frontiers in Big Data

ORIGINAL RESEARCH
published: 07 February 2022
doi: 10.3389/fdata.2021.756041

HPTMT Parallel Operators for High Performance Data Science and Data Engineering

Vibhatha Abeykoon[1], Supun Kamburugamuve[2], Chathura Widanage[2], Niranda Perera[2], Ahmet Uyar[2], Thejaka Amila Kanewala[1], Gregor von Laszewski[3] and Geoffrey Fox[3,4]*

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

Data-intensive applications are becoming commonplace in all science disciplines. They are comprised of a rich set of sub-domains such as data engineering, deep learning, and machine learning. These applications are built around efficient data abstractions and operators that suit the applications of different domains. Often lack of a clear definition of data structures and operators in the field has led to other implementations that do not work well together. The *HPTMT* architecture that we proposed recently, identifies a set of data structures, operators, and an execution model for creating rich data applications that links all aspects of data engineering and data science together efficiently. This paper elaborates and illustrates this architecture using an end-to-end application with deep learning and data engineering parts working together. Our analysis show that the proposed system architecture is better suited for high performance computing environments compared to the current big data processing systems. Furthermore our proposed system emphasizes the importance of efficient compact data structures such as Apache Arrow tabular data representation defined for high performance. Thus the system integration we proposed scales a sequential computation to a distributed computation retaining optimum performance along with highly usable application programming interface.

Keywords: exascale and HPC systems, cylon, parallel computation, deep learning, Apache software foundation

OPEN ACCESS

Edited by:
Domenico Talia,
University of Calabria, Italy

Reviewed by:
Giovanni Ponti,
Italian National Agency for New Technologies, Energy and Sustainable Economic Development (ENEA), Italy
Loris Belcastro,
University of Calabria, Italy
Patrizio Dazzi,
National Research Council (CNR), Italy

****Correspondence:***
Vibhatha Abeykoon
vibhatha@gmail.com

Specialty section:
This article was submitted to Data Mining and Management, a section of the journal Frontiers in Big Data

Received: *09 August 2021*
Accepted: *29 November 2021*
Published: *07 February 2022*

Citation:
Abeykoon V, Kamburugamuve S, Widanage C, Perera N, Uyar A, Kanewala TA, von Laszewski G and Fox G (2022) HPTMT *Parallel Operators for High Performance Data Science and Data Engineering. Front. Big Data 4:756041. doi: 10.3389/fdata.2021.756041*

1 INTRODUCTION

Data engineering and data science are two major branches of data-intensive applications. Data engineering deals with collecting, storing, and transforming data. Data science tasks (deep learning, machine learning, data engineering) comprises of several disciplines, out of them machine learning and deep learning are significant. This is the place where we use data to learn and gain insights. These two components, illustrated in **Figure 1** are designed on top of data structures and operators around them. The data engineering component primarily works with table data abstractions, while the machine learning and deep learning components mainly use tensors and matrices.

To run applications using multiple computers, we can partition the data and apply distributed operators. Current systems use several different strategies to provide distributed application programming interfaces (APIs) for data-intensive applications. An API for data-intensive applications is a combination of data structures, operators, and an execution model. There are thousands of operators defined around data structures such as vectors and tables by different frameworks. The current data systems use asynchronous and loosely synchronous execution models

