Future Generation Computer Systems 149 (2023) 250–264

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

In-depth analysis on parallel processing patterns for high-performance Dataframes

Niranda Perera [a,*], Arup Kumar Sarker [b,c], Mills Staylor [b], Gregor von Laszewski [c], Kaiying Shan [b], Supun Kamburugamuve [a], Chathura Widanage [a], Vibhatha Abeykoon [a], Thejaka Amila Kanewela [a], Geoffrey Fox [b,c]

[a] Indiana University Alumni, Bloomington, IN 47405, USA
[b] University of Virginia, Charlottesville, VA 22904, USA
[c] Biocomplexity Institute and Initiative, University of Virginia, Charlottesville, VA 22904, USA

ARTICLE INFO

Article history:
Received 13 March 2023
Received in revised form 27 May 2023
Accepted 2 July 2023
Available online 13 July 2023

Keywords:
Dataframes
High-performance computing
Data engineering
Relational algebra
MPI
Distributed Memory Parallel

ABSTRACT

The Data Science domain has expanded monumentally in both research and industry communities during the past decade, predominantly owing to the *Big Data* revolution. Artificial Intelligence (AI) and Machine Learning (ML) are bringing more complexities to data engineering applications, which are now integrated into data processing pipelines to process terabytes of data. Typically, a significant amount of time is spent on data preprocessing in these pipelines, and hence improving its efficiency directly impacts the overall pipeline performance. The community has recently embraced the concept of *Dataframes* as the de-facto data structure for data representation and manipulation. However, the most widely used serial Dataframes today (R, `pandas`) experience performance limitations while working on even moderately large data sets. We believe that there is plenty of room for improvement by taking a look at this problem from a high-performance computing point of view. In a prior publication, we presented a set of parallel processing patterns for distributed dataframe operators and the reference runtime implementation, *Cylon* [1]. In this paper, we are expanding on the initial concept by introducing a cost model for evaluating the said patterns. Furthermore, we evaluate the performance of *Cylon* on the ORNL Summit supercomputer.

1. Introduction

Artificial Intelligence (AI), Machine Learning (ML), and the *Big Data* revolution have introduced an abundance of complex data engineering applications in the data science domain. These applications are now required to process terabytes of data and are orchestrated as an intricate collection of data engineering pipelines. To achieve this, a significant amount of *developer time* is spent on data exploration, preprocessing, and prototyping. Therefore, improving the efficiency of such activities directly impacts the overall data engineering pipeline performance.

Databases and structured query language (SQL) have been the de-facto tool for data preprocessing applications. However, in the early 2000s, the focus shifted significantly towards *Big Data* toolkits and frameworks. These systems (eg. Hadoop [2] and map-reduce [3], Spark [4], Flink [5], etc.) enabled more capabilities than traditional relational database management systems (RDBMS), such as functional programming interface, consuming large structured and unstructured data volumes, deploying in the cloud at scale, etc. Coinciding with the big data developments, enterprise and research communities have invested significantly in artificial intelligence and machine learning (AI/ML) systems. Data analytics frameworks complement AI/ML by providing a rich ecosystem for preprocessing data, as these applications require enormous amounts of data to train their models properly.

In recent times, the data science community has increasingly moved away from established SQL-based abstractions and adopted Python/R-based approaches, due to their user-friendly programming environment, optimized execution backends, broad community support, etc. *Dataframes* play a pivotal role in this transformation [6] by providing a functional interface and interactive development environment for exploratory data analytics. Most dataframe systems available today (e.g. R-dataframe, Pandas) are driven by the open-source community. However, despite this popularity, many dataframe systems encounter performance limitations even on moderately large data sets. We believe that

* Corresponding author.
E-mail addresses: niranda@niranda.dev (N. Perera), djy8hg@virginia.edu (A.K. Sarker), qad5gv@virginia.edu (M. Staylor), laszewski@gmail.com (G. von Laszewski), shankaiying@gmail.com (K. Shan), supun@apache.org (S. Kamburugamuve), chathurawidanage@gmail.com (C. Widanage), vibhatha@gmail.com (V. Abeykoon), thejaka.amila@gmail.com (T.A. Kanewela), vxj6mb@virginia.edu (G. Fox).

https://doi.org/10.1016/j.future.2023.07.007
0167-739X/

dataframe systems have now exhausted the capabilities of a single computer and this paves the way for distributed and parallel dataframe processing systems.

1.1. Background: High-performance dataframes from parallel processing patterns

In the precursor publication, titled "High-Performance Dataframes from Parallel Processing Patterns" [1], we presented a framework that lays the foundation for building high-performance distributed-memory parallel dataframe systems based on parallel processing patterns. There, we analyzed the semantics of common dataframe operators to establish a set of generic distributed operator patterns. We also discussed several significant engineering challenges related to developing a scalable and high-performance distributed dataframe (DDF) system. The main goal of this framework is to simplify the DDF development process substantially by promoting existing serial/ local operators into distributed operators following the said patterns. They primarily focus on a distributed memory and Bulk Synchronous Parallel (BSP) [7,8] execution environment. This combination has been widely employed by the high-performance computing (HPC) community for exascale computing applications with admirable success. Based on this framework, we developed *Cylon*, an open-source high-performance distributed dataframe system [9].

In this paper, we present an in-depth analysis of the aforementioned parallel processing patterns based on a cost model. We encapsulate the parallel processing patterns concept into "*Cylon* Distributed Operator Model" and present "*Cylon* Communication Model" which allows plugging-in multiple communication runtimes into *Cylon* distributed execution. These two aspects constitute the "*Cylon* Distributed Memory Execution Model", which we will discuss in detail in the following sections. Furthermore, we will introduce a cost model to evaluate the performance of distributed memory execution. In addition, we demonstrate the scalability of *Cylon* on leadership-class supercomputing environments, which affirms the significance of the underlying framework. We have also conducted a scalability analysis between *Cylon* and related state-of-the-art data processing systems. This analysis demonstrates the applicability of the design across the board, on both distributed computing and supercomputing infrastructure. In the following sections, we use *Cylon* to refer to its underlying high-performance DDF framework interchangeably.

2. *Cylon* distributed-memory execution model

Cylon is based on the *distributed memory parallel* model, which isolates memory for each parallel process. These processes can manage their memory individually while communicating with others using message passing. This isolation makes distributed operator implementation easier to reason about. While it leaves room for improvement, especially using multi-threading execution, the results show that *Cylon* dataframes show superior scalability over the state-of-the-art systems. In addition, it is based on BSP execution in the distributed memory environment. Gao et al. [10] recently published a similar concept for scaling joins over thousands of Nvidia Graphical Processor Units (GPU). *Cylon* experiments demonstrate that this approach can be generalized to all operators and achieves commendable performance.

Conceptually, we can divide *Cylon* distributed execution model into two distinct sub-models, **1. Communication Model**, and **2. Distributed Operator Model**. We will discuss the former in Section 3 and the latter in Section 4.

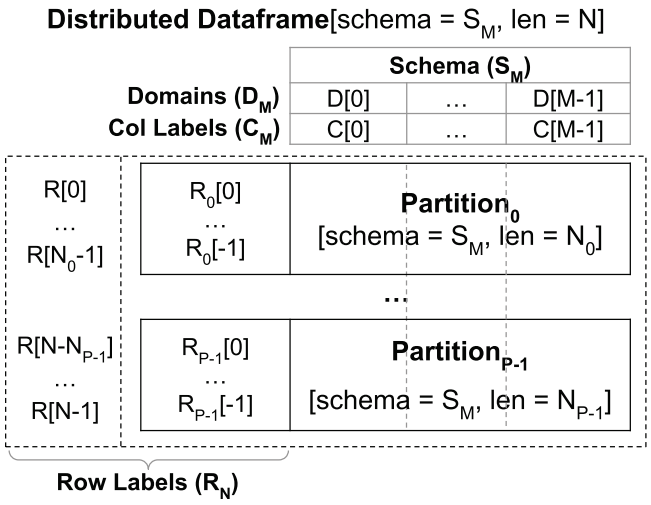


Fig. 1. Distributed memory dataframe abstraction.

2.1. Distributed memory parallel dataframe definition

The primary insight behind *Cylon* is to present a dataframe framework that promotes an already available *serial (local) operator* into a distributed memory parallel execution environment [11]. For this purpose, we formally defined a Distributed Memory Parallel Dataframe based on row-based partitioning in our previous publication [1]. This concept is depicted in Fig. 1. The dotted lines represent the *virtual* collection of *Partitions* in the distributed memory parallel environment. Users would not see a separate distributed API object but instead, continue to write their program as they would work on a single partition. The execution environment determines if the operator needs to be performed locally or in a distributed fashion based on the operator's semantics.

For example, Fig. 2(a) shows a Pandas script that reads data from two directories, joins them, sorts the result, and takes the top 10 rows. A corresponding *Cylon* script for distributed-memory Dataframes is shown in Fig. 2(b).

2.2. Apache Arrow Columnar Memory Layout

Cylon uses Apache Arrow Columnar format as the physical data representation. This is an integral component of the *Cylon* memory model. It provides several benefits, such as data adjacency for sequential access (scans), $O(1)$ (constant-time) random access, SIMD vectorization-friendly data structure, true zero-copy access in shared memory, etc. It also allows serialization-free data access from many language runtimes. Due to these benefits, many libraries including Pandas, PySpark [4], CuDF [12], and Ray [13], are now using the Apache Arrow format.

