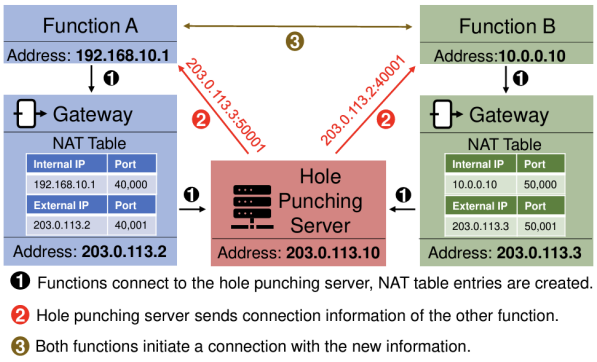

Figure 6: Network Address Translation (NAT) Hole Punching[7]

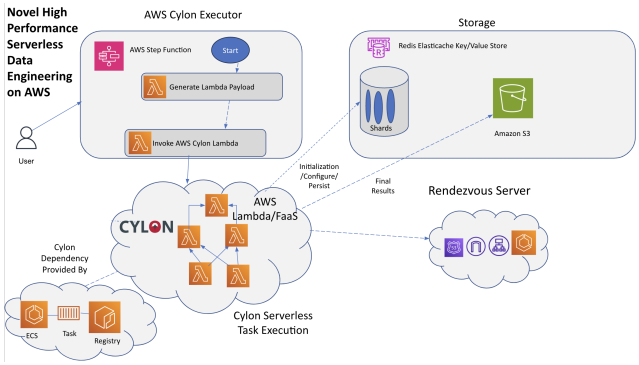

Figure 7: Serverless Data Engineering Architecture

approach proved unsuccessful due to the UCX TCP architecture and NAT hole-punching address exchange semantics.

We believe a more suitable approach is implementing FMI as a separate Cylon communicator, similar to the UCC/UCX communicator described earlier. During our initial work that led to this proposal, we successfully recreated and validated the FMI library's functionality as a custom AWS Python Lambda Docker image.

To achieve this, we created a Dockerfile that included all the necessary libraries for executing experiments. We also implemented the AWS Lambda interface and developed derivative experiments, such as the distributed join experiments detailed in the next section.

Including this library is an ideal choice, considering the limitations imposed by AWS Lambda. A higher-level abstraction is necessary for direct communication between Lambda functions. Furthermore, there are cost considerations based on the AWS Lambda cost model, where request rate, execution time, and function size directly impact the overall cost. By supporting direct communication in serverless, we can save costs associated with data storage compared to the current state-of-the-art for serverless applications in domains like Bioinformatics. These applications often require parallel, high-performance analysis that would benefit from high-performance communication. Based on results published in the paper, "FMI: Fast and Cheap Message Passing for Serverless Functions," we believe we can achieve similar savings (i.e., 397 times savings compared to the use of mediated storage for Machine Learning applications reported by Copik et al.)[7].

Figure 7 illustrates our proposed architecture. We suggest creating an AWS Step Function to implement the AWS Cylon Executor, as depicted in Figure 8. AWS Step Functions are a Software as a Service (SaaS) feature that facilitates the orchestration of AWS services into serverless workflows. The *Lambda: init* function will receive a payload containing various parameters, such as the number of rows, world size, number of iterations, target S3 bucket, script, and output paths. We propose utilizing a specific S3 bucket to store invocation results in a folder using the *Boto3* library. This approach is advantageous over parsing the data output from log files in AWS CloudWatch, a SaaS metrics repository created and utilized by AWS services, as it would be cumbersome. The Lambda invoke state will forward this input JSON payload to a *fmi_init* AWS Lambda function. This function will validate the rendezvous server's availability and generate an N-sized array of JSON payloads, which will then be sent to the *Map, ExtractAndInvokeLambda* states, which will be configured to use Distributed processing mode to enable highly parallel execution. Next, each generated payload will be sent to the Lambda: Invoke state from the pass *ExtractAndInvokeLambda* state. This state will pass the input to output, executing the *fmi_executor* AWS Lambda function. The runtime of the *fmi_executor* function will include the Cylon runtime proposed earlier, which consists of a communicator based on direct communication implemented by the FMI library described earlier. The incoming payload specifies the location of the target script in the S3 bucket. Upon execution of the Lambda function, the contents of the specified S3 bucket will be retrieved and persisted into temporary ephemeral storage. Subsequently, Python3 will be used to execute the script used for our experiments. We plan to support a single script and a folder containing Python scripts and modules.

