frontiers | Frontiers in High Performance Computing

TYPE Original Research
PUBLISHED 12 July 2024
DOI 10.3389/fhpcp.2024.1384619

OPEN ACCESS

EDITED BY
Scott Klasky,
Oak Ridge National Laboratory (DOE),
United States

REVIEWED BY
Jay Lofstead,
Sandia National Laboratories (DOE),
United States
Yuri Demchenko,
University of Amsterdam, Netherlands

*CORRESPONDENCE
Arup Kumar Sarker
djy8hg@virginia.edu

RECEIVED 10 February 2024
ACCEPTED 20 June 2024
PUBLISHED 12 July 2024

CITATION
Perera N, Sarker AK, Shan K, Fetea A, Kamburugamuve S, Kanewala TA, Widanage C, Staylor M, Zhong T, Abeykoon V, von Laszewski G and Fox G (2024) Supercharging distributed computing environments for high-performance data engineering. *Front. High Perform. Comput.* 2:1384619. doi: 10.3389/fhpcp.2024.1384619

Supercharging distributed computing environments for high-performance data engineering

Niranda Perera[1], Arup Kumar Sarker[2,3]*, Kaiying Shan[2], Alex Fetea[2], Supun Kamburugamuve[4], Thejaka Amila Kanewala[4], Chathura Widanage[4], Mills Staylor[2], Tianle Zhong[2], Vibhatha Abeykoon[4], Gregor von Laszewski[3] and Geoffrey Fox[2,3]

[1]Luddy School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN, United States, [2]Department of Computer Science, University of Virginia, Charlottesville, VA, United States, [3]Biocomplexity Institute and Initiative, University of Virginia, Charlottesville, VA, United States, [4]Indiana University Alumni, Bloomington, IN, United States

The data engineering and data science community has embraced the idea of using Python and R dataframes for regular applications. Driven by the big data revolution and artificial intelligence, these frameworks are now ever more important in order to process terabytes of data. They can easily exceed the capabilities of a single machine but also demand significant developer time and effort due to their convenience and ability to manipulate data with high-level abstractions that can be optimized. Therefore it is essential to design scalable dataframe solutions. There have been multiple efforts to be integrated into the most efficient fashion to tackle this problem, the most notable being the dataframe systems developed using distributed computing environments such as Dask and Ray. Even though Dask and Ray's distributed computing features look very promising, we perceive that the Dask Dataframes and Ray Datasets still have room for optimization In this paper, we present CylonFlow, an alternative distributed dataframe execution methodology that enables state-of-the-art performance and scalability on the same Dask and Ray infrastructure (*supercharging* them!). To achieve this, we integrate a *high-performance dataframe* system Cylon, which was originally based on an entirely different execution paradigm, into Dask and Ray. Our experiments show that on a pipeline of dataframe operators, CylonFlow achieves 30× more distributed performance than Dask Dataframes. Interestingly, it also enables superior sequential performance due to leveraging the native C++ execution of Cylon. We believe the performance of Cylon in conjunction with CylonFlow extends beyond the data engineering domain and can be used to consolidate high-performance computing and distributed computing ecosystems.

KEYWORDS

data engineering, data science, high performance computing, distributed computing, dataframes

1 Introduction

Data engineering has grown rapidly in recent decades, driven by the Big Data revolution and advances in machine learning (ML) and artificial intelligence (AI). In today's information age, data is measured in gigabytes and terabytes, stored in object repositories rather than megabytes, files, or spreadsheets. Managing this vast amount of data takes up significant developer time in preprocessing, detracting from the more

critical task of building data engineering models. Hence, it is essential to improve the efficiency of data preprocessing to develop effective data engineering pipelines.

Traditionally, data preprocessing was done using structured query language (SQL) in database systems. However, Python and R programming languages have increasingly taken on these SQL tasks in recent years. The Python library pandas has been instrumental in this transition, significantly boosting Python's popularity for data exploration. This discussion mainly focuses on the DataFrame (DF) API, a crucial part of the pandas framework. According to *PyPI* package index statistics, pandas consistently surpasses 100 million downloads per month, underscoring its leading role in the field (PyPI, n.d.). Despite its widespread use, both `Pandas` and R DF encounter performance limitations, even when handling moderately large datasets. (Petersohn et al., 2020; Widanage et al., 2020; Perera et al., 2022). For example, in an Intel® Xeon® Platinum 8160 high-end workstation with 240GB memory, it takes around 700s to `join` two DFs with 1 billion rows each for `pandas`, whereas traversing each dataframe only takes about 4 s. On the other hand, modern computer hardware offers significant computing power and substantial memory. On-demand elastic cloud computing services allow tasks to be executed on thousands of nodes with a single click. Therefore, we have abundant resources available to create more efficient distributed data engineering solutions.

Hadoop YARN, Dask, and Ray are examples of distributed execution runtimes that can manage thousands of computing resources. Developed mainly by the distributed and cloud computing communities, these engines offer application program interfaces (APIs) that allow users to easily deploy their logic across numerous nodes. In the data engineering community, we have seen several frameworks attempting to leverage these distributed runtimes to develop distributed dataframe (DDF) solutions. Spark SQL RDDs and Datasets was a breakthrough framework on this front, significantly improving the traditional map-reduce paradigm (Zaharia et al., 2012). Dask developed its own take on DDFs, Dask DDF, closely followed by Ray with Ray-Datasets. Modin is the latest attempt to develop scalable DF systems (Petersohn et al., 2020), which is also built on top of Dask and Ray. However, Ray Datasets have limitations, as they currently only support unary operators, and operations like *groupby* take too long to complete. Modin DDFs are restricted to broadcast joins and perform poorly with dataframes of similar sizes. Dask and Spark Datasets struggle with scalability in some operations, especially *groupby*, which may indicate issues with communication implementation. Spark also shows timing anomalies with 8–32 parallelism, which the Spark community needs to investigate. Each framework faces challenges in a pipeline of operators, such as communication overhead, limited support for certain operations, and scalability issues in specific scenarios. Users should consider these factors when selecting a framework based on their specific needs and requirements (Widanage et al., 2020; Perera et al., 2022).

In a previous publication, we developed an alternative to the existing DDFs named *Cylon* (Widanage et al., 2020), which looks at the problem from the HPC point of view. *Cylon* employs bulk synchronous parallel BSP model for DDF operator execution, and works on top of MPI runtimes (OpenMPI, MPICH, IBM Spectrum MPI, etc). Due to superior scalability and HPC descent, we differentiate *Cylon* as a *high performance DDF (HP-DDF)* implementation. Apart from running on BSP, another notable feature in HP-DDFs is the use of an optimized communication library (Perera et al., 2023).

Although Cylon has managed to achieve superior scalability compared to many well-known DDF systems, it is heavily dependent on the MPI ecosystem. The way MPI processes are initiated is closely linked to the specific MPI implementation being used, such as OpenMPI, which uses PMIx. This reliance on MPI poses difficulties for integrating with other distributed computing libraries like Dask and Ray. The strong dependence of Cylon on MPI for process initiation limits its ability to use MPI as a standalone communication library on top of these other libraries. Typically, libraries like Dask and Ray handle the initiation of their worker processes independently, and there is no simple method for the MPI runtime to connect with these pre-existing worker processes.

In this paper, we propose an alternative execution methodology to resolve this limitation. Our objective is to integrate *Cylon* with other execution runtimes without compromising its scalability and performance. It is a bipartite solution: (1) creating a stateful pseudo-BSP environment within the execution runtime resources; (2) using a modularized *communicator* that enables plugging-in optimized communication libraries. We named it *CylonFlow* because the idea carries parallels to workflow management. We demonstrate the robustness of this idea by implementing *Cylon* HP-DDF runtimes on top of Dask (*CylonFlow*-on-Dask) and Ray (*CylonFlow*-on-Ray) that outperform their own DDF implementations. We also confirm that the idea gives comparable or better results than MPI-based *Cylon* DDF on the same hardware. With *CylonFlow*, we have now enabled HP-DDFs from anywhere to personal laptops or exascale supercomputers. As depicted in Figure 1, it consolidates disparate execution models and communities under a single application runtime. To the best of our knowledge, this is the first attempt to adapt high-performance data engineering constructs to distributed computing environments. We believe that the methodology behind *CylonFlow* extends beyond the data engineering domain, and it could be used to execute many HPC applications in distributed computing environments.