3. *Cylon* communication model

In many dataframe applications, communication operations take up significant time creating critical bottlenecks. This is evident from our experiments (Section 6), where we evaluate communication and computation time breakdown applied to several dataframe operator patterns. Moreover, most frameworks (eg. Spark, Dask, Ray), provide special guidelines to reduce communication overheads (eg. shuffle routine) [14,15]. Therefore, careful attention has been given while developing the communication model for *Cylon*.

```
df1 = read_csv('dir/path/0') #read
df2 = read_csv('dir/path/1')

df_j = df1.merge(df2, ...) #join
df_s = df_j.sort_values(...) #sort
df_s.iloc[:10] # head(10)
```

(a) Pandas

```
df1 = read_csv_dist('dir/path/0', env=env) #dist read
df2 = read_csv_dist('dir/path/1', env=env)

df_j = df1.merge(df2, ..., env=env) #dist join
df_s = df_j.sort_values(..., env=env) #dist sort
df_s.iloc[:10, env] #dist head(10)
```

(b) Cylon

Fig. 2. Example script.

Table 1
Communication semantics in dataframe operators.

Operation	Data structure		
	Table	Array	Scalar
Send/ Recv	Common	Common	Common
Shuffle (AllToAll)	Common	Rare	N/A
Scatter	Common	Rare	N/A
Gather/AllGather	Common	Common	Common
Broadcast	Common	Common	Common
Reduce/AllReduce	N/A	Common	Common
Barrier	Common (independent of the data structure)		

BSP execution allows the program to continue independently until the next communication boundary is reached. Message passing libraries such as MPI (OpenMPI, MPICH, IBM Spectrum, etc.), Gloo, and UCX [16] provide communication routines for memory buffers, which by extension support homogeneously typed arrays. The most primitive routines are point-to-point (P2P) message passing, i.e., tag-based *async send* and *async receive*. Complex patterns (generally termed *collectives*) can be derived on top of these two primitive routines (eg. MPI-Collectives, UCX-UCC).

Unlike multi-dimensional arrays, heterogeneous data types in dataframes make communication routines more involved. The Arrow columnar data format represents a column by a tuple of buffers (`boolean` validity bitmap, `integer` offsets, & `byte` data). A dataframe incorporates a collection of such columns. Therefore, a communication routine would have to be called on each of these buffers. *Cylon* communication model outlines a set of communication collectives required to implement distributed memory parallel dataframes by inspecting the semantics of core dataframe operators. These are listed in Table 1 together with their frequency of usage for each data structure.

The key features of the *Cylon* communication model are,

1. **Modular architecture**: Allows plugging-in multiple communication libraries.
2. **Extensibility**: The communication model has been easily extended into Nvidia CUDA GPU hardware, in *GCylon* project.

Fig. 3 depicts the overall *Cylon* architecture.

3.1. Communicator

The *communicator* interface manages *Cylon* communication routines (Fig. 4). At the very top, the user API defines routines based on the data layer data structures, as described in Table 1. These are blocking routines for the user (e.g., `shuffle_table` will wait until completion).

The communicator implements these routines using two abstract constructs, (1). channels (for point-to-point/ send–receive communications) and (2). collective communications. The former works only on byte buffers, and the collectives can also be implemented using these channels. In fact, `table_shuffle` is implemented using channels due to a mismatch in traditional

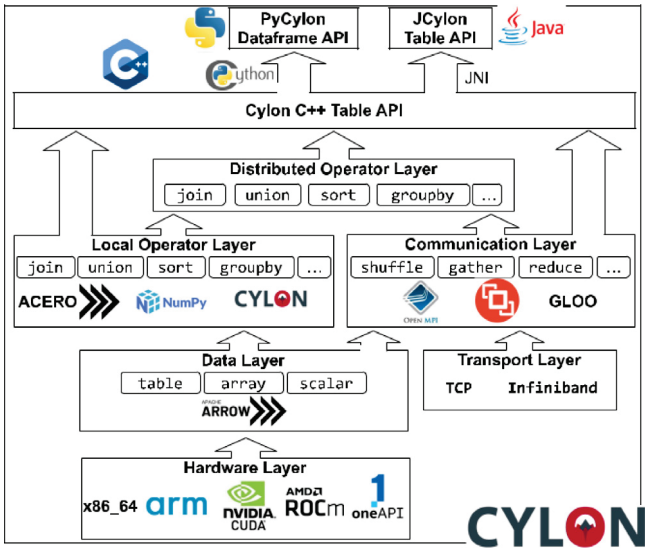


Fig. 3. *Cylon* Architecture.

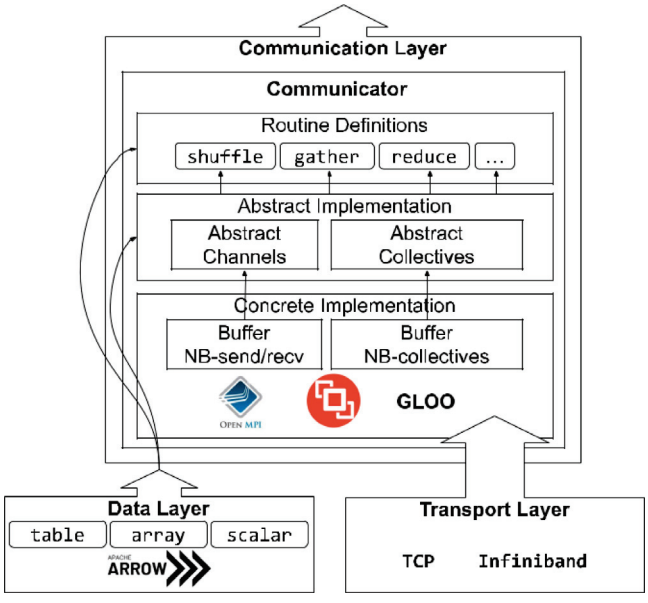


Fig. 4. *Cylon* Communicator model.

`MPI_Alltoall`. The abstract collective communications implement collective routines for composite data structures (tables,

arrays, and scalars), using collectives on buffers. This abstract implementation allows *Cylon* to easily plug in multiple communication libraries that support BSP semantics, such as OpenMPI [17], UCX [16], and Gloo [18].

3.2. Abstract channels

Channels are designed to be used for composite buffer communications in a non-blocking manner. During the initialization, it registers two callbacks which inform the caller that (1). the sending has been completed, (2). the data is received for a particular buffer. It then accepts *requests* that contain the buffer address and metadata (such as buffer size, buffer index, etc.) to be sent. The caller then has to progress through sends and receives. First, the channel exchanges buffer metadata, which is used to allocate memory for receiving buffers. Later on, it starts exchanging data. Both these progressions use non-blocking send/receive routines. Once each receiving buffer completes, it will be passed on to the caller using the receive-callback.

Channels give much flexibility to the caller to implement composite communication routines. However, there are disadvantages to this as well. Most importantly, each buffer collective routine must be implemented from scratch using channels. As listed in Table 3, we need to implement multiple communication algorithms to get the best performance for collectives. Managing such a custom communication library code base could be a cumbersome exercise. Currently, `shuffle` routine is implemented using the channels.

3.3. Abstract collectives

Abstract collectives are higher-level communication abstraction that implements table, array, or scalar collectives using non-blocking buffer collective routines. For example, an `allgather table` can be implemented as a collection of non-blocking `allgather` routines. To do this, we create a metadata structure with the buffer pointers, sizes, data types, etc. of the input table and call corresponding communication routines on each buffer. In the end, we recreate the resultant table based on the output buffers.

3.4. Supported communication libraries

Currently, *Cylon* communicator supports the following communication libraries that support BSP message-passing semantics.

3.4.1. OpenMPI

OpenMPI is a widely used open-source implementation of the MPI specification. It consists of two main components, (1). process management and (2). communication library. Currently, Process Management Interface Exascale (PMIx) standard [19] is used for the former, while various communication algorithms have been implemented (Table 3) as a part of the latter. It is a comprehensive communication library with a rich collection of communication routines for many distributed computing and HPC applications. *Cylon* communication model was also heavily influenced by OpenMPI.

3.4.2. Gloo

Gloo collective communications library is managed by Meta Inc. incubator [18] predominantly aimed at machine learning applications. PyTorch uses it 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). Gloo lacks a comprehensive algorithm implementation as an incubator project, yet our experiments confirmed that it scales admirably. We have extended the Gloo project to suit *Cylon* communication interface.

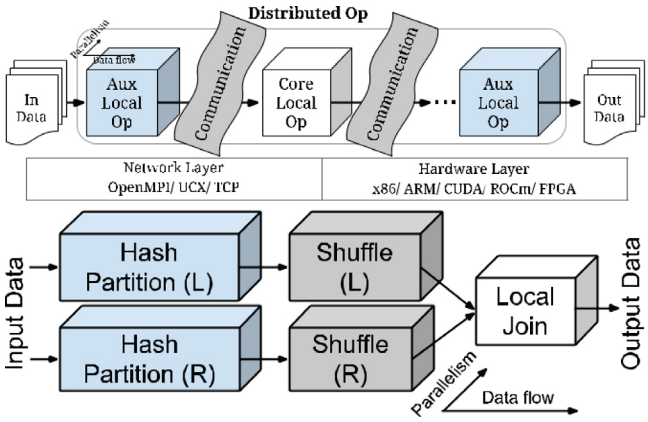


Fig. 5. Distributed DDF Sub-operator composition [1] (Bottom: Join operator example).

3.4.3. UCX/UCC

Unified Communication X (UCX) 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 Communications (UCC) is a collective communication operation API built on 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.