During execution, the *fmi_executor* will connect with the Rendezvous Server referenced in Figure 7. Once the second function in the pair connects to the Rendezvous Server, both servers will receive the destination function's public address. The Python script will continue to execute until the collective operations are performed. For the initial experiments discussed below, we used an AllReduce collective to calculate a summation of start-to-end execution across the world size of executing functions. Lastly, We also plan to use the cloudmesh stopwatch library to facilitate logging and benchmarking the runtime of our executing experiments[5].

4 Experiments

In preparation and as background for this proposal, we conducted microbenchmark experiments for the distributed join operation detailed in Table 1. The size of the infrastructure relative to memory and CPU cores was configured based on the memory available to AWS Lambda functions, which has a maximum of 10240 MB. Therefore, we configured a maximum and a close to half-size configuration for each set of experiments to be 10 GB and 6 GB.

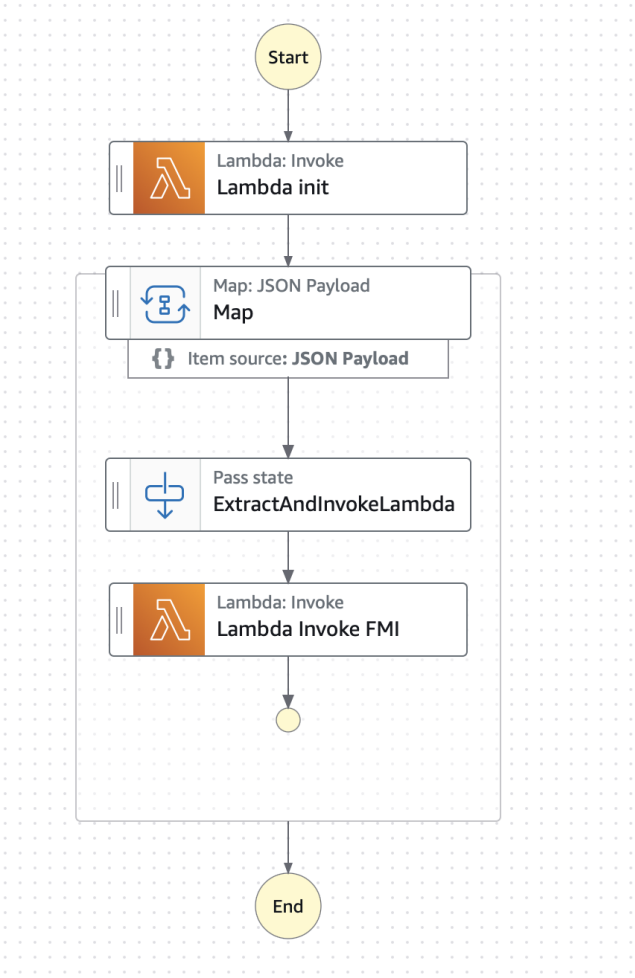

Figure 8: AWS Step Function for Serverless Data Parallel Task

4.1 Join Operation Scalability

For these experiments, we ran scaling experiments plotted as speedup on AWS Lambda, EC2 ECS clusters and Rivanna. The experiment design was to run experiments on derivative-configured infrastructure. For EC2, we used Ubuntu 22.04.2 LTS (Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz) for hosts configured with two virtual CPU and 15 GB of memory (m3xlarge) and for hosts configured with one virtual CPU and 7.5 GB of memory (m3large). For Rivanna, we ran experiments on 1 (standard partition) and 2, 4, 8, 16, 32, and 64 (parallel partition) CPU nodes (Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz). To match the performance and characteristics of AWS infrastructure on Rivanna, we configured experiments that would run on 10 and 6 GB memory configurations. Rivanna experiments were executed in a Singularity containers so that all experiments across infrastructure would be executed using dockerized processes. For AWS Lambda, functions were configured to use 6GB and 10GB of memory. Memory configuration influences the CPU available. For the 6 GB configuration, 4 CPU cores are available, and for the 10 GB configuration, 6 CPU cores are available. Both configurations use a custom-built AWS Lambda container with all necessary Python and native libraries. We used the FMI library and attempted to create a derivative experiment to Cylon that approximated the join operation. The Python code block in Listing 1 details the derivative experiment. In this case, we determine the use of FMI by the env variable, use Panda data frames versus Cylon data frames, call the Panda concat operation, and flatten as a list. We follow a similar procedure for the Cylon experiments, except we call the merge operation, which is implemented as a distributed operation. This is found in lines 17-23.