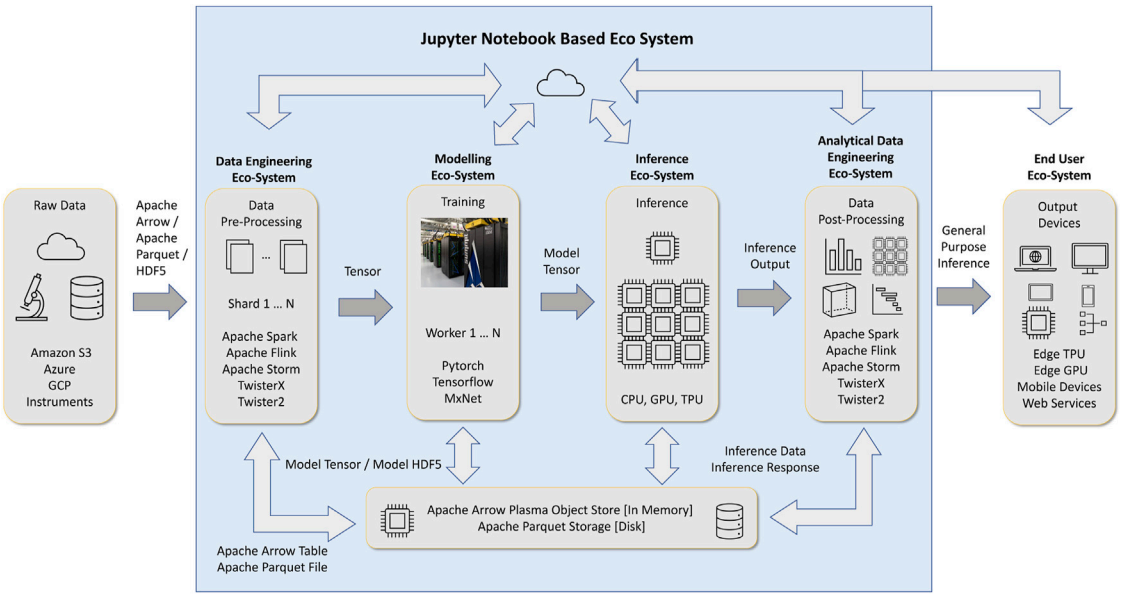

FIGURE 1 | Data science workflow with Jupyter Notebook interface and Data Engineering around Deep Learning.

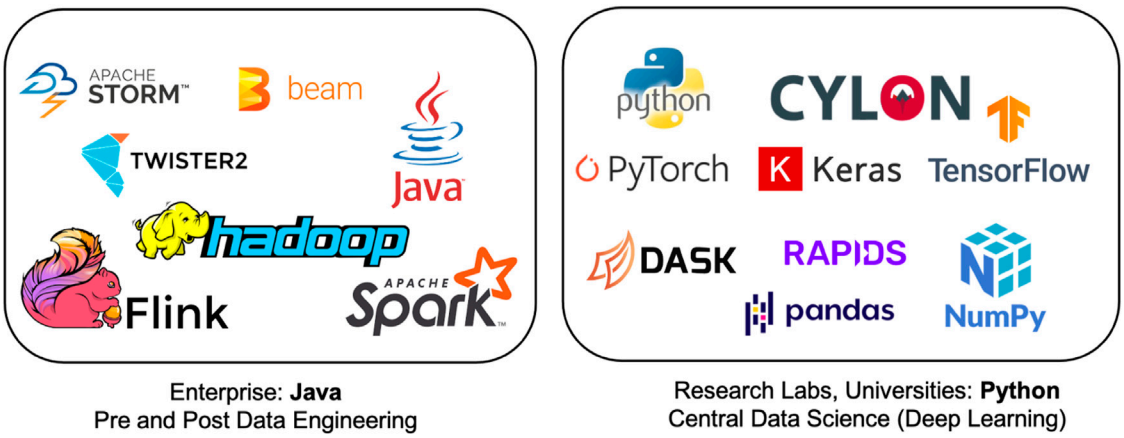

FIGURE 2 | The goal of HPTMT to achieve High Performance in each ecosystem and high-performance integration between ecosystems.

for running programs at scale. Asynchronous execution is popular in systems such as Spark (Zaharia et al., 2010), Dask (Rocklin, 2015) and Modin (Petersohn et al., 2020). Loosely synchronous distributed execution is used in systems such as PyTorch (Paszke et al., 2019), Cylon (Widanage et al., 2020) and Twister2 (Fox, 2017).