4. *Cylon* distributed operator model

Cylon distributed operator model provides the basis for elevating a local dataframe operator to a distributed memory parallel dataframe operator. This was the primary idea behind our precursor publication [1]. It comprises two key observations,

1. A distributed operator consists of three major sub-operators:
   (a) Core local operator
   (b) Auxiliary local operators
   (c) Communication operators

   For example, the bottom image in Fig. 5 shows how the distributed `join` is composed of these sub-operators.
2. By examining the composition of these sub-operators, they can be categorized into several parallel execution patterns, as depicted in Fig. 6. Therefore, rather than analyzing/optimizing each operator, we can focus on these parallel patterns. In addition, some operators can be implemented using multiple algorithms that show distinctive parallel patterns (e.g., `join` can be done by shuffling or by broadcasting). Hence, understanding these patterns is essential to choose the best runtime strategy.

Table 2
Generic dataframe operator patterns.

Pattern	Operators	Result semantic	Communication
Embarrassingly parallel	Select, Project, Map,Row-Aggregation	Partitioned	–
Loosely Synchronous			
• Shuffle Compute	Union, Difference,Join, Transpose	Partitioned	Shuffle
• Combine Shuffle Reduce	Unique, GroupBy	Partitioned	Shuffle
• Broadcast Compute	Broadcast-Join*	Partitioned	Bcast
• Globally Reduce	Column-Aggregation	Replicated	AllReduce
• Sample Shuffle Compute	Sort	Partitioned	Gather, Bcast, Shuffle, AllReduce
• Halo Exchange	Window	Partitioned	Send-recv
Partitioned I/O	Read/Write	Partitioned	Send-recv, Scatter, Gather

Specialized join algorithm.

Table 3
Complexity of Communication Operations.

Operation	Algorithm	Startup time ($T_{startup}$)	Transfer time ($T_{transfer}$)	Reduction time (T_{reduce})
Shuffle/AllToAll	isend-irecieve [20]	$O(P)$	$O(\frac{P-1}{P} * n)$	–
	Ring [21]	$O(P)$	$O(P * n)$	–
	Pairwise Exchange [20]	$O(P)$	$O(n)$	–
	Bruck [22]/ Modified Bruck [21]	$O(\log P)$	$O(\log P * \frac{n}{2})$	–
AllGather	Ring [20]	$O(P)$	$O(\frac{P-1}{P} * N)$	–
	Recursive Doubling [20]	$O(\log P)$	$O(\frac{P-1}{P} * N)$	–
	Bruck [20]	$O(\log P)$	$O(\frac{P-1}{P} * N)$	–
Broadcast	Binomial Tree [20]	$O(\log P)$	$O(\log P * n)$	–
	Scatter-AllGather [23]	$O(\log P + P)$	$O(\frac{P-1}{P} * n)$	–
Reduce	Binomial Tree [20]	$O(\log P)$	$O(\log P * n)$	$O(\log P * n)$
	Reduce-Scatter Gather [24]	$O(\log P)$	$O(\frac{P-1}{P} * n)$	$O(\frac{P-1}{P} * n)$
AllReduce	Binomial Tree [20]	$O(\log P)$	$O(\log P * n)$	$O(\log P * n)$
	Recursive Doubling [20]	$O(\log P)$	$O(\log P * n)$	$O(\log P * n)$
	Reduce-Scatter AllGather [24]	$O(\log P)$	$O(\frac{P-1}{P} * n)$	$O(\frac{P-1}{P} * n)$

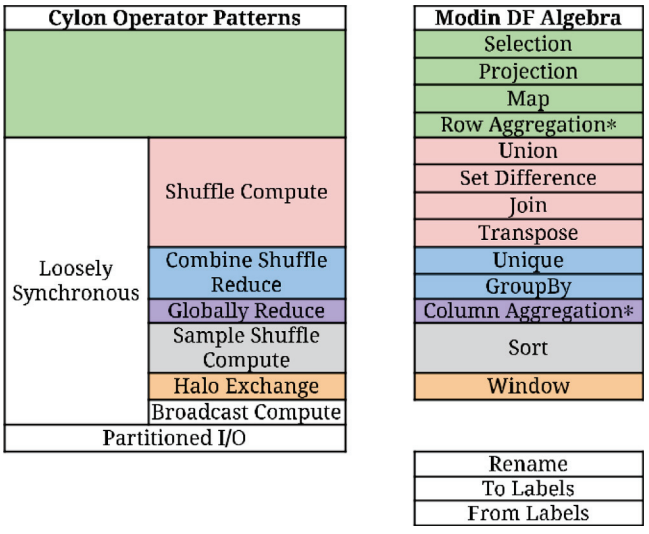


Fig. 6. *Cylon* Operator Patterns & Modin DF Algebra.

We believe understanding distributed dataframe operator patterns reduce the burden of parallelizing a massive API, such as Pandas. To address the same problem, Petersohn et al. [25] introduced a primitive set of dataframe operators that could be used as a *basis* for the rest, termed *Dataframe Algebra*. Our dataframe operator patterns are a complementary concept to dataframe algebra, as shown in Fig. 6.

4.1. Core local operator

These refer to single-threaded implementations of primitive operators. There could be one or more libraries that provide this functionality, such as `numpy`, `pandas`, RAPIDS CuDF [12], Acero (Apache Arrow Compute), etc, or locally developed as a part of *Cylon*. The choice of the library depends on the language runtime, the underlying memory format, and the hardware architecture. This is to prevent redundant development efforts for reinventing the existing functionality.

4.2. Auxiliary sub-operators

Partition operators are essential for distributed memory applications. Partitioning determines how a local data partition is split into subsets so they can be sent across the network. This operator is closely tied with *Shuffle* communication routine. Hash partition, range partition, and rebalance are several key auxiliary operators.

4.3. Parallel processing patterns & operator implementations

According to our previous publication, dataframe operators can be broadly separated into three categories [1], as described in Table 2.

1. Embarrassingly parallel: Operators that require no communication required

2. Loosely synchronous: Operators that require communication at some stage in its implementation. This is a broad category; therefore, it is separated into the following subcategories.
   (a) Shuffle-compute
   (b) Sample-shuffle-compute
   (c) Combine-shuffle-reduce
   (d) Broadcast-compute
   (e) Globally reduce
   (f) Halo exchange
3. Partitioned I/O: I/O operators in distributed memory parallel environments require communication to load balance data amongst the workers.

5. Cost model for evaluation

A cost model can be applied to the *Cylon* distributed operator model to estimate the execution time/ cost of each operator pattern. As observed before, each pattern comprises three sub-operators. Hence, the total cost estimate (T_{total}) is the sum of the cost of each sub-operator.

$$T_{total} = T_{core} + T_{aux} + T_{comm}$$

1. $T_{core} \rightarrow$ Core local operator cost
2. $T_{aux} \rightarrow$ Auxiliary local operator cost
3. $T_{comm} \rightarrow$ Communication operator cost

We analyze the communication and computation cost of distributed dataframe operators in the subsequent sections, and the following notation has been used.

- $P \rightarrow$ Parallelism
- $N \rightarrow$ Total number of rows
- $n = N/P \rightarrow$ Number of rows per process
- $c \rightarrow$ Number of columns (constant for row-partitioned data)
- $\mathbf{N} = N \times c \rightarrow$ Total amount of distributed work/ total data
- $\mathbf{n} = \mathbf{N}/P \rightarrow$ Work per process/ rows per process
- $\mathbf{C} \rightarrow$ Cardinality of data

5.1. Communication cost (T_{comm})

Based on the literature, Hockney [26], LogP [27], and LogGP [28] are some of the most commonly used cost models to evaluate collective communication operations. *Hockney model* provides a simple communication cost estimation, and therefore, it has been used in many recent publications [20–22,29]. The model fails to capture the network congestion. However, it provides an adequate cost estimation to evaluate *Cylon*. The model assumes that the taken to send a message between any two nodes can be modeled as,

$$T = \alpha + n\beta$$

1. $n \rightarrow$ Message size/ number of bytes transferred
2. $\alpha \rightarrow$ Latency/ startup time per message (independent of n)
3. $\beta \rightarrow$ Transfer time per byte

Let us take *Shuffle (AllToAll)* for an example. *Cylon* uses non-blocking send-receive-based implementation. Each worker would shuffle $\mathbf{n}$ data with others in P iterations. In each iteration, it would send and receive $\frac{\mathbf{n}}{P}$ amount of data (on average, for uniformly distributed data). Out of the P iterations, one iteration is a local data transfer. Therefore,

$$T_{shuffle} = (P-1)(\alpha + \frac{\mathbf{n}}{P}\beta) = (P-1)\alpha + \frac{(P-1)\mathbf{n}}{P}\beta$$

Therefore, for row-partitioned data,

$$T_{shuffle} = T_{startup} + T_{transfer} = O(P) + O(\frac{P-1}{P} \times n)$$

Table 3 describes the communication costs of communication routines used in distributed dataframe operator implementations for multiple algorithms based on the Hockney model. It uses the definitions described in Section 5.