Table 2 summarizes the weak scaling results from join operations for this set of experiments. We used 9.1 million rows for weak scaling; for strong scaling, we used 4.5 million rows. The choice of 4.5 million rows was necessary based on memory limitations imposed by AWS Lambda. Note that executions on AWS Fargate were intended, but changes were implemented by AWS that resulted in breaking the UCX transport discovery mechanism. As part of this proposal, we plan to re-execute experiments using a custom Cylon communicator for AWS Fargate and AWS Lambda described earlier.

For these scaling experiments, we calculate speedup as the execution time ratio to the base case, which is represented by blue dotted lines in Figure 9 and Figure 10. The base case is considered the slowest execution or the first node across infrastructure. Speedup is defined as:

$$Speedup = \frac{\text{Baseline Time}}{\text{Parallel Time}}$$

For the standard deviation or error propagation, we used the following formula applied to the observed standard deviation across executions (i.e., four separate experiments were completed for each node).

Table 1: Experiments Setup for Comparative to AWS Lambda on UVA Rivanna HPC and Amazon Web Services. WS/SS = weak/strong scaling; M=Million

Experiment Description	Rows	Rows Size
AWS ECS EC2 4 CPU/15 GB Mem WS/SS	1-64	[9.1 | 4.5M]
AWS ECS EC2 2 CPU/7.5 GB Mem WS/SS	1-64	[9.1 | 4.5M]
Rivanna 10 GB Mem WS/SS	1-64	[9.1 | 4.5M]
Rivanna 6 GB Mem WS/SS	1-64	[9.1 | 4.5M]
Lambda 10 GB Mem WS/SS	1-64	[9.1 | 4.5M]
Lambda 6 GB Mem WS/SS	1-64	[9.1 | 4.5M]

```python
import pandas as pd
from numpy.random import default_rng



rng = default_rng(seed=rank)
data1 = rng.integers(0, int(max_val * u), size=(num_rows, 2))
data2 = rng.integers(0, int(max_val * u), size=(num_rows, 2))

if data['env'] == 'fmi':
    df1 = pd.DataFrame(data1).add_prefix("col")
    df2 = pd.DataFrame(data2).add_prefix("col")
else:
    df1 = DataFrame(pd.DataFrame(data1).add_prefix("col"))
    df2 = DataFrame(pd.DataFrame(data2).add_prefix("col"))
...

if data['env'] == 'fmi':
    df3 = pd.concat([df1, df2], axis=1)
    result_array = df3.to_numpy().flatten().tolist()

else:
    df3 = df1.merge(df2, on=[0], algorithm='sort', env=env)

if data['env'] == 'fmi':
    sum_t = communicator.allreduce(t, fmi.func(fmi.op.sum),
          fmi.types(fmi.datatypes.double))
    tot_l = len(communicator.allreduce(result_array,
          fmi.func(fmi.op.sum),
          fmi.types(fmi.datatypes.int_list, len(result_array))))

else:
    sum_t = communicator.allreduce(t, ReduceOp.SUM)
    tot_l = communicator.allreduce(len(df3), ReduceOp.SUM)
...
```