In a previous paper (Kamburugamuve et al., 2021), the authors proposed the *HPTMT* (High-Performance Tensors, Matrices, and Tables); an operator-based architecture for data-intensive applications as a scalable and interoperable way for designing rich data-intensive applications. With *HPTMT* we focus, as depicted in **Figure 2**, on the interoperability of distributed operators and how one can build large-scale applications using different data abstractions. **Figure 2** contains two aspects of the data analytics. One is the data processing which includes data loading, data cleaning, feature engineering, etc. which are the main steps followed in obtaining a final dataset which is ready for mathematical evaluations. The big data systems such as Apache Spark, Apache Flink, Apache Storm, are written on Java language. The other aspect is the mathematical computations required to process the data further. Majority of these data structures are fully or partially represented in terms of dataframes (set of arrays: Dask, Pandas) and separate arrays (Numpy, Tensors). Frameworks like Numpy, Dask, Pandas are written on Python to provide much easier access to data scientists to write analytical programs without concerning about the

underlying computation models. The likes of Dask and Cylon further enhances the ability to such computations in parallel to support computation intensive jobs. On top of these computations systems such as PyTorch and Tensorflow allows to run complex mathematical models based on machine learning or deep learning algorithms.

This paper will showcase the importance of *HPTMT* architecture through an application that uses various data abstractions in a single distributed environment to compose a rich application. It highlights the scalability of the architecture and its applicability to high-performance computing systems.

The rest of the paper is organized as follows. **Section 2** gives an overview of the *HPTMT* architecture. **Section 3** describes the distributed execution of various frameworks and how they can work together according to the *HPTMT*. **Section 4** describes an end-to-end application while **section 5** highlights the performance. In **section 6** we describe related work and conclude in **section 8**.

2 *HPTMT* ARCHITECTURE

HPTMT architecture defines an operator model along with an execution model for scaling data-intensive applications. The primary goal of *HPTMT* is the efficient composability of distributed operators around different data structures to define complex data engineering applications. We see this architecture as a good candidate for exascale software environments. Its simple premise—*put the parallelism into interoperable libraries* seems practical to implement well on heterogeneous collections of accelerators and CPUs. Note that one of the most widely adopted approaches to parallel computing is the use of runtime libraries of well-implemented parallel operations. This was a key tenet of frameworks such as, High-Performance Fortran HPF (Dongarra et al., 2003) and related parallel environments [HPJava (Carpenter et al., 1998), HPC++ (Johnson and Gannon, 1997), Chapel (Chamberlain et al., 2007), Fortress (Allen et al., 2005), X10 (Charles et al., 2005), Habanero-Java (Imam and Sarkar, 2014)]. Even though these frameworks became popular in parallel computing/HPC, they had limited success in data engineering. We believe that a major reason behind this is, the lack of well-defined data engineering operators. Historically, such systems were used in sophisticated computational science simulations with large linear algebra operators (ex: BLAS routines). *HPTMT* attempts to bridge this gap between HPC and data-intensive applications by providing a set of well-defined data engineering operators with highly scalable execution model.

2.1 Principles

HPTMT architecture defines several core principles for a framework to be compatible with it. These are summarized below and more details can be found in the paper (Kamburugamuve et al., 2021).

- Use of multiple data abstractions (Tensors, Matrices, Tables) and operators around them that are suitable for each class of applications.
- Loosely Synchronous Execution - In an asynchronous framework, operators and the scheduler are coupled making it harder work across different systems.
- Operators should be independent of the parallel execution environment—A parallel environment manages the processes and various resources required by operators, such as the network. If the implementation of operators is coupled to the execution environment, we can only use the operators specifically designed for it.
- Same operator on different hardware—The same operator can be implemented on GPUs, CPUs or FPGA (Field Programmable Gate Arrays). Also, they should be able to use different networking technologies such as Ethernet and InfiniBand.

2.2 Operators

An application domain such as deep learning or data engineering comprises of a combination of operators to build the total job. Based on the data distribution, these operators can be categorized into two groups, namely, local operators (single machine) and distributed operators (across multiple machines). Some operators are purely local or purely distributed, and some can be either. A local operator only works with a single piece of data inside the memory of a single node in a cluster. They give rise to what is called embarrassingly or pleasingly parallel models for distributed execution. Operator based methods are not just used to support parallelism but have several other valuable capabilities.