2 Distributed computing models and libraries

In order to understand the design and implementation of both *Cylon* and *CylonFlow*, it is important to discuss the existing distributed computing models and prevalent libraries that implement them. A distributed computing model provides an abstract view of how a particular problem can be decomposed and executed from the perspective of a machine. It describes how a distributed application expresses and manages parallelism. *Data parallelism* executes the same computation on different parts (partitions) of data using many compute units. We see this at the instruction level, *single-instruction multiple-data (SIMD)*, as well as in program level *single-program multiple-data (SPMD)*. On the

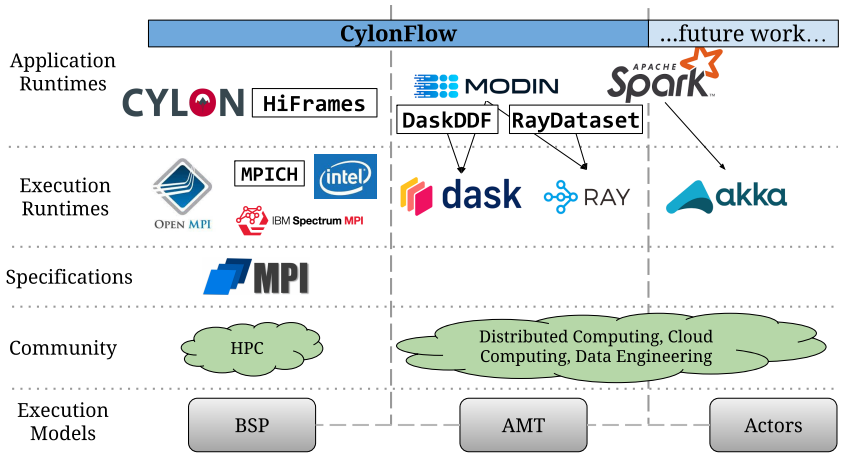

FIGURE 1
Current status Quo and *CylonFlow* contribution.

other hand, *task parallelism* involves executing multiple tasks in parallel over many compute units. This is a form of *multiple-program multiple-data (MPMD)* at the program level.

2.1 Bulk synchronous parallel (BSP)

BSP or Communicating Sequential Processors (CSP) model (Fox et al., 1989; Valiant, 1990) is the most common model that employs SPMD and *data parallelism* over many compute nodes. Message Passing Interface (MPI) is a formal specification of BSP model that has matured over 30+ years. OpenMPI, MPICH, MSMPI, IBM Spectrum MPI, etc. are some notable implementations of this specification. MPI applications display *static parallelism* since most often parallelism needs to be declared at the initiation of the program. From the point of view of the data, this would mean that the data partitions are tightly coupled to the parallelism. At the beginning of the application, data partitions would be allocated to executors/workers. Executors then own data partitions until the end of the application and perform computations on them. When the workers reach a communication operation in the program, they synchronize with each other by passing messages. Many high performance computing (HPC) applications use the BSP model on supercomputing clusters and have shown admirable scalability. However, only a handful of data engineering frameworks have adopted this model, including *Twister2* (Kamburugamuve et al., 2020) and *Cylon*.

2.2 Asynchronous many-tasks

Asynchronous many-tasks (AMT) model relaxes the limitations of BSP by decomposing applications into independent transferable sub-programs (many tasks) with associated inputs (data dependencies). AMT runtimes usually manage a distributed queue that accepts these tasks (*Manager/Scheduler*). A separate group of executors/workers would execute tasks from this queue, thus following MPMD and *task parallelism*. Dependencies between tasks are handled by the scheduling order. This allows the application to set parallelism on-the-fly, and the workers are allowed to scale up or down, leading to *dynamic parallelism*. AMT also enables better resource utilization in multi-tenant/multi-application environments by allowing free workers to pick independent tasks, thereby improving the overall throughput of the system. Furthermore, task parallelism enables task-level fault tolerance where failed tasks can be rerun conveniently. These benefits may have prompted many distributed dataframe runtimes, including Dask DDF and Ray Datasets, to choose AMT as the preferred execution model.

2.3 Actors

Actor model was popularized by *Erlang* (Armstrong, 2010). An actor is a primitive computation which can receive messages from other actors, upon which they can execute a computation, create more actors, send more messages, and determine how to respond to the next message received. Compared to executors and tasks in AMT, actors manage/maintain their own state, and the state may change based on the computation/communication. Messages are sent asynchronously and placed in a *mailbox* until the designated actor consumes them. Akka is a popular actor framework that was used as the foundation for the Apache Spark project. Interestingly, Dask and Ray projects also provide an actor abstraction on top of their distributed execution runtimes mainly aimed at reducing expensive state initializations.

3 Distributed data dataframes

With the exponential growth in dataset sizes, it is fair to conclude that data engineering applications have already exceeded the capabilities of a single workstation node. Modern hardware offers many CPU cores/threads for computation, and the latest cloud infrastructure enables users to spin many such nodes

instantaneously. As a result, there is abundant computing power available at users' disposal, and it is essential that data engineering software make use of it. Furthermore, every AI/ML application requires a pre-processed dataset, and it is no secret that data pre-processing takes significant developer time and effort. Several AI/ML surveys suggest that it could even be more than 60% of total developer time (Anaconda, 2021). For these reasons, using scalable *distributed dataframe (DDF)* runtime could potentially improve the efficiency of data engineering pipelines immensely. Based on our experiments with some widely used DDF systems (Section 5), we believe that the idea of a *high performance scalable DDF runtime* is still a work in progress.

3.1 Dataframes

Let us first define a dataframe. We borrow definitions from the relations terminology proposed by Abiteboul et al. (1995). Similar to SQL tables, DFs contain heterogeneously typed data. These elements originate from a known set of *domain*s, $Dom = \{dom_1, dom_2, ...\}$. For a DF, these *domain*s represent all the data types it supports. A *Schema* of a DF S_M is a tuple (D_M, C_M), where D_M is a vector of M domains and C_M is a vector of M corresponding column labels. Column labels usually belong to *String*/*Object* domain. A *Dataframe (DF)* is a tuple (S_M, A_{NM}, R_N), where S_M is the Schema with M domains, A_{NM} is a 2-D array of entries where actual data is stored, and R_N is a vector of N row labels belonging to some domain. *Length* of the dataframe is N, i.e. the number of rows.

Heterogeneously typed schema clearly distinguishes DFs from multidimensional arrays or tensors. However data along a column is still homogeneous, so many frameworks have adopted a columnar data format that enables vectorized computations on columns. A collection of `numpy NDArrays` would be the simplest form of DF representation. Alternatively, Apache Arrow columnar format (Apache Software Foundation, n.d.) is commonly used by many DF runtimes. Arrow arrays are composed of multiple buffers such as data, validity and offsets for variable-length types (e.g. `string`). As identified in previous literature, many commonly used DF operators are defined over the vertical axis (row-wise) (Petersohn et al., 2020; Perera et al., 2022). Even though columnar representation allows contiguous access along a column, it makes indexing or slicing rows non-trivial. Furthermore, many DF operators are defined on a set of *key columns*, while the rest (i.e. *value columns*) move along with the keys. As a consequence, traditional BLAS (basic linear algebra subprograms) routines cannot be directly used for DF operators.

3.2 DDF system design

The composition of a DF introduces several engineering challenges in designing distributed DF systems. Similar to any distributed/parallel system design, let us first examine the computation and communication aspects broadly.

3.2.1 Computation

Petersohn et al. (2020) recognize that many `Pandas` operators can potentially be implemented by a set of core operators, thereby reducing the burden of implementing a massive DDF API. Correspondingly, in a recent publication, we observed that DF operators follow several generic distribution execution patterns (Perera et al., 2022). The *pattern* governs how these sub-operators are arranged in a directed acyclic graph (DAG). We also identified that a DDF operator consists of three major sub-operators: (1) core local operator; (2) auxiliary local operators; and (3) communication operators. Figure 2 depicts a distributed `join` operation composition, and Figure 3 shows the relationship between the concepts of *Cylon* and Modin. A framework may choose to create tasks (i.e. the definition for a unit of work) for each of these sub-operators. A typical application would be a pipeline of multiple DDF operators.