Listing 1: Python Scaling Join Experiment

$$\frac{\Delta S}{S} = \sqrt{\left(\frac{\Delta T_1}{T_1}\right)^2 + \left(\frac{\Delta T_2}{T_2}\right)^2}$$

Where:

S is the Speedup: S = $\frac{T_1}{T_2}$

ΔS is the error in speedup

T_1 is the baseline execution time (the time for one node)

ΔT_1 is the error in baseline execution time

T_2 is the parallel execution time

ΔT_2 is the error in parallel execution time

Figure 9 depicts the weak scaling experiments. In an ideal scenario, we should see close to constant performance. However, based on system overhead, we mitigate variation by executing ten iterations for each experiment. In general, as the parallelism is increased, resource and scheduling overhead contributes to the decreasing performance across weak scaling experiments. We anticipate that our proposed integration of FMI as a communicator into Cylon will result in performance that approximates EC2 weak scaling with deviation(s) of less than 1 percent. Weak scaling of FMI shows better performance in this set of experiments due to the differences between the experiments (i.e., distributed merge versus array flattening for Cylon and local Pandas without distributed merge usage

Table 2: Execution Time of Weak Scaling from Join Operations Across Infrastructure.

Infrastructure	Scaling	Parallelism	Execution Time time (seconds)
EC2 15GB Mem 4 vCPUs	Weak	1	31.57 ±0.20
		2	40.42 ±1.09
		4	42.48 ±1.67
		8	44.08 ±1.50
		16	47.84 ±0.78
		32	49.83 ±0.47
		64	52.70 ±0.34
EC2 7.5GB Mem 2 vCPUs	Weak	1	31.71 ±0.48
		2	43.63 ±0.43
		4	46.56 ±0.31
		8	49.11 ±0.56
		16	51.12 ±0.22
		32	50.97 ±0.30
		64	54.98 ±0.49
Lambda FMI 10 GB Mem	Weak	1	20.41 ±0.32
		2	20.03 ±0.83
		4	20.69 ±0.53
		8	21.07 ±0.25
		16	20.14 ±0.52
		32	22.21 ±0.70
		64	22.82 ±0.40
Lambda FMI 6 GB Mem	Weak	1	18.67 ±0.93
		2	20.05 ±0.68
		4	20.77 ±0.48
		8	21.21 ±0.42
		16	22.92 ±0.97
		32	21.72 ±0.39
		64	22.60 ±0.22
Rivanna 10 GB Mem	Weak	1	18.24 ±0.04
		2	20.60 ±0.12
		4	20.78 ±0.04
		8	21.40 ±0.37
		16	23.05 ±0.50
		32	24.03 ±0.25
		64	36.92 ±1.70
Rivanna 6 GM Mem	Weak	1	18.27 ±0.10
		2	20.60 ±0.11
		4	20.72 ±0.22
		8	21.42 ±0.33
		16	23.05 ±0.34
		32	24.89 ±1.26
		64	36.14 ±0.79

with FMI). The intention in this particular case was to approximate Cylon experiments with FMI usage. However, the disparities are apparent in weak scaling, necessitating FMI integration into Cylon to validate our claims related to communicator performance.