Table 4
Core local operator cost (T_{core})

Local operation	Cost (T_{core})	Output size (n_{new})
Selection, Map	$O(n)$	$O(n)$
Row-aggregation	$O(nc) = O(n)$	$O(n)$
Projection	$O(c)$	$O(nc)$
Union	$O(nc) = O(n)$ (hash-based)	$O(n\mathbf{C})$
Set-difference	$O(nc) = O(n)$ (hash-based)	$O(n)$
Hash-Join	$O(n) + O(\frac{n}{\mathbf{C}})$	$O(\frac{n}{\mathbf{C}})$
Sort-Join	$O(n \log n) + O(\frac{n}{\mathbf{C}})$	$O(\frac{n}{\mathbf{C}})$
Transpose	$O(nc)$	$O(nc)$
Unique	$O(nc) = O(n)$ (hash-based)	$O(n\mathbf{C})$
GroupBy	$O(n)$ (hash-based)	$O(n\mathbf{C})$
Column Aggregation	$O(nc) = O(n)$	$O(c)$
Sort	$O(n \log n)$	$O(n)$

5.2. Computation cost ($T_{core} + T_{aux}$)

Core local operator cost (T_{core}) & auxiliary local operator cost (T_{aux}) constitutes the computation cost. Since these are local operations, the cost can be derived from *time complexity of the algorithm*. For example, a local `sort` operation would take (when using a quick-sort algorithm for uniformly distributed data), $T_{sort} = O(n \log n)$ Table 4 describes the time complexities of commonly used local dataframe operators (Core local operator cost, T_{core}) and their output size (n_{new}).

5.3. Total cost of dataframe operator patterns

We will look at the total cost of each operator pattern in the following subsections.

5.3.1. Embarrassingly parallel

This is the most trivial class of operators since they do not require any communication to parallelize the computation. *Select*, *Project*, *Map*, and *Row-Aggregation* fall under this pattern. Arithmetic operations (ex: `add`, `mul`, etc.) are also good examples of this pattern. Embarrassingly parallel distributed operators can simply call the corresponding local operator, and therefore the cost estimation of this pattern is,

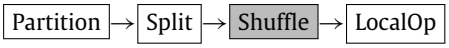
$$T_{EP} = O(n)$$

5.3.2. Shuffle compute

This common pattern can be used for operators that depend on *Equality/Key Equality of rows*. Of the core dataframe operators, `join`, `union` and `difference` directly fall under this pattern. In contrast, `transpose` follows a more nuanced approach.

Partitioning and shuffling communication routines rearrange the data so that equal/key-equal rows are on the same partition

at the end of the operation. This guarantees that the corresponding local operation can be called at the end of the shuffling stage. *Join*, *Union* and *Difference* operators follow this pattern:

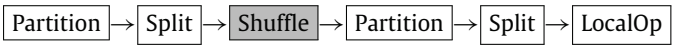


Therefore, the cost estimation of shuffle compute for each worker is,

$$T_{shuffle\_compute(hash)} = O(n) + O(P) + O(\frac{P-1}{P} \times n) + T_{core}$$

$$T_{shuffle\_compute(range)} = O(\log P) + O(n) + O(P) + O(\frac{P-1}{P} \times n) + T_{core}$$

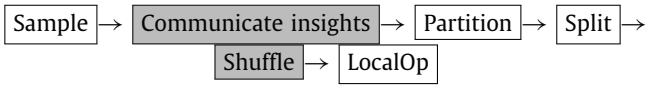
Typically partitioning schemes (hash, range, etc.) are map operators and, therefore, access memory locations contiguously. These can be efficiently executed on modern SIMD-enabled hardware. However, the local operator may need to access memory randomly (e.g., a join that uses a hash table). Therefore, allowing the local operator to work on in-cache data improves the efficiency of the computation. This can be achieved by simply attaching a *local partition* block at the end of the `shuffle`.



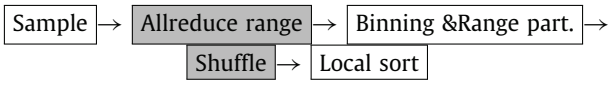
A more complex scheme would be to partition data into much smaller sub-partitions from the beginning of the pipeline. Possible gains on each scheme depend heavily on runtime characteristics such as the data distribution.

5.3.3. Sample shuffle compute

This pattern is an extension of the shuffle-compute pattern. Sampling is commonly used for operators such as distributed `sort`. It gives an overview of the data distribution, which needs to be communicated among the other workers to determine an ordered (range) partition scheme. This can be achieved trivially by calling `all reduce` operation, or by a composite of communication & computation steps (eg. sample sort).

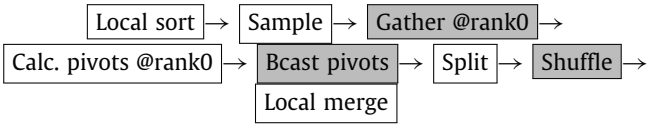


Cylon uses multiple algorithms for distributed `sort` implementation. The data can be range-partitioned for numerical key columns based on a key-data histogram, and it would have the following total cost per worker.



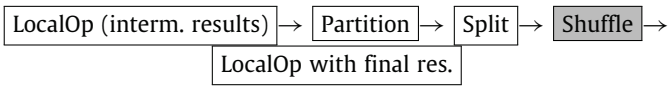
$$T_{sort(range)} = O(\log P) + O(n) + O(P) + O(\frac{P-1}{P} \times n) + O(n \log n)$$

For the rest, *Cylon* uses *sample sort* with regular sampling [30]. It sorts data locally and sends a sample to a central entity that determines pivot points for data. Based on these points, sorted data will be split and shuffled. Finally, all executors merge the received sub-partitions locally.



5.3.4. Combine shuffle reduce

Another extension of the *Shuffle-Compute* pattern, Combine-Shuffle-Reduce, is semantically similar to the map-reduce [3] paradigm. The operations that reduce the output length, such as *Groupby* and *Unique*, benefit from this pattern. The effectiveness of combine-shuffle-reduce over shuffle-compute depends on the *Cardinality* (**C**) (i.e., the ratio of unique rows to the total length). It follows,

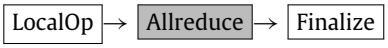


The initial local operation reduces data into a set of intermediate results (similar to the Combine step in *MapReduce*), which would then be shuffled. Upon their receipt, a local operation is performed to finalize the results. The author also discusses this approach for dataframe reductions in a recent publication [31]. At the end of the initial local operation, the output dataframe size (in each worker) is $O(n\mathbf{C})$. Therefore, the total cost per worker would be,

$$T_{comb\_shuf\_red} = T_{core}(n) + O(n\mathbf{C}) + O(P) + O(\frac{P-1}{P} \times n\mathbf{C}) + T_{core}(n\mathbf{C})$$

5.3.5. Globally reduce

This pattern is most commonly seen in dataframe *Column-Aggregation* operators. It is similar to the embarrassingly parallel pattern but requires an extra communication step to arrive at the final result. For example, calculating the column-wise `mean` requires a local summation, a global reduction, and a final value calculation.



Some utility methods such as *distributed length* and *equality* also follow this pattern. For large data sets, the complexity of this operator is usually governed by the computation rather than the communication.

5.3.6. Halo exchange

This pattern is observed in window operations. A window operation performs an aggregation over a sliding partition of values. Pandas API supports rolling and expanding windows. For row partitions, the windows at the boundaries would have to communicate with their neighboring partitions and exchange partially computed results. The amount of data sent/received is based on the window type and individual length of partitions.

5.3.7. Broadcast compute

Broadcast compute is a scaled-down pattern from shuffle-compute. Rather than shuffling, certain operators like broadcast-join can use broadcasting. This strategy only becomes useful when there is a smaller relation so that it can be broadcasted without shuffling the large relation. It reduces communication overhead significantly. However, broadcast-joins would perform poorly if the relations were of the same order. This effect was observed in Modin [25], where out-of-memory errors are reported even for moderately large datasets because it only employs broadcast joins.

5.3.8. Partitioned I/O

Partitioned Input parallelizes the input data (CSV, JSON, Parquet) by distributing the files to each executor. It may distribute a list of input files to each worker evenly. Alternatively, it receives a custom one-to-many mapping from the worker to input file(s). It reads the input files according to the custom assignment. For Parquet files, *Partitioned Input* tries to distribute the number of rows to each partition as evenly as possible when metadata is present. Suppose an executor does not receive data from reading. In that case, it constructs an empty dataframe with the same schema as the other partitions. In *Partitioned Output*, each executor writes its partition dataframe to one file.

5.4. Runtime aspects

5.4.1. Cardinality

Equality of rows governs the *Cardinality* of a Dataframe **C**, which is the number of unique rows relative to the length. Therefore, $\mathbf{C} \in [\frac{1}{N}, 1]$, where $\mathbf{C} = \frac{1}{N} \implies$ rows are identical and $\mathbf{C} = 1 \implies$ all rows are unique. In the *Combine-Shuffle-Reduce* pattern, the initial local operation has the potential to reduce communication order to $n' < n$. This gain depends on the *Cardinality* ($\mathbf{C}$) of the dataframe $\mathbf{C} \in [\frac{1}{N}, 1]$, which is the number of unique rows relative to the length. $\mathbf{C} \sim \frac{1}{N} \implies n' \lll n$, making the combine-shuffle-reduce much more efficient than a shuffle-compute. Consequently, when $\mathbf{C} \sim 1 \implies n' \sim n$ may in fact worsen the combine-shuffle-reduce complexity. In such cases, the shuffle-compute pattern is more efficient. This incident is very evident from the cost model.

$$T_{comb\_shuf\_red} = T_{core}(n) + O(n\mathbf{C}) + O(P) + O(\frac{P-1}{P} \times n\mathbf{C}) + T_{core}(n\mathbf{C})$$

vs

$$T_{shuf\_comp} = O(n) + O(P) + O(\frac{P-1}{P} \times n) + T_{core}(n)$$

When, $\mathbf{C} \to 1 \implies T_{comb\_shuf\_red} \to T_{shuf\_comp}$, and in fact, it is worse because the core local operation would have to be carried out twice.