When using the AMT model, these tasks would be further expanded for each data partition (parallelism). Every task would produce input data for subsequent tasks. This dataflow governs the dependencies between tasks. When there are several operators in a DAG, it is common to see multiple local tasks grouped together. An *execution plan optimizer* may identify such tasks and coalesce them together into a single local task. We see these optimizations in the Apache Spark SQL *Tungsten* optimizer (Apache Spark's, 2020). Previously mentioned in Section 1, data parallelism is natively supported by the BSP model. Since the executors own data partitions until the end of an application, they have the ability to perform all local compute tasks until they reach a communication boundary. As such, coalescing subsequent local tasks are inherently supported by the model itself compared to AMT.

3.2.2 Communication

Implementing DDF operators requires point-to-point (P2P) communication, as well as complex message passing between worker processes. We have identified several such collective communication routines, such as shuffle (all-to-all), scatter, (all)gather, broadcast, (all)reduce, etc, that are essential for DDF operators (Perera et al., 2022). Typically, communication routines are performed on data buffers (ex: MPI, TCP), but the DF composition dictates that these routines be extended on data structures such as DFs, arrays, and scalars. Such data structures may be composed of multiple buffers (Section 1) which could further complicate the implementation. For example, `join` requires a DF to be `shuffled`, and to do this we must `AllToAll` the buffer sizes of all columns (counts). We then shuffle column data based on these counts. In most DF applications, communication operations may take up significant wall time, creating critical bottlenecks. This is evident from Section 5.1, where we evaluate the distribution of communication and computation time over several DF operator patterns. Moreover, developer documentation of Spark SQL, Dask DDF, Ray Datasets, etc, provide special guidelines to reduce `shuffle` routine overheads (Shuffling Performance, n.d.; Welcome to the Ray, n.d.).

While these communication routines can be implemented ingenuously using point-to-point message passing, implementation of specialized algorithms has shown significant performance

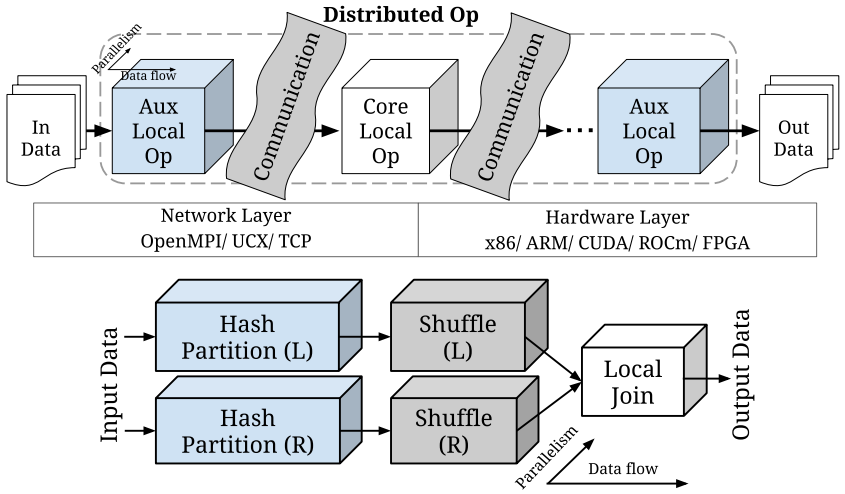

FIGURE 2
Distributed DDF sub-operator composition (Perera et al., 2022) (**Bottom**: Join Operator Example).

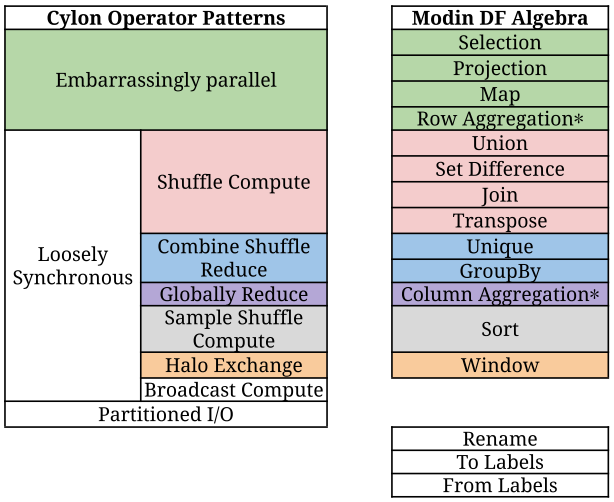

FIGURE 3
Cylon operator patterns and modin DF algebra. *Not categorized in the original Modin paper.

improvements (Bruck et al., 1997; Thakur et al., 2005; Träff et al., 2014). For instance, OpenMPI implements several such algorithms for its collective communications, which can be chosen based on the application. Typically in AMT run-times, communications between tasks are initiated with the help of a *Scheduler*. Another approach is to use a distributed object store or a network file system to share data rather than sending/receiving data explicitly, although this could lead to severe communication overhead.

3.3 DDF systems examined

Let us examine several of the most commonly used DDF systems to understand their distributed execution models and broad design choices. We will then compare these systems with our novel approach described in Section 4.

3.3.1 Dask DDF

Dask DDF is a distributed DF composed of many `Pandas` DFs partitioned along the vertical axis. Operators on Dask DDFs are decomposed into tasks which are then arranged in a DAG (Figure 4 depicts a Join operation). Dask-Distributed Scheduler then executes these tasks on Dask-Workers. This DDF execution is a good example of AMT model. Core local operators are offloaded to `Pandas`. Communication operators (mainly `shuffle`) support point-to-point TCP message passing using *Partd* disk-backed distributed object store.

3.3.2 Ray datasets

Ray Datasets is a DDF-like API composed of Apache Arrow tables or Python objects stored in the distributed object store. Similar to Dask, distributed operators (*Transforms*) follow the AMT model. Interestingly, they support a *task strategy* as well as an *actor strategy*. The latter is recommended for expensive state initialization (e.g. for GPU-based tasks) to be cached. As per communication, a map-reduce style `shuffle` is used which maps tasks to partition blocks by value and then reduces tasks to merge co-partitioned blocks together. Essentially, Ray communication operators are backed by the object store. For larger data, the documentation suggests using a *push-based shuffle*.

3.3.3 Apache spark dataset

It is fair to say that Apache Spark is the most popular actor-based data engineering framework available today, and it has attracted a large developer community since its initial publication, *Resilient Distributed Datasets (RDDs)* (Zaharia et al., 2012). *PySpark Dataset* is a DDF-like API, and recently a `Pandas`-like DDF named *Pandas on Spark* was also released. Similar to AMT,

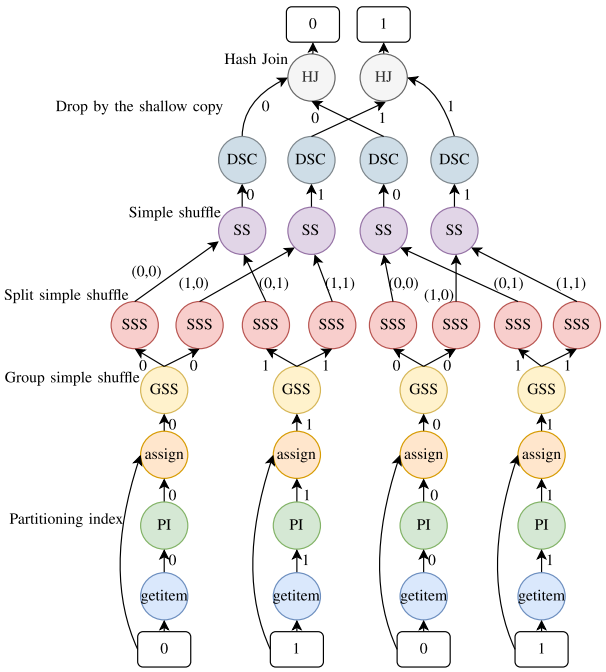

FIGURE 4
Dask DDF `join` (2 partitions).