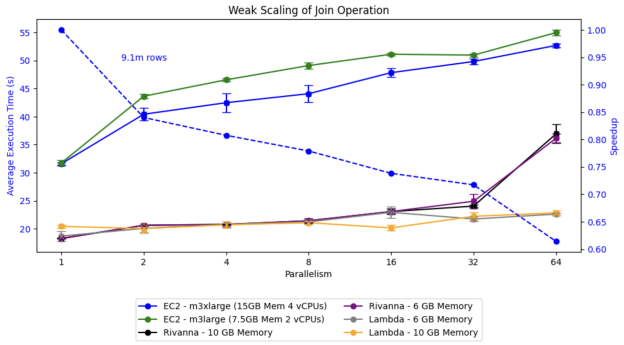

Figure 9: Comparison of weak scaling for the Join operation across infrastructure including AWS Lambda

Table 3 presents the results of strong scaling experiments involving 4.5 million rows. Like weak scaling, we conducted each experiment four times, performing ten iterations for each trial. The results of strong scaling are illustrated in Figure 10. We observe high latencies across the infrastructure for fewer than four nodes. However, we also observe a general convergence of latency as the number of nodes ranges from 16 to 64. This supports utilizing cloud resources as a viable alternative for data-parallel preprocessing workloads. Similar to weak scaling, we will re-execute experiments on AWS Lambda based on the dissimilar nature of the current experiments that used FMI without distributed merge following FMI integration in Cylon. We believe the performance of AWS will fall below one percent of strong scaling experiments with EC2 and Rivanna based on implementing direct communication via NAT Traversal TCP Hole Punching for serverless architectures such as AWS Lambda and AWS Fargate.

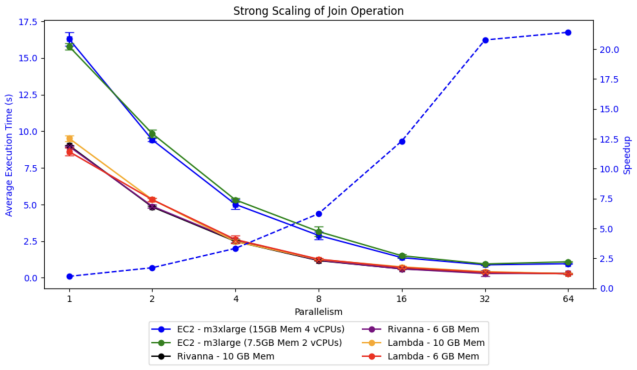

Figure 10: Comparison of strong scaling for the Join operation across infrastructure including AWS Lambda

Table 3: Execution Time of Strong Scaling from Join Operations Across Infrastructure.

Infrastructure	Scaling	Parallelism	Execution Time time (seconds)
EC2 15GB Mem 4 vCPUs	Strong	1	16.28 ±0.45
		2	9.41 ±0.11
		4	5.00 ±0.32
		8	2.89 ±0.27
		16	1.37 ±0.01
		32	0.88 ±0.01
		64	0.96 ±0.01
EC2 7.5GB Mem 2 vCPUs	Strong	1	15.78 ±0.22
		2	9.83 ±0.25
		4	5.31 ±0.12
		8	3.15 ±0.33
		16	1.50 ±0.10
		32	0.94 ±0.04
		64	1.09 ±0.01
Lambda FMI 10 GB Mem	Strong	1	9.51 ±0.18
		2	5.33 ±0.09
		4	2.47 ±0.13
		8	1.24 ±0.02
		16	0.74 ±0.06
		32	0.411 ±0.06
		64	0.28 ±0.030
Lambda FMI 6 GB Mem	Strong	1	8.61 ±0.25
		2	5.33 ±0.10
		4	2.60 ±0.26
		8	1.26 ±0.05
		16	0.70 ±0.04
		32	0.37 ±0.02
		64	0.27 ±0.02
Rivanna 10 GB Mem	Strong	1	9.03 ±0.01
		2	4.83 ±0.05
		4	2.48 ±0.09
		8	1.17 ±0.003
		16	0.61 ±0.007
		32	0.37 ±0.0007
		64	0.27 ±0.01
Rivanna 6 GM Mem	Strong	1	8.96 ±0.04
		2	4.88 ±0.10
		4	2.53 ±0.12
		8	1.19 ±0.001
		16	0.60 ±0.001
		32	0.29 ±0.19
		64	0.30 ±0.02

5 Future Work

Cylon implements about 30% of the operators supported by Pandas and we would like to converge Cylon and Panda operators using a comparative, high-performance API built using Cython.