5.4.2. Data distribution

Data distribution heavily impacts the partitioning operators. Some executors may be underutilized when unbalanced partitions exist, affecting the overall distributed performance. *Work-stealing* scheduling is a possible solution to this problem. In a BSP environment, pseudo-work-stealing execution can be achieved by storing partition data in a shared object-store. Furthermore, some operations could employ different operator patterns based on the data distribution. For instance, when one relation is very small by comparison, *Join* could use a `broadcast_join` (broadcast-compute) rather than a hash-shuffle join (shuffle-compute) to achieve better performance.

5.4.3. Out-of-core execution

Currently, *Cylon* is limited by the memory available to the workers. With the data immutability guarantees, it always allocates new memory for the columns that get modified. Therefore, loosely synchronous patterns may require a workspace of $3-4\times$ the size of the table. This could be a challenging requirement for memory-constrained environments and limits the dataset size we could process. Therefore, the system needs to be able to execute operators out-of-core.

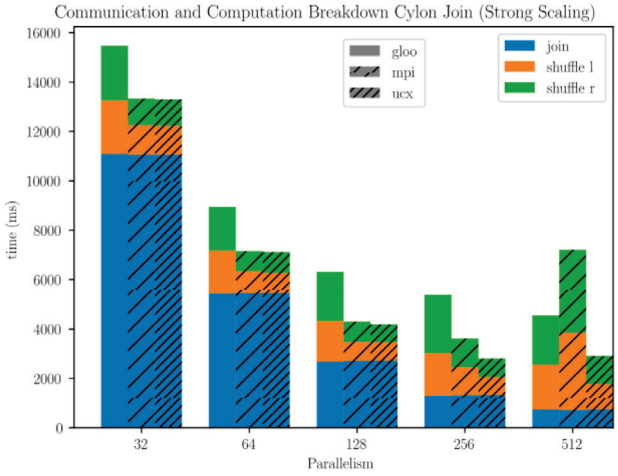


Fig. 7. Computation and communication breakdown - `join` (Strong scaling).

5.4.4. Logical plan optimizations

A typical SQL query may translate to multiple Dataframe operators, and the application script can include several such queries. Semantically, these operators construct a DAG (directed acyclic graph) or a *logical plan*. SQL and data engineering engines generate an *optimized logical plan* based on rules (ex: predicate push-down) or cost metrics. While these optimizations produce significant gains in real-life applications, this is an orthogonal detail to the individual operator patterns we focus on in this paper.

6. Experiments

To evaluate the performance of *Cylon* distributed-memory execution model, we have conducted the following experiments.

- Communication and computation breakdown of *Cylon* operators for strong and weak scaling
- Running *Cylon* in Oak Ridge National Laboratory Summit supercomputer
- Comparing *Cylon* performance against the state-of-the-art data processing systems

For the following experiments, uniformly random distributed data was used with two `int64` columns in column-major format (Fortran order). Data uses a cardinality of 90% (i.e. 90% of rows are unique), which constitutes a worst-case scenario for key-based operators (eg. join, sort, groupby, etc.). The main focus of these experiments is to micro-benchmark the distributed operator implementation. Using a generated dataset allows the input dataset to be uniformly distributed and thereby evaluate the true performance of the kernels. Barthels et al. followed a similar approach to evaluate distributed join kernels [32].

6.1. Communication & computation

These experiments were carried out on a 15-node Intel® Xeon® Platinum 8160 cluster. Each node comprises 48 hardware cores on two sockets, 255 GB RAM, and SSD storage, and is connected via Infiniband with 40Gbps bandwidth.

Fig. 7 shows communication and computation time breakdown for `join` operation for a strong scaling test (1B rows per table). Moreover, Fig. 8 shows the same for a weak scaling test (25M per worker per table). Out of many operators, `joins` have

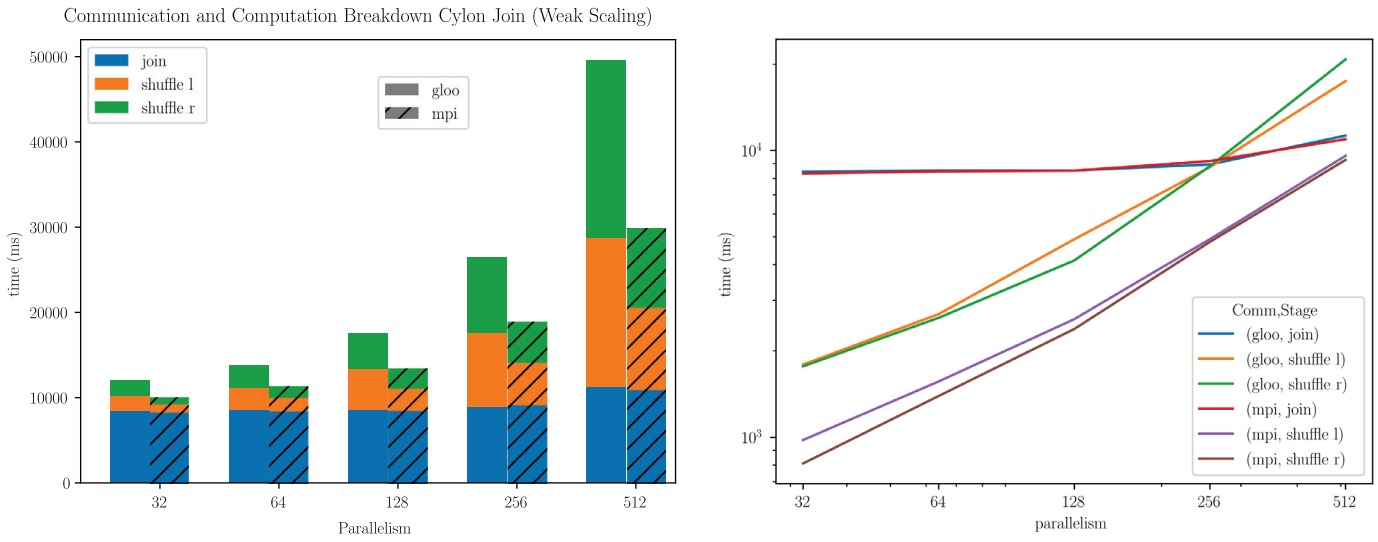


Fig. 8. Computation and communication breakdown - `join` (Weak scaling).

the most communication overhead, as it is a binary operator (2 input DFs).

In the strong scaling plot, 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, the author needed more expertise in the Spark, Dask, or Ray DDF code base to run a similar micro-benchmark. This experiment shows that communication plays a significant role in dataframe operator implementation. Despite using libraries specialized for message passing, *Cylon* still encounters significant communication overhead. Therefore, careful consideration must be given to communication while developing distributed dataframe runtimes.

The weak scaling plot can further analyze the impact of communication performance. The work per process is fixed; therefore, we should see a flat graph. However, as we see in Fig. 8, the time increases along the parallelism axis, indicating that the communication overhead increases. The graph on the right plots each stage (log–log). The local join computation is relatively flat, while both shuffle stages (left & right) show a linear increase.

6.1.1. Examining the results using the cost model

By looking at the cost model in Section 5, the cost of `join` would be,

$$T_{shuffle} = O(P-1) + O(\frac{P-1}{P} \times n)$$

$$T_{join(sort)} = O(P-1) + O(\frac{P-1}{P} \times n) + O(n) + O(n \log n) + O(\frac{n}{\mathbf{C}})$$

Substituting $n = N/P$,

$$T_{join(sort)} = O(P-1) + O(\frac{P-1}{P} \times \frac{N}{P}) + O(\frac{N}{P}) + O(\frac{N}{P} \log \frac{N}{P}) + O(\frac{N}{P\mathbf{C}})$$

For strong scaling, N is constant. Therefore, as P increases, the components that depend on n (in computation and communication) reduce. This results in a downward trend in wall time. However, the $O(P-1)$ component (coming from the communication cost) overtakes the gains of reducing n. This explains the increase in wall time in higher parallelisms.

Similarly, for weak scaling, n is kept constant, which reduces the cost to $O(P-1) + O(\frac{P-1}{P})$. For the parallelism values tested in the experiments (Fig. 8), this explains the increasing wall-time values and linear upward trends in shuffle timings. Even though the amount of data transferred per worker remains constant (n), the cost model does not account for network congestion. This could explain the increasing gradient at higher parallelisms.

In the following sections, we will see that *Cylon* outperforms the state-of-the-art data engineering systems available today. However, the weak scaling indicates that *Cylon* still needs to improve on the communication operator performance (such as *shuffle*). It would be worthwhile evaluating other algorithms such as Pairwise Exchange [20], Bruck [22]/ Modified Bruck [21], etc., that have better time complexity as the parallelism increases. Another option would be to completely offload the shuffle implementation to the communication library (MPI, Gloo, UCX) and let the library decide which algorithm to choose based on runtime characteristics.

6.2. Cylon on ORNL Summit supercomputer

Cylon was run on the Summit supercomputer at Oak Ridge National Laboratory (ORNL) as a part of large-scale testing. Each node in Summit consists of two IBM POWER9 processors and six Nvidia Tesla V100 accelerators, and there are 4600 of these nodes available for computation, reaching a theoretical peak double-precision performance of approximately 200 PF. Each node consists of 512 GB of RAM and 42 hardware cores. Fig. 9 shows the architecture of a single node in Summit. For *Cylon* workloads, only the CPU nodes were used.