Spark decomposes operators into a collection of map-reduce tasks, after which a manager process schedules these tasks in executors allocated to the application. It uses *Akka-Actors* to manage the driver (i.e. the process that submits applications), the manager, and executors. Essentially, Spark implements AMT using the actor model for map-reduce tasks. All these processes run on a Java Virtual Machine (JVM), and could face significant (de)serialization overheads when transferring data to and from Python. As an optimization, the latest versions of PySpark enable Apache Arrow columnar data format.

3.3.4 Modin DDF

Modin (Petersohn et al., 2020) is the latest addition to the DDF domain. It introduces the concept of *DF algebra* (Figure 3), where a DDF operator can be implemented as a combination of core operators. It executes on Dask and Ray backends, which also provide the communication means for DDF. Modin distinguishes itself by attempting to mirror the `Pandas` API and follow eager execution.

4 *Cylon* and *CylonFlow*: high performance DDFs in Dask and Ray

Through our research, we have encountered several performance limitations while using the aforementioned DDF systems for large datasets. As discussed in Section 5, many of these DDFs show limited scalability, and we believe the limitations of the AMT model could be a major contributor to that. A centralized scheduler might create a scheduling bottleneck. Additionally, the lack of a dedicated optimized communication mechanism further compounds the issues. It is fair to assume that the optimization of communication routines is orthogonal to designing distributed computing libraries such as Dask/Ray, and re-purposing generic distributed data-sharing mechanisms for complex communication routines may lead to suboptimal performance when used in DDF implementations.

In a recent publication we proposed an alternative approach for DDFs that uses BSP execution model, which we named *Cylon* (Widanage et al., 2020). It is built on top of MPI and uses MPI collective communication routines for DDF operator implementations. MPI libraries (OpenMPI, MPICH, IBM-Spectrum) have matured over the past few decades to employ various optimized distributed communication algorithms, and *Cylon* benefits heavily from these improvements. It also profits from data parallelism and implicit coalescing of local tasks by employing the BSP model. Experiments show commendable scalability with *Cylon*, fittingly differentiating it as a *high performance DDF (HP-DDF)*. Even though high performance DDFs seem encouraging, having to depend on an MPI environment introduces several constraints. MPI process bootstrapping is tightly coupled to the underlying MPI implementation, e.g. OpenMPI employs PMIx. As a result, it is not possible to use MPI as a separate communication library on top of distributed computing libraries such as Dask/Ray. Usually these libraries would bootstrap their worker processes by themselves. There is no straightforward way for the MPI runtime to bind to these workers.

We strongly believe it is worthwhile to expand on the HP-DDF concept beyond MPI-like environments. Current advancements in technology and the high demand for efficient data engineering solutions encourage this position. Our main motivation for this paper is to develop an execution environment where we could strike a balance between the scalability of BSP and the flexibility of AMT. Dask and Ray have proven track records as distributed computing libraries. So rather than building a new system from scratch, we focused on bridging the gap between BSP and these libraries. We propose a two-pronged solution to this problem. First, creating a stateful pseudo-BSP execution environment using the computing resources of the execution runtime. This lays the foundation for HP-DDF execution. The second step is using a modularized *communicator* abstraction (i.e. interface that defines communication routines) that enables pluging-in optimized communication libraries. We named this project *CylonFlow*, as it embraces the idea of *managing a workflow*.

4.1 Stateful pseudo-BSP execution environment

Within this pseudo-BSP environment, executors initialize an optimized communication library and attach it to the state of the executor. The state would keep this communication context alive for the duration of an *CylonFlow* application. This allows *CylonFlow* runtime to reuse the communication context without having to reinitialize it, which could be an expensive exercise for larger parallelisms. Once the environment is set up, the executors implicitly coalesce and carry out local operations until a

communication boundary is met. The state can also be used to share data between *CylonFlow* applications as discussed in Section 4.3.

This proposition of creating stateful objects matches perfectly with the actor model. Thus we leveraged the actor APIs available in Dask and Ray to implement *CylonFlow*-on-Dask and *CylonFlow*-on-Ray (Figure 5). An actor is a reference to a designated object (`CylonActor` class) residing in a remote worker. The driver/user code would call methods on this remote object, and during the execution of this call, *CylonFlow* runtime passes the communication context as an argument. Inside these methods, users can now express their data engineering applications using *Cylon* DDFs.

This approach enables partitioning of the cluster resources and scheduling *independent applications*. It would be a much more coarsely grained work separation, but we believe the abundance of computing units and storage in modern processor hardware, and their availability through cloud computing, could still sustain it. To the best of our knowledge, this is the first time actors are being used together with a dedicated communication library to develop HP-DDF runtimes. This approach is semantically different from actors in Apache Spark, where they would still be executing *independent tasks* in an AMT manner. Neither should it be confused with other orthogonal projects like *Dask-MPI*, which is used to deploy a Dask cluster easily from within an existing MPI environment.

Upon the initialization of the application, *CylonFlow* sends *Cylon* Actor definition (a class) to a partition of workers in the cluster based on the required parallelism. Workers then initialize these as an actor instance (remote object). At the same time, the actor instances initialize communication channels between each other, which is the entry point for creating *Cylon* DDFs (i.e. `Cylon_env`). Instantiating an `Cylon_env` could be an expensive operation, especially with large parallelism, as it opens up P2P communication channels between the remote objects.

The *Cylon* actor class exposes three main endpoints.

1) `start_executable`: Allows users to submit an executable class that would be instantiated inside the actor instance.
2) `execute_Cylon`: Execute functions of the executable that accepts an `Cylon_env` object and produces a Future.
3) `run_Cylon`: Execute a lambda function that accepts an `Cylon_env` object and produces a Future.

The following is an example code which creates two *Cylon* DFs using Parquet files and performs a join (`merge`) on them.

```
def foo(env:CylonEnv=None):
    df1 = read_parquet(..., env=env)
    df2 = read_parquet(..., env=env)
    write_parquet(df1.merge(df2, ..., env=env), ..., env=env)

init()
wait(CylonExecutor(parallelism=4).run_Cylon(foo))
```

4.1.1 Spawning Dask actors

Dask does not have a separate API endpoint to reserve a set of workers for an application. Consequentially, *CylonFlow* uses the `Distributed.Client` API to collect a list of all available workers. It then uses the `Client.map` API endpoint with a chosen list of workers (based on the parallelism) to spawn the actor remote objects. Dask actor remote objects open up a direct communication channel to the driver, which they would use to transfer the results back. This avoids an extra network hop through the scheduler and achieves lower latency.

4.1.2 Spawning Ray actors

Ray provides a *Placement Groups* API that enables reserving groups of resources across multiple nodes (known as gang-scheduling). *CylonFlow* creates a placement group with the required parallelism and submits the *Cylon* Actor definition to it. In Ray documentation (Welcome to the Ray, n.d.), communicating actors such as this are called out-of-band communication.

4.2 Modularized communicator

Once the pseudo-BSP environment is set up, *Cylon* HP-DDF communication routines can pass messages amongst the executors. However, we would still not be able to reuse the MPI communications due to the limitations we discussed previously. To address this, we had to look for alternative communication libraries which could allow us to implement *Cylon* communication routines outside of MPI without compromising its scalability and performance. We achieved this by modularizing *Cylon* communicator interface and adding abstract implementations of DDF communication routines as discussed in Section 3. This allowed us to conveniently integrate Gloo and UCX/UCC libraries as alternatives to MPI. Communicator performance experiments in Section 5.2 demonstrate that these libraries perform as good as if not better than MPI on the same hardware.

4.2.1 Gloo

Gloo is a collective communications library managed by Meta Inc. incubator (facebookincubator/gloo, n.d.) predominantly aimed at machine learning applications. PyTorch uses this for distributed all-reduce operations. It currently supports TCP, UV, and ibverbs transports. Gloo communication runtime can be initialized using an MPI Communicator or an NFS/Redis key-value store (P2P message passing is not affected). Within MPI environments *Cylon* uses the former, but for the purposes of *CylonFlow* it uses the latter. As an incubator project, Gloo lacks a comprehensive algorithm implementation, yet our experiments confirmed that it scales admirably. We have extended the Gloo project to suit *Cylon* communication interface.