- Allow interpreted languages to be efficient as overhead is amortized over the execution of a (typically large) operation
- Support mixed language environments where invoking language (e.g., Python) is distinct from the language that implements the operator (e.g., C++)
- Support proxy models where user programs in an environment that runs not just in a different language but also on a different computing system from the executing operators. This includes the important case where the execution system includes GPUs and other accelerators.
- Support excellent performance even in non-parallel environments. This is the case for Numpy and Pandas operators.

Recent frameworks such as Apache Arrow (Apache Arrow, 2021, Apache Software Foundation, Accessed 2021/Aug), and Parquet (Apache Parquet, 2021, Apache Software Foundation, Accessed 2021/Aug) provide essential tools which are crucial to our approach to *HPTMT*, and they (or equivalent technologies) are vital for any high-performance multi-language multi-operator class system. They provide efficient language-agnostic column storage for Tables and Tensors that allows vectorization for efficiency and performance. Note that distributed parallel computing performance is typically achieved by decomposing the rows of a table across multiple processors. Then within a processor, columns can be vectorized. This, of course, requires a large amount of data so that each processor has a big enough workflow to process efficiently. It is a well-established principle

TABLE 1 | Sample set of tensor operations as specified by PyTorch.

Operation class	Description
Create	Create tensors from files, in-memory data or other data structures such as NumPy
Math	Multiplication, addition
Statistics	Statistical function such as mean, median, std
Indexing	Different methods to access values of tensors
Conversion	Convert a tensor to another format such as NumPy or change the shape of a tensor

TABLE 2 | Operators on tables.

Operator	Description
Select	Filters out some records based on the value of one or more columns
Project	Creates a different view of the table by dropping some of the columns
Union	Applicable on two tables having similar schemas to keep all the records from both tables and remove the duplicates
Cartesian Product	Applicable on two tables having similar schemas to keep the set of all possible record pairs that are present in both tables
Difference	Retains all the records of the first table, while removing the matching records present in the second table
Intersect	Applicable on two tables having similar schemas to keep only the records that are present in both tables
Join	Combines two tables based on the values of columns. Includes variations Left, Right, Full, Outer, and Inner joins
OrderBy	Sorts the records of the table based on a specified column
Aggregate	Performs a calculation on a set of values (records) and outputs a single value (record). Common aggregations include summation and multiplication
GroupBy	Groups the data using the given columns; GroupBy is usually followed by aggregate operations

that the problem needs to be large enough for the success of parallel computing (Fox et al., 1994), which the latest Big Data trends also follow. Note that the most compelling parallel algorithms use block (i.e., row and column) decompositions in scientific computing to minimize communication/compute ratios. Such block decompositions can be used in Big Data (Huai et al., 2014) (i.e. table data structures), but could be less natural due to the heterogeneous data within it.

For Big Data problems, individual operators are sufficiently computationally intensive to consider the basic job components as parallel operator invocations. Any given problem typically involves the composition of multiple operators into an analytics pipeline or more complex topology. Each node of the workflow may run in parallel. This can be efficiently and elegantly implemented using workflow such as Parsl (Babuji et al., 2019), Swift (Wilde et al., 2011), Pegasus (Deelman et al., 2015), Argo (Argo Home Page, 2021 https://argoproj.github.io/argo-workflows/, Accessed 2021/Aug), Kubeflow (Kubeflow, 2021 home page https://www.kubeflow.org/, 2021), Kubernetes (Burns et al., 2016) or dataflow (Spark, Flink, Twister2) preserving the parallelism of *HPTMT*.