For example, we have begun to implement the windowing operator and plan to complete that work soon. Utilizing the cost model (detailed by Perera in "Towards Scalable High Performance Data Engineering Systems"), we can evaluate data parallel operator patterns to identify bottlenecks in communication operations. We believe we can significantly improve performance by implementing other algorithms with lower latency[22]. Additionally, related to communicator support, we would also like to add libfabric as a communicator based on implementations across cloud services based on GPU hardware supported by cloud providers.

A key limitation with AWS Lambda is the upper limit of fifteen minutes for a given function invocation. With a BSP-styled architecture, this limits the applicability of data parallel processing on this infrastructure based on the time required to process large datasets. We want to implement a similar checkpointing feature is implemented in the Twister2 library. This will allow for the recovery of unfinished executions based on upper-limit time constraints imposed by AWS Lambda. This will also provide general fault tolerance for MPI-workloads outside of the serverless use case.

Lastly, we would like to integrate real-world applications and detail the performance and implications of serverless as a possibility for deep learning and machine learning inference on clouds. For example, we would like to implement hydrology and earthquake prediction deep learning on both serverful and serverless AWS to demonstrate the applicability and potential scale offered by cloud providers such as AWS.

6 Conclusion

In recent years, the growth of artificial intelligence and deep learning has necessitated improvements to data engineering systems to handle the increasing size and complexity of data. We have introduced Cylon, a high-performance C++ library that supports ETL pipelines in Python. Previous papers have highlighted Cylon's superior performance compared to other similar Python data engineering frameworks, such as Abeykook et al.'s paper, "Data Engineering for HPC with Python."

In this paper, we detailed our integration of UCX and UCC to replace MPI in BSP communication between processes based on the clear applicability to serverful cloud and HPC environments. The novelty of this work is apparent based on a survey of the literature on similar work involving distributed data frames. We applied this work directly and refined the initial implementation to support both serverful and serverless architectures of cloud providers like Amazon Web Services. Additionally, we implemented direct support for Python via Cython, enabling the execution of data engineering tasks in Python without the need to write native code. We also completed the integration of UCC into Cylon by removing our implementation of *AllGatherV* in favor of UCC's support for variable-length data. Furthermore, we implemented container support across infrastructures to facilitate the execution of experiments and library usage.

During this research, we encountered several challenges that resulted in delays. We first validated running Cylon on AWS by building the library using AWS Workspaces, which was generally successful. Using AWS facilitated building and running on bare metal because of complete control of the underlying ECS host. Comparatively, building on bare metal on Rivanna was plagued with difficulties based on library changes and security imposed by system administrators. In related work, RP-Cylon encountered similar delays in months due to challenges in building Cylon on Rivanna. That problem was ultimately solved by addressing fundamental problems with the build-to-install process. Later, we experienced a similar issue trying to build UCX Cylon after a year when updates broke the previously successful build process. We solved this problem by using Apptainer or Singularity and running Cylon in a container. We believe using Apptainer or Docker containers is the future for Cylon based on the inherent challenges associated with building native code on constrained hardware clusters such as Rivanna.

A second problem detailed elsewhere in this paper was related to implementing NAT hole punching within the UCX library. We intended to run Cylon in serverless and serverful environments using UCC/UCX. The idea was to modify the TCP capability in this library and implement a mechanism similar to that implemented in the FMI library. The reasoning behind this choice was FMI shared a similar architecture being developed in C++. After many months of effort, we were not able to achieve this. We learned that the semantics associated with endpoint communication establishment within UCX do not necessarily follow the specifics required by NAT Traversal techniques. In retrospect, we believe a fully custom UCX transport would have been an achievable albeit potentially complex solution. A key related challenge was the undocumented nature of the UCX library and the complexity of the C code within this library. Lastly, the device discovery mechanism implemented by UCX uses system-level calls to determine the most ideal and optimal communication hardware. For serverless environments, access to these system calls is prohibited. This led to the failure to execute AWS Fargate experiments based on a change or changes implemented by AWS.