6.2.1. Setting up cylon in summit

Setting up *Cylon* environment in Summit proved to be a tedious undertaking. Generally, *Cylon* is installed via a Conda Python environment [34], which conveniently installs dependencies using the official Anaconda packages. However, due to the Summit node hardware architecture, some of these default packages were failing unexpectedly. Most notably, we encountered memory allocation errors from the Apache Arrow library. Since

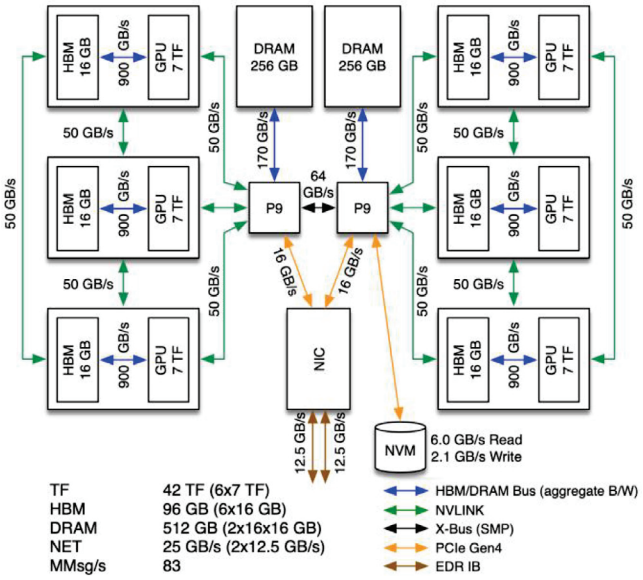


Fig. 9. ORNL summit node architecture [33].

this is an essential requirement for *Cylon*, we had to rebuild Apache Arrow natively on Summit hardware architecture. This was done by the native *Cylon* installation script which uses PyPI (`pip`) environment [35].

Additionally, Summit supercomputer uses its own MPI implementation based on IBM Spectrum MPI [36]. At the time, *Cylon* was tested on OpenMPI and Microsoft MPI only, and therefore, several minor changes were required to properly link with Summit MPI modules.

The recommended way of using custom software in Summit is to create a module and load it (with dependencies) in batch scripts. However, this requires advanced expertise in Summit package management. We bypassed this requirement by installing *Cylon* and its dependencies into a PyPI environment using a login node. This PyPI environment resides in the user space in the file system. When submitting a batch job, we would activate this environment and run our *Cylon* script.

Following is an example batch script for a *Cylon* workload.

```
#!/bin/bash
#BSUB -P <project name>
#BSUB -W 1:30
#BSUB -nnodes 8
#BSUB -alloc_flags smt1
#BSUB -J cylonrun-s-8
#BSUB -o cylonrun-s-8.    #BSUB -e cylonrun-s-8.
module load python/3.7.7 gcc/9.3.0
source $HOME/CYLON/bin/activate
BUILD_PATH=$HOME/cylon/build
export LD_LIBRARY_PATH=$BUILD_PATH/arrow/install/lib64:
     $BUILD_PATH/glog/install/lib64:$BUILD_PATH/lib64:
     $BUILD_PATH/lib:$LD_LIBRARY_PATH

time jsrun -n $((8*42)) -c 1 python $HOME/cylon/summit/
     scripts/cylon_scaling.py -n 9999994368 -s s
```

Both installation and batch scripts are available in the *Cylon* GitHub repository [9].

Table 5
Summit weak scaling results.

Cores	Rows (Mn)	Size (GB)	Throughput (Tuples/s)
1	50	1	3,261
42	2,100	34	110,437
84	4,200	67	186,267
168	8,400	134	384,137
336	16,800	269	729,943
672	33,600	538	1,377,837
1,344	67,200	1,075	2,561,797
2,688	134,400	2,150	4,513,890
5,376	268,800	4,301	7,657,451
10,752	537,600	8,602	11,814,754

6.2.2. Strong scaling

A strong scaling experiment was carried out on *Cylon* `join` operation of two 10 billion row tables. The size of each table is around 160 GB. The parallelism was increased from 4 nodes ($4 \times 42 = 168$ cores) to 25 nodes ($256 \times 42 = 10{,}752$ cores). Fig. 10 plots the results on a log–log scale.

Fig. 10(a) shows 10 billion rows per table experiment. As the parallelism increases from 168 to 2688, the wall time reduces almost linearly with fairly consistent timings. However, from thereon, the timings take a drastic turn and show a higher variance. From 5376 onward, the computation component is less than 2 million rows per table per core. Therefore, communication would dominate the final wall time.

To further analyze this scenario, another 50 billion rows per table experiment was carried out (Fig. 10(b)). There, smaller parallelism experiments were unsuccessful due to memory limitations. However, for higher parallelisms, the wall time reduces fairly linearly, as expected. This indicates that, as long as the computation dominates the communication, performance gains can be achieved by adding more resources. For 50 billion cases, the inflection point would occur at higher parallelism than 10,752.

6.2.3. Weak scaling

A weak scaling experiment was carried out again on *Cylon* `join` operation. The intention was to utilize the memory available in the node allocation fully. Considering the 512 GB RAM and 42 cores per node, it was decided to use 50 million row tables per core. The number of cores has been increased from 1 to 10,752, where the last experiment joins more than 1 trillion rows from the two tables. The results are depicted in Fig. 11.

As we saw in the previous weak scaling experiments, the wall time increases with parallelism. This is not ideal for a weak scaling plot. However, the main culprit for this increase is the *shuffle* communication overhead. However, *Cylon* was able to successfully process more than 17 terabytes (TB) of data across 10,752 cores which is a commendable achievement. When looking at the throughput of the operation, it steadily increases to close to 12 million tuples/second (see Table 5) .

6.3. Cylonvs. the state-of-the-art

In order to evaluate the performance of the distributed-memory execution model discussed in this paper, we performed a strong scaling analysis on several state-of-the-art distributed dataframe systems that are described in the related work section (Section 7). Experiments were also carried out on Pandas [37] to get a serial performance baseline. The following frameworks were considered. We tried our best to refer to publicly available documentation, user guides, and forums while carrying out these tests to get the optimal configurations.

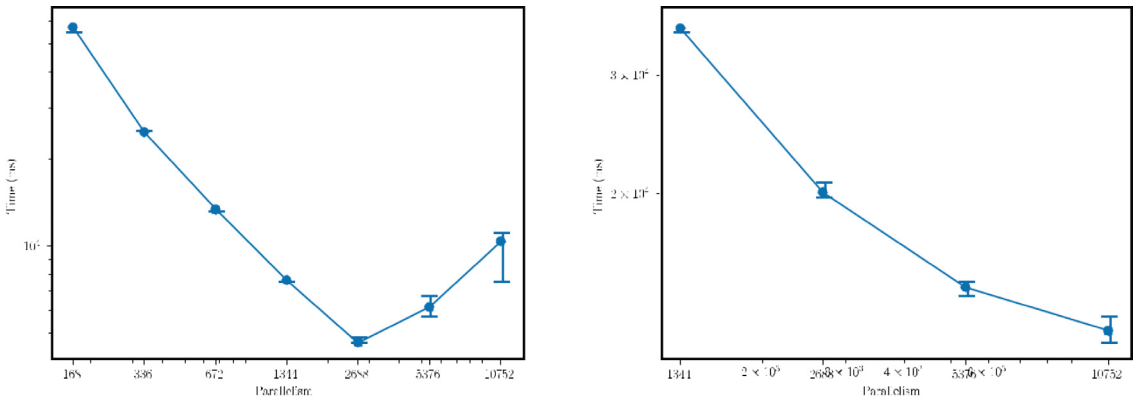


Fig. 10. *Cylon* Strong Scaling on Summit.

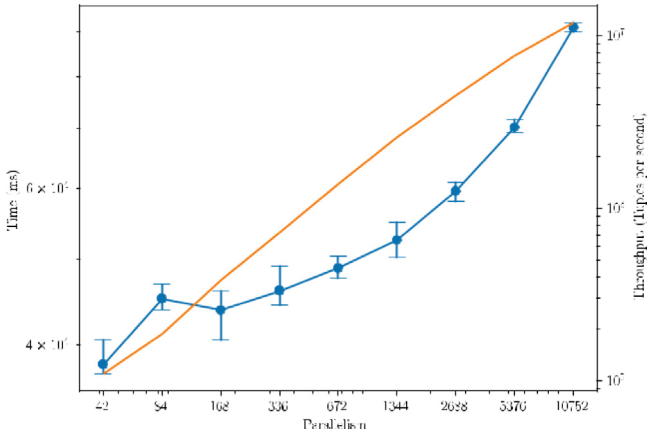


Fig. 11. *Cylon* Weak scaling on summit.

- Dask Distributed Dataframes v2022.8
- Ray Datasets v1.12
- Modin Distributed Dataframe v0.13
- Apache Spark (Pandas-on-Spark) v3.3

We have carried out similar strong scaling analyses in the precursor publication [1,38], and several others [11,39,40]. In this publication, the results have been updated to the latest versions of software and their dependencies. The same 15-node Intel® Xeon® Platinum 8160 cluster described in Section 6.1 was also used for these experiments.

The following dataframe operator patterns were used for the experiments. When evaluating large-scale data engineering use cases (eg. TPC benchmarks [41], Deep Learning Recommendation Model (DLRM) preprocessing [42], etc.) and based on our prior experience, these operator patterns [11,38] consume the majority of the computation time.

- Shuffle Compute - *Join* operator
- Combine Shuffle Reduce - *GroupBy* operator
- Sample Shuffle Compute - *Sort* operator

Fig. 12 depicts two sets of strong-scaling experiments. *Left* column represents tests on one billion-row dataset with all systems, while the *Right* column represents a smaller 100 million-row dataset with *Cylon*, Dask, and Spark systems. *Cylon* was using the UCX/UCC [16] communicator, as it shows the best distributed performance.