4.2.2 Unified communication X (UCX)

UCX (Shamis et al., 2015) is a collection of libraries and interfaces that provides an efficient and convenient way to construct widely used HPC protocols on high-speed networks, including MPI tag matching, Remote Memory Access (RMA) operations, etc. Unlike MPI runtimes, UCX communication workers are not bound to a process bootstrapping mechanism. As such, it is being used by many frameworks, including Apache Spark and RAPIDS (Dask-CuDF). It provides primitive P2P communication operations. Unified Collective Communication (UCC) is a collective communication operation API built on top

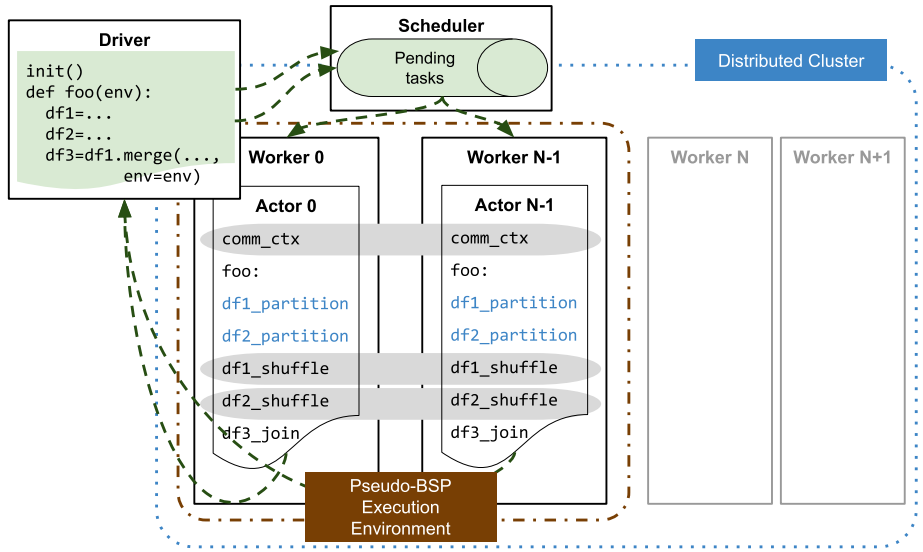

FIGURE 5
CylonFlow using actors for HP-DFF.

of UCX which is still being developed. Similar to MPI, UCC implements multiple communication algorithms for collective communications. Based on our experiments, UCX + UCC performance is on par with or better than OpenMPI. *CylonFlow* would use Redis key-value store to instantiate communication channels between *Cylon* actors.

4.3 Sharing results with downstream applications

As discussed in Section 4.1, this approach allows partitioning of the cluster resources and scheduling of individual applications. These applications may contain data dependencies, for example, multiple data preprocessing applications feeding data into a distributed deep learning application. However this typically produces DDFs, and it would not be practical to collect intermediate results to the driver program. We propose an *CylonFlow* data store (i.e. `Cylon_store`) abstraction to retain these results. In the following example, `data_df` and `aux_data_df` will be executed in parallel on two resource partitions, and `main` function would continue to execute the deep learning model.

```
def process_aux_data(env:CylonEnv=None, store:CylonStore=None):
    aux_data_df = ...
    store.put("aux_data", aux_data_df, env=env)

def main(env:CylonEnv=None, store:CylonStore=None):
    data_df = ...
    aux_data_df = store.get("aux_data", timeout=..., env=env)
    df = data_df.merge(aux_data_df, ...)

    x_train = torch.from_numpy(df.to_numpy()).to(device)
    model = Model(...)
    ...

init()
CylonExecutor(parallelism=4).run_Cylon(process_aux_data)
wait(CylonExecutor(parallelism=4).run_Cylon(main))
```

`Cylon_store` could be backed by an NFS or distributed object store (ex: Ray's Object Store). This feature is currently being developed under *CylonFlow*, and is mentioned here only for completeness. In instances where applications choose different parallelism values, the store object may be required to carry out a `repartition` routine.

4.4 *CylonFlow* features

The proposed actor-based solution *CylonFlow* provides several benefits compared to traditional MPI-like (BSP) environments as well as distributed computing environments.

4.4.1 Scalability

Experiments show that *CylonFlow*-on-Dask and *CylonFlow*-on-Ray offer better operator scalability on the same hardware compared to Dask DDF and Ray Datasets, which employ AMT model (Section 5). It also surpasses Spark Datasets, which uses a conventional actor model. *CylonFlow* provides data engineering users a high performance and scalable DF alternative to their existing applications with minimum changes to execution environments.

4.4.2 Application-level parallelism

Partitioning resources within a distributed computing cluster enables parallel scheduling of multiple *CylonFlow* tasks. These would have much more coarsely grained parallelism compared to a typical task composed of a DDF operator. A future improvement we are planning to introduce is an execution plan optimizer that splits the DAG of a DF application into separate sub-applications (e.g. coalesce an entire branch). These sub-applications can then be individually scheduled in the cluster. Outputs (which are already

partitioned) could be stored in a distributed object store to be used by subsequent sub-programs. We are potentially looking at large binary outputs which can be readily stored as objects rather than using the object store for internal communication routines. This *application-level parallelism* could also enable multi-tenant job submission.

4.4.3 Interactive programming environment

Petersohn et al. (2020) observed that an interactive programming environment is key for exploratory data analytics. R and Python being interpreted languages suits very well with this experience. One major drawback of *Cylon* is that it cannot run distributed computations on a notebook (e.g. Jupyter). *CylonFlow* readily resolves this problem by enabling users to acquire a local/remote resource (managed by Dask/Ray) and submit *Cylon* programs to it interactively.

4.4.4 High performance everywhere

The concept of *CylonFlow* is not limited to distributed computing libraries, but also extends to larger computing environments such as supercomputers. We are currently developing an *CylonFlow* extension for leadership class supercomputers. Our end goal is to enable high performance scalable data engineering everywhere, from a personal laptop to exascale supercomputers.

5 Experiments

The following experiments were carried out on a 15-node Intel® Xeon® Platinum 8160 cluster. Each node is comprised of 48 hardware cores on two sockets, 255GB RAM, SSD storage, and are connected via Infiniband with 40 Gbps bandwidth. The software used were Python v3.8; Pandas v1.4; *Cylon* (GCC v9.4, OpenMPI v4.1, and Apache Arrow v5.0); Dask v2022.8; Ray v1.12; Modin v0.13; Apache Spark v3.3; SQLite3. SQLite is used to compare a join and sort operations with 1 M rows table in a single core in Table 1. The target is to show how Cylon performs comparable results within a single database. Performing distributed operations with SQLite poses additional overheads. We checked two cases: firstly, a single table from a database is accessed by multiple workers to perform sort and join operations. We had to ensure data consistency by implementing lock operations for each worker which added additional latencies. Secondly, multiple tables are also used from different databases to perform distributed sort and join operations. In that case, additional overhead is added to create a database connection with each worker along with joining multiple tables. We moved our focus on distributed data frames with underlying execution with CylonFlow on Dask and Ray clusters. To do that, uniformly random distributed data was used with two `int64` columns, 10^9 rows (∼16 GB) in column-major format (Fortran order). Data uses a cardinality (i.e. % of unique keys in the data) of 90%, which constitutes a worst-case scenario for key-based operators. The scripts to run these experiments are available in Github (CylonData, n.d.). Out of the operator patterns discussed in our previous work (Perera et al., 2022), we have only chosen `join`, `groupby`, and `sort` operators. These cover some of the most complex routines from the point of view of DDF operator design. Only operator timings have been considered (without data loading time). Input data will either be loaded from the driver to the workers or loaded as Parquet files from the workers themselves (Dask and Apache Spark discourage the former). Data is then repartitioned based on parallelism and cached.