A third problem was related to AWS funding. We were initially successful in receiving funding, but unfortunately, research funding expired before we could complete our work. It is an ongoing challenge to perform research on AWS based on the unavailability of AWS as a service by the University and the implicit requirement to apply for funding to use cloud providers for research, as detailed in this paper.

In this paper, we demonstrate experiments using Cylon and UCC/UCX, comparing them to similar experiments conducted on the Rivanna HPC cluster to validate the effectiveness and applicability of running data engineering tasks at scale on clouds. An emerging paradigm in cloud computing is serverless infrastructure, which allows developers to focus on software or applications without the burden of managing infrastructure or other undifferentiated workloads that SaaS can handle. We also discuss the challenges of using serverless, particularly in communication. We propose an approach that utilizes NAT traversal through NAT hole punching to enable direct communication between AWS Lambda functions. Additionally, we highlight the relevance of this work to serverless containers such as AWS Fargate and emphasize the importance of architecting communication layers at high levels of abstraction when integrating with serverless compute. We suggest the integration of NAT Traversal through NAT hole punching to achieve strong scaling performance similar to what is achievable on HPCs

such as Rivanna in Cylon. We believe this approach can be applied to bioinformatics analysis, genome research, hydrology, astronomy, earthquake prediction, and other time series foundational models based on a survey of the current state of the art in these fields and results published in Copik et al.'s paper, "FMI: Fast and Cheap Message Passing for Serverless Functions," which claims improvements between 105 and 1025 times that of storage mechanisms such as AWS S3, Redis, or AWS DynamoDB. Lastly, the author believes that integration and experiments should be achievable within one to three months, as documented in the Detailed Research Plan section. The only potential challenge could be an unidentified difference in communication semantics concerning the FMI library as it is currently implemented and documented on GitHub (https://github.com/spcl/fmi). The author considers this a low likelihood based on a thorough review of Cylon and FMI.

7 Detailed Research Plan

- February 3, 2025: Qualification Exam Proposal Presentation to Qualification Committee
- February 14, 2025: Written Qualification Exam Proposal submitted to Qualification Committee
- February 17, 2025: Begin Work on FMI Integration into Cylon
- March 15, 2025: Complete FMI Integration into Cylon
- March 31, 2025: Complete distributed Join experiment on AWS Lambda using integration NAT Traversal approach
- April 15, 2025: Draft Qualification Exam Report
- April 21, 2025: Deliver Qualification Exam Report to Qualification Exam Committee
- April 28, 2025: Qualification Exam

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Vibhatha Abeykoon, Niranda Perera, Chathura Widanage, Supun Kamburugamuve, Thejaka Amila Kanewala, Hasara Maithree, Pulasthi Wickramasinghe, Ahmet Uyar, and Geoffrey Fox. 2020. Data Engineering for HPC with Python. *arXiv preprint arXiv:2010.06312* (2020).

[3] Rob Aboukhalil. 2024. Serverless Genomics. https://robaboukhalil.medium.com/serverless-genomics-c412f4bed726 Accessed: February 7, 2025.

[4] Amazon Web Services. 2025. AWS Well-Architected Framework. https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html Accessed: 2025-01-09.

[5] Cloudmesh Community. 2025. *cloudmesh-common*. https://github.com/cloudmesh/cloudmesh-common Accessed: 2025-01-17.

[6] Wikipedia contributors. 2025. *Docker (software)*. https://en.wikipedia.org/wiki/Docker_(software) Accessed: 2025-01-17.

[7] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the 37th International Conference on Supercomputing*. 373–385.

[8] genome.gv. 2022. National Human Genome Research Institute. https://www.genome.gov/about-genomics/fact-sheets/Genomic-Data-Science. (Accessed on 01/03/2025).