Unfortunately, several challenges were encountered with running tests on Ray Datasets. It only supports unary operators (single input) currently. Therefore it has been omitted from *Join* experiments. Moreover, Ray `groupby` did not complete within 3 h, and `sort` did not show presentable results. Several issues came up with Modin as well. It only supports `broadcast_join` implementation, which performs poorly on two similar-sized dataframe *Join*. Only the Ray backend worked well with the data sets. Another observation was that Modin defaults to Pandas for *Sort* (ie. limited distributed scalability).

The one billion-row strong scaling timings show that *Cylon* shows better scalability compared to the rest. Dask & Spark Datasets show commendable scalability for *Join* and *Sort*, however the former displays very limited scalability for *GroupBy*. A 100 million row test case (right column of Fig. 12) was performed to investigate Dask & Spark further. This constitutes a communication-bound operation because the partition sizes are smaller. This reduces the computation complexity, however, these smaller partitions need to be communicated across the same number of workers. Under these circumstances, both Dask and Spark diverge significantly at higher parallelisms, indicating limitations in their communication implementations. There was a consistent anomaly in Spark timings for 8–32 parallelism. We hope to investigate this further with the help of the Spark community.

We also observe that the serial performance of *Cylon* outperforms the rest consistently, which could be directly related to *Cylon*'s C++ implementation and the use of Apache Arrow format. At every parallelism, *Cylon* distributed performance is $2-4\times$ higher than Dask/Spark consistently. These results confirm the efficacy of the proposed distributed execution model in this paper.

7. Related work

In a previous publication, we proposed a formal framework for designing and developing high-performance data engineering frameworks that include data structures, architectures, and program models [43]. Kamburugamuve et al. proposed a similar big

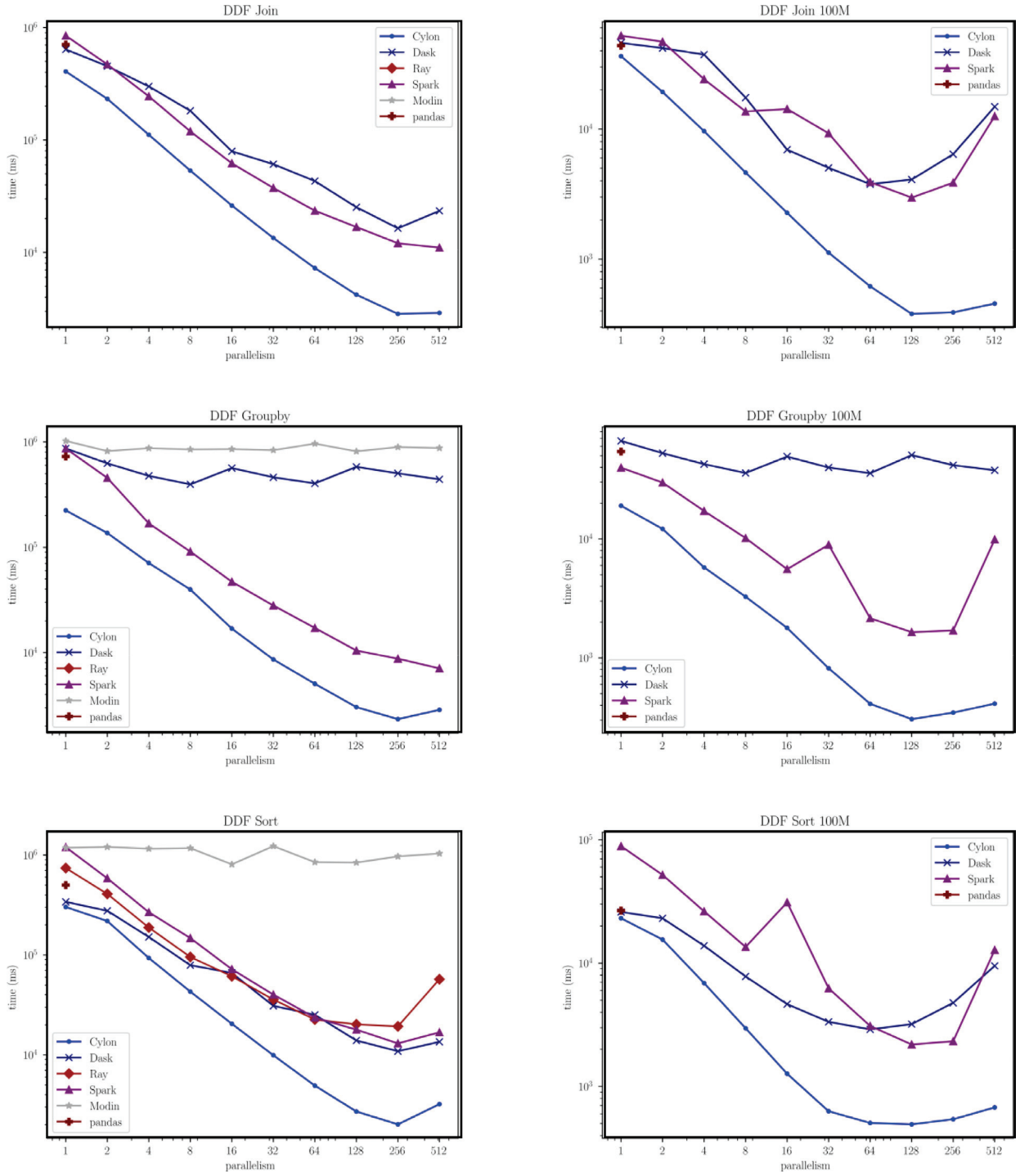


Fig. 12. Strong scaling of distributed dataframe operators (Log–Log), Left: 1B rows, Right: 100M rows (Only *Cylon*, Dask, & Spark).

data toolkit named *Twister2* [44], 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 the native performance of hardware and provides a more robust integration to Python and R.

In parallel to *Cylon*, Totoni et al. also suggested a similar HP-DDF runtime named *HiFrames* [45]. 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 high-performance distributed dataframe system available at the moment.

Dask [46,47] is one of the pioneering distributed dataframe implementations out there. It provides a Pandas-like API and is built on top of the Dask distributed execution environment. CuDF [12] extends this implementation in Dask-CuDF to provide distributed dataframe capabilities in Nvidia GPUs. Modin [25,48] is another dataframe implementation built on top of Dask and

Ray. It provides an API identical to Pandas so that existing applications can be easily ported to a distributed execution. Apache Spark [4,49] also provides a Pandas-like DDF named *Pandas on Spark*.

In addition to these systems, we would also like to recognize some exciting new projects. Velox is a C++ vectorized database acceleration library managed by the Meta Inc. incubator [50]. Currently, it does not provide a DF abstraction, but still offers most of the operators shown in Fig. 6. Photon is another C++-based vectorized query engine developed by Databricks [51] 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 [52].

8. Limitations and future work

Cylon currently covers about 30% of the Pandas API, and more distributed operators are being added, significantly, *Window* operators. Furthermore, the cost model for evaluating dataframe operator patterns has allowed us to identify areas of improvement. For example, communication operations could be improved by introducing algorithms that have lower latency costs.

Additionally, in Section 6.1 we saw significant time being spent on communication. These observations can be further analyzed using MPI profiler tools (eg. TAU - Tuning and Analysis Utilities, LLNL mpiP, etc.) and distributed debugging tools (eg. Arm/Linaro DDT, etc.). Some of these tools are available in the Summit supercomputer, which could give an in-depth look at the communication bottlenecks. 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*.

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

9. Conclusion

We recognize that today's data science communication operations could be improved by introducing algorithms that have lower latency costs. The data science community requires scalable solutions to meet its ever-growing data demand. Dataframes are at the heart of such applications, and in this paper, we discussed a cost model for evaluating the performance of distributed dataframe operator patterns introduced in our prior publication [1]. We also extended the execution model described in the previous work, by introducing a communication model. With these additions, we strongly believe we have presented a comprehensive execution model for distributed dataframe operators in distributed memory environments. Additionally, we presented *Cylon*, a reference runtime developed based on these concepts. We use the proposed model to analyze the communication and computation performance and identify bottlenecks and areas of improvement. We also showcased the importance of this work by conducting large-scale experiments on the ORNL Summit supercomputer where it showed admirable scalability in both strong and weak scaling experiments. *Cylon* also showed superior scalability compared to the state-of-the-art distributed dataframe systems, which further substantiates the effectiveness of the execution model presented in this paper.

CRediT authorship contribution statement

Niranda Perera: Conceptualization, Methodology, Software, Formal analysis, Writing – original draft, Investigation. **Arup Kumar Sarker:** Software, Writing – review & editing. **Mills Staylor:** Software, Writing – review & editing. **Gregor von Laszewski:** Software, Investigation, Writing – review & editing. **Kaiying Shan:** Software, Writing – review & editing. **Supun Kamburugamuve:** Conceptualization, Methodology, Software. **Chathura Widanage:** Conceptualization, Methodology, Software. **Vibhatha Abeykoon:** Conceptualization, Methodology, Software. **Thejaka Amila Kanewela:** Conceptualization, Methodology. **Geoffrey Fox:** Conceptualization, Supervision, Funding acquisition, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data used are synthetically generated. The code for the experiments is provided in the text.

Acknowledgments

We gratefully acknowledge the support of NSF grants 2210266 (CINES) and 1918626 (GPCE).