We admit that in real applications, operator performance alone may not portray a comprehensive idea of the overall performance of a data engineering framework. However we believe it is reasonably sufficient for the purpose of proposing an alternative approach for execution. Dask DDFs, Ray Datasets, Spark Datasets, and Modin DDFs are only used here as baselines. We tried our best to refer to publicly available documentation, user guides and forums while carrying out these tests to get the optimal configurations.

5.1 Communication and computation

Out of the three operators considered, `join`s have the most communication overhead, as it is a binary operator (two input DFs). We investigated how the communication and computation

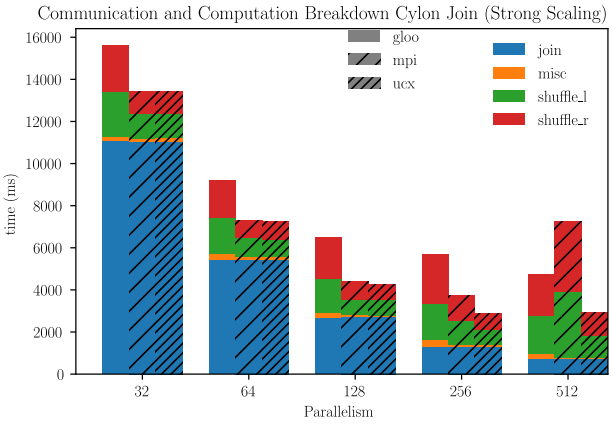

FIGURE 6
Communication and computation breakdown of *Cylon* join operation (1B rows).

TABLE 1 Experiments on UVA.Rivanna with SQLite and CylonFlow with join and sort operations in a single node.

ID	Experiment type	AT (SQLite)	AT (CylonFlow)	#Rows	#Nodes	Dataset size
A	Join operation	9.1514	9.0137	[1M]	1	1.5GB
B	Sort operation	18.4042	17.9173	[1M]	1	1.5GB

We calculate average time (AT) in seconds for SQLite and CylonFlow versions.

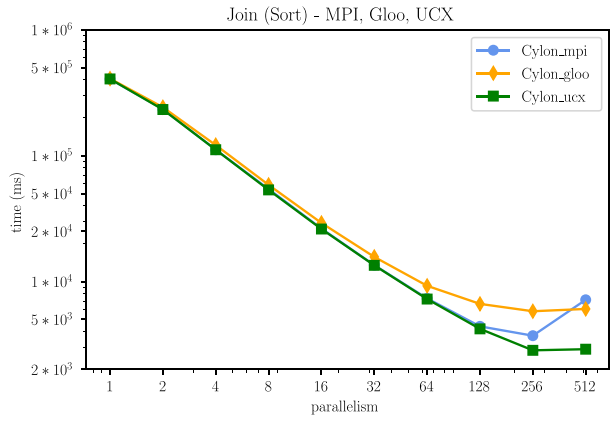

FIGURE 7
OpenMPI, Gloo, vs. UCX/UCC (1B rows, Log-Log) - Processes spawned by `mpirun`.

time varies based on the parallelism (Figure 6). Even at the smallest parallelism (32), there is a significant communication overhead (Gloo 27%, MPI 17%, UCX 17%), and as the parallelism increases, it dominates the wall time (Gloo 76%, MPI 86%, UCX 69%). Unfortunately, we did not have enough expertise in the Spark, Dask, or Ray DDF code base to run a similar micro-benchmark. But even while using libraries specialized for message passing, *Cylon* encounters significant communication overhead.

5.2 OpenMPI vs. Gloo vs. UCX/UCC

In this experiment, we test the scalability of *Cylon* communicator implementations (for `join` operation). As discussed in Section 4, we would not be able to use MPI implementations inside distributed computing libraries. Figure 7 confirms that our alternative choices of Gloo and UCX/UCC show equivalent performance and scalability. In fact, UCX/UCC outperforms OpenMPI in higher parallelisms. We have seen this trend in other operator benchmarks as well.

5.3 *CylonFlow*-on-Dask and *CylonFlow*-on-Ray

In this experiment we showcase the performance on the proposed HP-DDF approach for distributed computing libraries (Dask and Ray) against their own DDF implementations (Dask DDF and Ray Datasets). Unfortunately we encountered several challenges with Ray Datasets. It only supports unary operators currently, therefore we could not test `join`s. Moreover, Ray `groupby` did not complete within 3 h, and `sort` was showing presentable results. We have also included Apache Spark, since the proposed approach leverages actor model. We enabled Apache Arrow in PySpark feature because it would be more comparable. We also added Modin DDFs to the mix. Unfortunately, it only supports `broadcast joins` which performs poorly on two similar sized DFs. We could only get Modin to run on Ray backend with our datasets, and it would default to `Pandas` for `sort`. `Pandas` serial performance is also added as a baseline comparison.

Looking at the 1 billion rows strong scaling timings in Figure 8, we observe that *Cylon*, *Cylon*-on-Dask, and *Cylon*-on-Ray are nearly indistinguishable (using Gloo communication). Thus it is evident that the proposed *CylonFlow* actor approach on top of Dask/Ray does not add any unexpected overheads to vanilla *Cylon* HP-DDF performance. Dask and Spark Datasets show commendable scalability for `join` and `sort`, however former `groupby` displays very limited scalability. We investigated Dask and Spark further by performing a 100 million row test case (bottom row of Figure 8) which constitutes a communication-bound operation. Under these circumstances, both systems diverge significantly at higher parallelisms, indicating limitations in their communication implementations. We also noticed a consistent anomaly in Spark timings for 8–32 parallelism. We hope to further investigate this with the help of the Spark community. *CylonFlow* also shows decreasing scalability with much smoother gradients and displays better communication performance. These findings reinforce our suggestion to use a pseudo-BSP environment that employs a modular communicator. In fact, our preliminary tests suggested that using UCX/UCC communicator could potentially improve the performance further in the same setup (Section 5.2).

At 512 parallelism, on average *CylonFlow* performs 142×, 123×, and 118× better than `Pandas` serial performance for `join`, `groupby`, and `sort` respectively. We also observe that the serial performance of *CylonFlow* outperforms others consistently, which could be directly related to *Cylon*'s C++ implementation and the use of Apache Arrow format. At every parallelism, *CylonFlow* distributed performance is $2 - 4\times$ higher than Dask/Spark consistently. These results confirm the efficacy of the proposed approach.

5.4 Pipeline of operators

We also tested the following pipeline on *CylonFlow*, Dask DDF, and Spark Datasets, `join` → `groupby` → `sort` → `add_scalar`. As depicted in Figure 9, the gains of *CylonFlow* become more pronounced in composite use cases. Average speed-up over Dask DDFs ranges from $10 - 24\times$, while for Spark Datasets it is $3 - 5\times$. As mentioned in Section 4, *Cylon* execution coalesces all local operators that are in-between communication routines in the pipeline, and we believe this is a major reason for this gain.

6 Limitations and future work

From our findings in Section 4, the idea of using BSP execution environments is a very common use case in HPC and supercomputing clusters, and the *CylonFlow* concept readily fits these environments. We are currently working with Radical-Cybertools and Parsl teams to extend *CylonFlow* to leadership class supercomputers based on workflow management software stack. In