[9] Piotr Grzesik, Dariusz R Augustyn, Łukasz Wyciślik, and Dariusz Mrozek. 2022. Serverless computing in omics data analysis and integration. *Briefings in bioinformatics* 23, 1 (2022), bbab349.

[10] Sadegh Hashemipour and Maaruf Ali. 2020. Amazon web services (aws)–an overview of the on-demand cloud computing platform. In *International Conference for Emerging Technologies in Computing*. Springer, 40–47.

[11] Junyang He, Ying-Jung Chen, Anushka Idamekorala, and Geoffrey Fox. 2024. Science Time Series: Deep Learning in Hydrology. *arXiv preprint arXiv:2410.15218* (2024).

[12] Ling-Hong Hung, Xingzhi Niu, Wes Lloyd, and Ka Yee Yeung. 2020. Accessible and interactive RNA sequencing analysis using serverless computing. *bioRxiv* (2020).

[13] Mohaiminul Islam and Shamim Reza. 2019. The Rise of Big Data and Cloud Computing. *Internet of Things and Cloud Computing* 7, 2 (2019), 45–53. https://doi.org/10.11648/j.iotcc.20190702.12 arXiv:https://article.sciencepublishinggroup.com/pdf/10.11648.j.iotcc.20190702.12

[14] Alireza Jafari, Geoffrey Fox, John B Rundle, Andrea Donnellan, and Lisa Grant Ludwig. 2024. Time Series Foundation Models and Deep Learning Architectures for Earthquake Temporal and Spatial Nowcasting. *GeoHazards* 5, 4 (2024), 1247–1274.

[15] A. McKenna. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research* 20 9 (2010), 1297–303. https://doi.org/10.1101/gr.107524.110

[16] Wes McKinney. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[17] Daniel William Moyer. 2021. *Punching Holes in the Cloud: Direct Communication between Serverless Functions Using NAT Traversal.* Ph. D. Dissertation. Virginia Tech.

[18] OpenUCX Contributors. 2024. UCC: Unified Collective Communication. https://github.com/openucx/ucc. Accessed: February 7, 2025.

[19] OpenUCX Contributors. 2024. UCX: Unified Communication X. https://github.com/openucx/ucx. Accessed: February 7, 2025.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[21] Dilshan Niranda Perera. 2023. Towards Scalable High Performance Data Engineering Systems. (2023).

[22] Niranda Perera, Arup Kumar Sarker, Mills Staylor, Gregor von Laszewski, Kaiying Shan, Supun Kamburugamuve, Chathura Widanage, Vibhatha Abeykoon, Thejaka Amila Kanewela, and Geoffrey Fox. 2023. In-depth analysis on parallel processing patterns for high-performance Dataframes. *Future Generation Computer Systems* (2023).

[23] RAPIDS AI. 2024. RAPIDS cuDF API Documentation. https://docs.rapids.ai/api/cudf/stable/ Accessed: February 7, 2025.

[24] Arup Kumar Sarker, Aymen Alsaadi, Niranda Perera, Mills Staylor, Gregor von Laszewski, Matteo Turilli, Ozgur Ozan Kilic, Mikhail Titov, Andre Merzky, Shantenu Jha, et al. 2024. Radical-Cylon: A Heterogeneous Data Pipeline for Scientific Computing. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 84–102.

[25] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[26] Kaiying Shan, Niranda Perera, Damitha Lenadora, Tianle Zhong, Arup Kumar Sarker, Supun Kamburugamuve, Thejaka Amila Kanewela, Chathura Widanage, and Geoffrey Fox. 2022. Hybrid Cloud and HPC Approach to High-Performance Dataframes. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2728–2736.

[27] Mills Staylor, Amirreza Dolatpour Fathkouhi, Md Khairul Islam, Kaleigh O'Hara, Ryan Ghiles Goudjil, Geoffrey Fox, and Judy Fox. 2025. Scalable Cosmic AI Inference using Cloud Serverless Computing with FMI. *arXiv preprint arXiv:2501.06249* (2025).