References

[1] Niranda Perera, Supun Kamburugamuve, Chathura Widanage, Vibhatha Abeykoon, Ahmet Uyar, Kaiying Shan, Hasara Maithree, Damitha Lenadora, Thejaka Amila Kanewala, Geoffrey Fox, High performance dataframes from parallel processing patterns, 2022, arXiv preprint arXiv:2209.06146.
[2] Apache hadoop, https://hadoop.apache.org/.
[3] Jeffrey Dean, Sanjay Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[4] Apache Spark™ - Unified Engine for large-scale data analytics, https://spark.apache.org/.
[5] Apache flink: Stateful computations over data streams, https://flink.apache.org/.
[6] Wes McKinney, et al., Pandas: A foundational python library for data analysis and statistics, Python High Perform. Sci. Comput. 14 (9) (2011) 1–9.
[7] Leslie G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[8] GC Fox, M Johnson, G Lyzenga, S Otto, J Salmon, D Walker, Richard L White, Solving problems on concurrent processors vol. 1: general techniques and regular problems, Comput. Phys. 3 (1) (1989) 83–84.
[9] Cylondata, Cylon, https://github.com/cylondata/cylon.
[10] Hao Gao, Nikolay Sakharnykh, Scaling joins to a thousand GPUs, in: 12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS @ VLDB, 2021.

[11] Chathura Widanage, Niranda Perera, Vibhatha Abeykoon, Supun Kamburugamuve, Thejaka Amila Kanewala, Hasara Maithree, Pulasthi Wickramasinghe, Ahmet Uyar, Gurhan Gunduz, Geoffrey Fox, High performance data engineering everywhere, in: 2020 IEEE International Conference on Smart Data Services, (SMDS), IEEE, 2020, pp. 122–132.
[12] rapidsai/cudf: cuDF - GPU dataframe library, https://github.com/rapidsai/cudf.
[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al., Ray: a distributed framework for emerging {AI} applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 18), 2018, pp. 561–577.
[14] Shuffling for groupby and join — Dask documentation, https://docs.dask.org/en/stable/dataframe-groupby.html.
[15] Performance tips and tuning — Ray 2.0.0, https://docs.ray.io/en/latest/data/performance-tips.html.
[16] 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., UCX: An open source framework for HPC network APIs and beyond, in: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, IEEE, 2015, pp. 40–43.
[17] Open MPI: Open source high performance computing, https://www.open-mpi.org/.
[18] facebookincubator/gloo: Collective communications library with various primitives for multi-machine training, https://github.com/facebookincubator/gloo.
[19] PMIx | process management interface - Exascale copyright 2017-2020 PMIx community, https://pmix.github.io/.
[20] Rajeev Thakur, Rolf Rabenseifner, William Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.
[21] Jesper Larsson Träff, Antoine Rougier, Sascha Hunold, Implementing a classic: zero-copy all-to-all communication with MPI datatypes, in: Proceedings of the 28th ACM international conference on Supercomputing, 2014, pp. 135–144.
[22] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, Derrick Weathersby, Efficient algorithms for all-to-all communications in multiport message passing systems, IEEE Trans. Parallel Distrib. Comput. 8 (11) (1997) 1143–1156.
[23] Mohak Shroff, Robert A. Van De Geijn, Collmark: MPI collective communication benchmark, in: International Conference on Supercomputing, Citeseer, 2000, p. 10.
[24] Rolf Rabenseifner, Optimization of collective reduction operations, in: International Conference on Computational Science, Springer, 2004, pp. 1–9.
[25] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, Aditya Parameswaran, Towards scalable dataframe systems, 2020, arXiv preprint arXiv:2001.00888.
[26] Roger W. Hockney, The communication challenge for MPP: intel paragon and meiko CS-2, Parallel Comput. 20 (3) (1994) 389–398.
[27] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, Thorsten Von Eicken, LogP: towards a realistic model of parallel computation, in: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, pp. 1–12.
[28] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, Chris Scheiman, Loggp: incorporating long messages into the LogP model for parallel computation, J. Parallel Distrib. Comput. 44 (1) (1997) 71–79.
[29] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, Jack J Dongarra, Performance analysis of MPI collective operations, Cluster Comput. 10 (2) (2007) 127–143.
[30] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, Hanmao Shi, On the versatility of parallel sorting by regular sampling, Parallel Comput. 19 (10) (1993) 1079–1103.
[31] Niranda Perera, Vibhatha Abeykoon, Chathura Widanage, Supun Kamburugamuve, Thejaka Amila Kanewala, Pulasthi Wickramasinghe, Ahmet Uyar, Hasara Maithree, Damitha Lenadora, Geoffrey Fox, A fast, scalable, universal approach for distributed data reductions, in: International Workshop on Big Data Reduction, IEEE Big Data, 2020.
[32] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, Torsten Hoefler, Distributed join algorithms on thousands of cores, Proc. VLDB Endow. 10 (5) (2017) 517–528.
[33] Summit user guide - OLCF user documentation, https://docs.olcf.ornl.gov/.
[34] Conda - conda documentation, https://docs.conda.io/.
[35] PyPI - The python package index, https://pypi.org/.
[36] IBM, IBM spectrum MPI - Overview, https://www.ibm.com/products/spectrum-mpi.
[37] pandas - Python data analysis library, https://pandas.pydata.org/.
[38] Niranda Perera, Supun Kamburugamuve, Chathura Widanage, Vibhatha Abeykoon, Ahmet Uyar, Kaiying Shan, Hasara Maithree, Damitha Lenadora, Thejaka Amila Kanewala, Geoffrey Fox, High performance dataframes from parallel processing patterns, in: Parallel Processing and Applied Mathematics: 14th International Conference, PPAM 2022, Gdansk, Poland, September 11–14, 2022, Revised Selected Papers, Part I, Springer, 2023, pp. 291–304.
[39] Dilshan Niranda Perera, Towards Scalable High Performance Data Engineering Systems (Ph.D. thesis), Indiana University, 2023.
[40] Niranda Perera, Kaiying Shan, Supun Kamburugamuwe, Thejaka Amila Kanewela, Chathura Widanage, Arup Sarker, Mills Staylor, Tianle Zhong, Vibhatha Abeykoon, Geoffrey Fox, Supercharging distributed computing environments for high performance data engineering, 2023, arXiv preprint arXiv:2301.07896.
[41] TPC-Homepage, https://www.tpc.org/default5.asp.
[42] NVIDIA, Optimizing the deep learning recommendation model on NVIDIA GPUs, https://developer.nvidia.com/blog/optimizing-dlrm-on-nvidia-gpus/.
[43] Supun Kamburugamuve, Chathura Widanage, Niranda Perera, Vibhatha Abeykoon, Ahmet Uyar, Thejaka Amila Kanewala, Gregor Von Laszewski, Geoffrey Fox, Hptmt: operator-based architecture for scalable high-performance data-intensive frameworks, in: 2021 IEEE 14th International Conference on Cloud Computing, (CLOUD), IEEE, 2021, pp. 228–239.
[44] Supun Kamburugamuve, Kannan Govindarajan, Pulasthi Wickramasinghe, Vibhatha Abeykoon, Geoffrey Fox, Twister2: design of a big data toolkit, Concurr. Comput.: Pract. Exper. 32 (3) (2020) e5189.
[45] Ehsan Totoni, Wajih Ul Hassan, Todd A Anderson, Tatiana Shpeisman, Hiframes: high performance data frames in a scripting language, 2017, arXiv preprint arXiv:1704.02341.
[46] Dask | Scale the python tools you love, https://www.dask.org/.
[47] Matthew Rocklin, Dask: parallel computation with blocked algorithms and task scheduling, in: Proceedings of the 14th python in science conference, Vol. 130, Citeseer, 2015, p. 136.
[48] Modin, Scale your pandas workflow by changing a single line of code — Modin 0.18.0 documentation, https://modin.readthedocs.io/en/stable/.
[49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, Ion Stoica, Resilient distributed datasets: A {*Fault — Tolerant*} abstraction for {*In — Memory*} cluster computing, in: 9th USENIX Symposium on Networked Systems Design and Implementation, (NSDI 12), 2012, pp. 15–28.
[50] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, Biswapesh Chattopadhyay, Velox: Meta's unified execution engine.
[51] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al., Photon: A fast query engine for lakehouse systems, in: Proceedings Of The 2022 International Conference On Management Of Data, 2022, pp. 2326–2339.
[52] substrait-io/substrait: A cross platform way to express data transformation, relational algebra, standardized record expression and plans, https://github.com/substrait-io/substrait.

Niranda Perera was born in Colombo, Sri Lanka, on January 10, 1990. He attended Nalanda College, Colombo, for his primary and secondary education and graduated with exemplary results in GCE Advanced Level Examinations from the Physical Science stream. The most outstanding student of the year, the best result in the Science section, and the best result in the Physical Science stream were some of the notable accolades awarded to him by the college in 2008.

In 2009, Niranda entered the University of Moratuwa for his undergraduate studies. In 2014 he received the degree of Bachelor of Science in Electronic and Telecommunication Engineering with honors and a First Class. Niranda also obtained the Chartered Institute of Management Accountants (CIMA), UK professional qualification, alongside his bachelor's.

Upon graduation in 2014, Niranda joined WSO2 Inc, Colombo as a software engineer, developing enterprise middleware solutions for batch, streaming, & predictive data analytics. In 2017, he joined the University of Moratuwa again as a research assistant in the Computer Science and Engineering Department. There he developed a cloud-based weather modeling framework for the Center for Flood Control and Water Management, Sri Lanka.

In 2018, Niranda received a scholarship for doctoral studies at Indiana University (IU), Bloomington, USA. There, he received his M.Sc. and Ph.D. in Intelligent Systems Engineering. He defended his doctoral thesis, 'Towards Scalable High Performance Data Engineering Systems' in 2023, under the mentorship of Prof. Geoffrey Fox at the at the IU Digital Science Center.