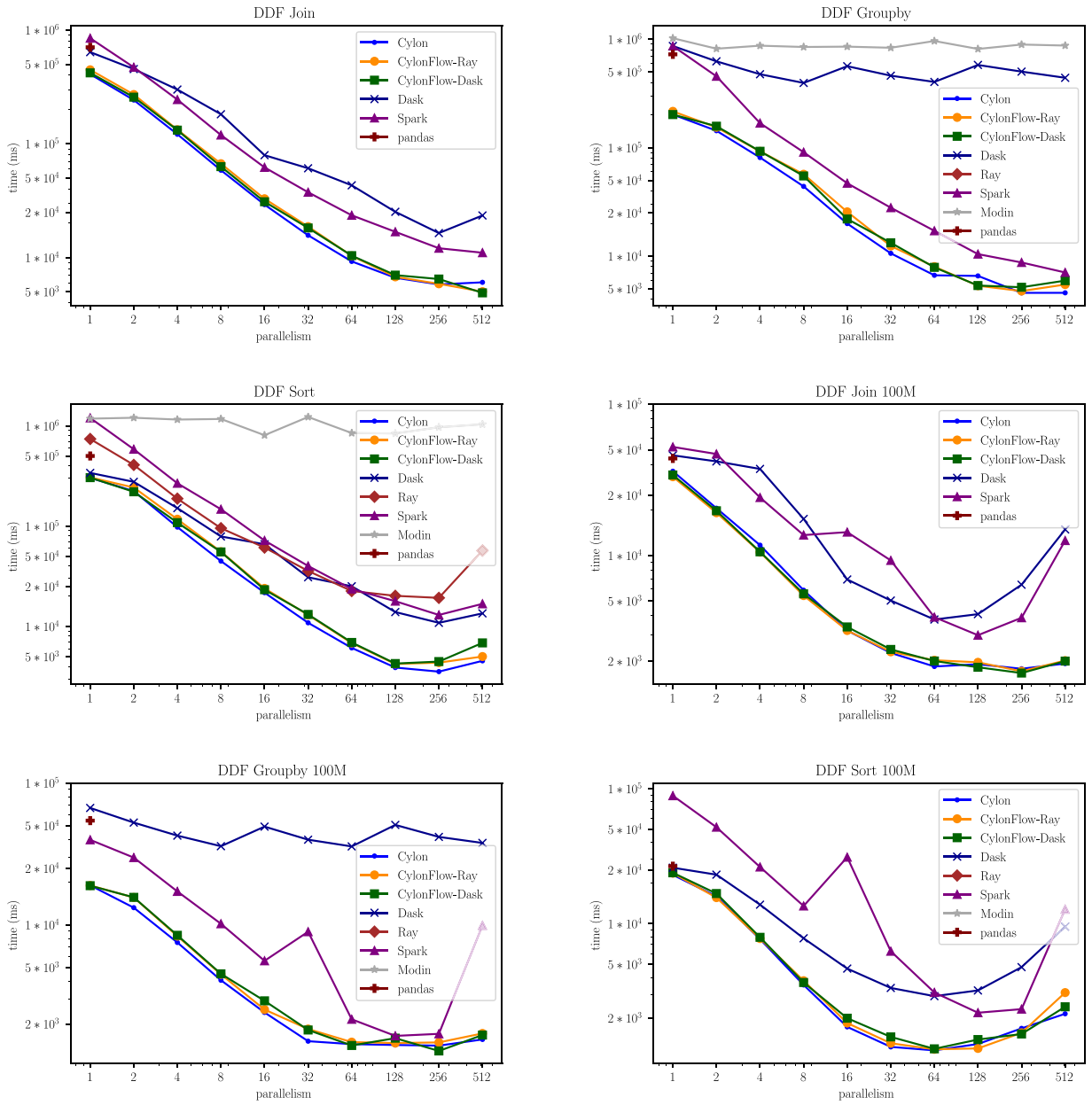

FIGURE 8
Strong scaling of DDF operators (Log-Log), **(Top)** 1B rows, **(Bottom)** 100M rows.

addition, we plan to extend *CylonFlow* on top of pure actor libraries such as Akka. This would enable *Cylon*'s native performance on the JVM using Java Native Interface (JNI). We are currently adding these JNI bindings to *Cylon* and *CylonFlow*.

In Section 5 we saw significant time being spent on communication. In modern CPU hardware, we can perform computation while waiting on communication results. Since an operator consists of sub-operators arranged in a DAG, we can exploit *pipeline parallelism* by overlapping communication and computation. Furthermore, we can also change the granularity of a computation such that it fits into CPU caches. We have made some preliminary investigations on these ideas, and we were able to see significant performance improvements for *Cylon*. Section 4 proposed an *CylonFlow* data store that allows sharing data with downstream applications. This work is still under active development.

Providing fault tolerance in an MPI-like environment is quite challenging, as it operates under the assumption that the communication channels are alive throughout the application. This means providing communication-level fault tolerance would be complicated. However, we are planning to add a checkpointing mechanism that would allow a much coarser-level fault tolerance. Load imbalance (especially with skewed datasets) could starve some processes and might reduce the overall throughput. To

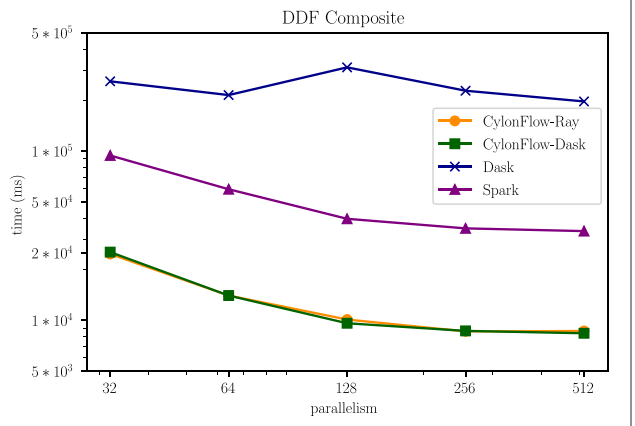

FIGURE 9
Pipeline of operators (1B rows, Log-Log).

avoid such scenarios, we are working on a sample-based repartitioning mechanism.

Dataframe features discussed in this paper closely relate to the relational algebra operations in database management systems (DBMS). Therefore, comparing Cylon and other dataframe abstractions' performances with DBMSs would interest the scientific computing community. We would be taking this up in a future publication.

7 Related work

Initially, traditional database management systems embraced distributed query processing. This process involves coordinating the retrieval and aggregation of data from various distributed sources. It demands consistency and concurrent control, which can add complexity and overhead (Ceri and Pelagatti, 1983). Pandas offers numerous direct advantages, including concurrent access, data persistence, integrity, and optimized querying. But to maintain these metrics in any relational database(SQLite), additional overheads are added to execution time on distributed operations. Although, in single-node operation, SQLite has a similar performance as CylonFlow (Table 1), multiple-node operations have significant complexity in creating connections and performing join/sort operations to create the global tables. Additionally, pandas provides a scripting-based programming interface facilitating integration with other systems like data visualization, machine learning, and web applications (McKinney, 2022). Our focus lies in implementing a distributed data engineering framework that inherits the capabilities of pandas and arrow-based columnar data structures, enabling the processing of big data through bulk synchronized parallel patterns. In a previous publication, we proposed a formal framework for designing and developing high-performance data engineering frameworks that includes data structures, architectures, and program models (Kamburugamuve et al., 2021). Kamburugamuve et al. (2020) proposed a similar big data toolkit named *Twister2*, which is based on Java. There the authors observed that using a BSP-like environment for data processing improves scalability, and they also introduced a DF-like API in Java named *TSets*. However, *Cylon* being developed in C++ enables native performance of hardware and provides a more robust integration to Python and R. Being an extension built in Python, *CylonFlow* still manages to achieve the same performance as *Cylon*.

In parallel to *Cylon*, Totoni et al. (2017) also suggested a similar HP-DDF runtime named *HiFrames*. They primarily attempt to compile native MPI code for DDF operators using `numba`. While there are several architectural similarities between *HiFrames* and *Cylon*, the latter is the only open-source HP-DDF available at the moment. The former is still bound to MPI, hence it would be impractical to use it in distributed computing libraries like Dask/Ray.

Horovod utilizes Ray-actors that use Gloo communication for data parallel deep learning in its *Horovod-on-Ray* project (Horovod documentation, n.d.). From the outset, this has many similarities to *CylonFlow*-on-Ray, but the API only supports communications on tensors. *Cylon*/*CylonFlow* is a more generic approach that could support both DFs and tensors. In fact, these could be complementary frameworks, where data preprocessing and deep learning are integrated together in a single pipeline.

In addition to the DDF runtimes we discussed in this paper, we would also like to recognize some exciting new projects. Velox is a C++ vectorized database acceleration library managed by the Meta Inc. incubator (Pedreira et al., 2022). Currently it does not provide a DF abstraction, but still offers most of the operators shown in Figure 3. Photon is another C++ based vectorized query engine developed by Databricks (Behm et al., 2022) that enables native performance to the Apache Spark ecosystem. Unfortunately, it has yet to be released to the open source community. Substrait is another interesting model that attempts to produce an independent description of data compute operations (Substrait-io/substrait, n.d.).

8 Conclusion

Scalable dataframe systems are vital for modern data engineering applications, but despite this many systems available today could be improved to meet the scalability expectations. In this paper, the we present an alternative approach for scalable dataframes, *CylonFlow*, which attempts to bring high performance computing into distributed computing runtimes. The proposed stateful pseudo-BSP environment and modularized communicator enable state-of-the-art scalability and performance on Dask and Ray environments, thereby *supercharging them*. *CylonFlow* is compared against Dask and Ray's own dataframe systems as well as Apache Spark, Modin, and `Pandas`. Using *Cylon* HP-DDF C++ backend and Apache Arrow format give *CylonFlow* superior sequential performance to the competition. Due to the modular communicator in *CylonFlow*, it is possible to swapp underlaying distributed communication libraries such as swapping Gloo and UCX/UCC for DDF communications, which enables scalable distributed performance on Dask/Ray environments. Hence, *CylonFlow* creates a ubiquitous data engineering ecosystem that unifies both HPC and distributed computing communities.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

NP: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. AS: Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Writing – original draft, Writing – review & editing, Conceptualization. KS: Data curation, Methodology, Validation, Writing – review & editing. AF: Data curation, Validation, Writing – review & editing. SK: Formal analysis, Supervision, Writing – review & editing. TK: Formal analysis, Investigation, Writing – review & editing. CW: Formal analysis, Investigation, Writing – review & editing. MS: Formal analysis, Investigation, Software, Writing – review & editing. TZ: Validation, Writing – review & editing. VA: Formal analysis, Writing – review & editing. GL: Formal analysis, Funding acquisition, Investigation, Project administration, Resources, Supervision, Writing – review & editing. GF: Formal analysis, Funding acquisition, Investigation, Project administration, Resources, Supervision, Writing – review & editing, Conceptualization, Methodology, Visualization, Writing – original draft.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article.

Acknowledgments

We gratefully acknowledge the support of NSF grants 2210266 (CINES) and DE-SC0023452: FAIR Surrogate Benchmarks Supporting AI and Simulation Research.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases, Volume 8*. Reading, MA: Addison-Wesley Reading.

Anaconda (2021). *State of Data Science 2020-anaconda.com*. Available online at: https://www.anaconda.com/state-of-data-science-2020 (accessed February 27, 2022).

Apache Software Foundation (n.d.). *Arrow Columnar Format x2014; Apache Arrow v9.0.0 – arrow.apache.org*. Available online at: https://arrow.apache.org/docs/format/Columnar.html (accessed April 23, 2022).

Apache Spark's (2020). *Catalyst and tungsten: Apache spark's speeding engine*. Available online at: https://www.linkedin.com/pulse/catalyst-tungsten-apache-sparks-speeding-engine-deepak-rajak (accessed May 14, 2022).

Armstrong, J. (2010). erlang. *Commun. ACM* 53, 68–75. doi: 10.1145/1810891.1810910

Behm, A., Palkar, S., Agarwal, U., Armstrong, T., Cashman, D., Dave, A., et al. (2022). "Photon: a fast query engine for lakehouse systems," in *Proceedings of the 2022 International Conference on Management of Data* (New York, NY: ACM), 2326–2339. doi: 10.1145/3514221.3526054

Bruck, J., Ho, C.-T., Kipnis, S., Upfal, E., and Weathersby, D. (1997). Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst.* 8, 1143–1156. doi: 10.1109/71.642949

Ceri, S., and Pelagatti, G. (1983). Correctness of query execution strategies in distributed databases. *ACM Trans. Database Syst.* 8, 577–607. doi: 10.1145/319996.320009

CylonData (n.d.). *Cylonflow experiments repository*. Available online at: https://github.com/cylondata/cylon_experiments (accessed June 1, 2024).

facebookincubator/gloo (n.d.). *facebookincubator/gloo: Collective communications library with various primitives for multi-machine training*. Available online at: https://github.com/facebookincubator/gloo (accessed June 5, 2022).

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D., et al. (1989). Solving problems on concurrent processors vol. 1: General techniques and regular problems. *Comput. Phys.* 3, 83–84. doi: 10.1063/1.4822815

Horovod documentation (n.d.). *Horovod on ray – horovod documentation*. Available online at: https://horovod.readthedocs.io/en/stable/ray_include.html (accessed May 4, 2022).

Kamburugamuve, S., Govindarajan, K., Wickramasinghe, P., Abeykoon, V., and Fox, G. (2020). Twister2: design of a big data toolkit. *Concurr. Comput. Pract. Exp.* 32:e5189. doi: 10.1002/cpe.5189

Kamburugamuve, S., Widanage, C., Perera, N., Abeykoon, V., Uyar, A., Kanewala, T. A., et al. (2021). "Hptmt: operator-based architecture for scalable high-performance data-intensive frameworks," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* (IEEE), 228–239. doi: 10.1109/CLOUD53861.2021.00036

McKinney, W. (2022). *Python for Data Analysis*. Sebastopol, CA: O'Reilly Media, Inc.

Pedreira, P., Erling, O., Basmanova, M., Wilfong, K., Sakka, L., Pai, K., et al. (2022). Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 3372–3384. doi: 10.14778/3554821.3554829

Perera, N., Kamburugamuve, S., Widanage, C., Abeykoon, V., Uyar, A., Shan, K., et al. (2022). High performance dataframes from parallel processing patterns. *arXiv* [Preprint]. arXiv:2209.06146. doi: 10.48550/arXiv:2209.06146

Perera, N., Sarker, A. K., Staylor, M., von Laszewski, G., Shan, K., Kamburugamuve, S., et al. (2023). In-depth analysis on parallel processing patterns for high-performance dataframes. *Future Gener. Comput. Syst.* 149, 250–264. doi: 10.1016/j.future.2023.07.007

Petersohn, D., Macke, S., Xin, D., Ma, W., Lee, D., Mo, X., et al. (2020). Towards scalable dataframe systems. *arXiv* [Preprint]. arXiv:2001.00888. doi: 10.48550/arXiv.2001.00888

PyPI (n.d.). *Pypi download stats: Pandas*. Available online at: https://pypistats.org/packages/pandas (accessed April 10, 2022).

Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., et al. (2015). "UCX: an open source framework for HPC network Apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects* (Santa Clara, CA: IEEE), 40–43. doi: 10.1109/HOTI.2015.13

Shuffling Performance (n.d.). *Shuffling for group by and join – Dask documentation*. Available online at: https://docs.dask.org/en/stable/dataframe-groupby.html (accessed May 14, 2022).

Substrait-io/Substrait (n.d.). *Substrait-io/substrait: a cross platform way to express data transformation, relational algebra, standardized record expression and plans*. Available online at: https://github.com/substrait-io/substrait (accessed October 22, 2023).

Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.* 19, 49–66. doi: 10.1177/1094342005051521

Totoni, E., Hassan, W. U., Anderson, T. A., and Shpeisman, T. (2017). Hiframes: high performance data frames in a scripting language. *arXiv* [Preprint]. arXiv:1704.02341. doi: 10.48550/arXiv:1704.02341

Träff, J. L., Rougier, A., and Hunold, S. (2014). "Implementing a classic: zero-copy all-to-all communication with mpi datatypes," in *Proceedings of the 28th ACM international conference on Supercomputing* (IEEE), 135–144.

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM* 33, 103–111. doi: 10.1145/79173.79181

Welcome to the Ray (n.d.). *Welcome to the ray documentation – ray 2.0.0*. Available online at: https://docs.ray.io/en/latest/index.html (accessed May 4, 2022).

Widanage, C., Perera, N., Abeykoon, V., Kamburugamuve, S., Kanewala, T. A., Maithree, H., et al. (2020). "High performance data engineering everywhere," in *2020 IEEE International Conference on Smart Data Services (SMDS)* (Beijing: IEEE), 122–132. doi: 10.1109/SMDS49396.2020.00022

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., et al. (2012). "Resilient distributed datasets: a {Fault-Tolerant} abstraction for {In-Memory} cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (IEEE), 15–28.