2.2.1 Categorizing Operators

There are thousands of operators defined for arrays, tensors, tables, and matrices. Note that tensors are similar to arrays but have an important deep learning utility. Matrices are similar to arrays and tensors but typically two dimensional. Tables (and dataframes) are characterized by entries of heterogeneous types. This is widely used in databases where the different columns can have strings to dates to numbers. **Table 1** shows some common operator categories for tensors as defined by PyTorch, Tensorflow or Keras. These deep learning frameworks define over 700 operators on tensors. Numpy lists 1085 array operations. **Table 2** shows some of the popular operations on tables, where the Python Pandas library has around 224 dataframe operators out of a listed total of 4782. Also, optimized linear algebra operators are used internally in most widely used math and tensor compute libraries. **Table 3** contains a classification of BLAS operators, which are local or distributed. The (old but standard) library SCALAPACK has 320 functions (operators) at a given precision and a total of over one thousand.

2.2.2 Distributed Operators

A distributed operator works across data in multiple processes in many nodes of a cluster. A distributed operator needs communication options and local operators. Compared to the number of local operators defined on a data structure, there are a limited set of communication operators for a given data structure, and some of them are listed in **Table 4** where 720 MPI operators support classic parallel computing. Higher-level distributed operations are built by combining these communication operations with local operations, as shown in **Table 5**. These include the famous MapReduce (Dean and Ghemawat, 2008) which abstraction showed clearly the similarity between distributed operators in the technical and database computing domains. MapReduce and its implementation in Hadoop enabled parallel databases as in Apache Hive. They added Group-By and key-value pairs to the Reduce operation common in the previous HPF family simulation applications. The powerful yet straightforward MapReduce operation was expanded in Big Data systems, primarily through the operators of Databases (union, join, etc.), Pandas, and the Spark, Flink, Twister2 family of systems.

TABLE 3 | Operations as specified by BLAS.

Operation	Description
Level 1	Operations on vectors i.e., adding two vectors
Level 2	Operations for combination of vectors and matrices. i.e., matrix and vector multiplication
Level 3	Matrix operations i.e., matrix and matrix multiplication

TABLE 4 | Communication operations for data structures.

Data structure	Operations
Arrays	Reduce, AllReduce, Gather, AllGather, Scatter, AllToAll, Broadcast, Point-to-Point
Tables	Shuffle (Similar to AllToAll but specifically designed for Tables), Broadcast, All-gather

TABLE 5 | Higher level distributed operations.

Distributed operation	Implementation
Sorting tables	Shuffle followed by a local sorting operation
Join tables	Partitioning of records, shuffle and local join operation
Matrix multiplication	Point to point communication and local multiplication
Vector addition	AllReduce with SUM

2.3 Distributed Execution

There are two main distributed execution methods used in current systems. They are fully asynchronous execution and loosely synchronous execution (Fox, 1989; Valiant, 1990). In an asynchronous system, the parallel task instances can execute independently using task queues to decouple them in time. This is seen in systems like Spark, Dask, and Hadoop. In a loosely synchronous system, the parallel tasks assume they can directly send messages to other similar jobs. It is called loosely synchronous because synchronization only happens when they need to communicate with each other. Otherwise, parallel task instances can work independently. This makes loosely synchronous applications highly scalable and more performant.

The asynchronous execution demands the system to be tightly integrated with a central coordinator and a scheduler. It may also employ "mail-boxes" or shared storage to fully decouple each task in the execution (this is important because, there may be in-flight messages and the corresponding receiver task would consume them at a later stage). While this model allows features such as fault-tolerance, dynamic resource allocation, effective usage of compute resources, it is susceptible to scheduler overheads, message passing delays, etc. Thereby, the async model incurs a performance penalty, and also makes it harder to develop distributed operators independently and make them work together.

We observe that the current technology and hardware advancements provide more reliable, highly available compute resources with faster networks. And we believe that these trends enable loosely synchronous execution in modern computing environments, and thereby develop high performance and highly scalable data engineering applications. We are seeing this trend being employed successfully in similar complementary domains, such as distributed data parallel deep learning (PyTorch, Tensorflow, Horovod, etc). Therefore, *HPTMT* architecture embraces this execution model, and attempts to broaden horizons of data engineering and data science in terms of performance and scalability.

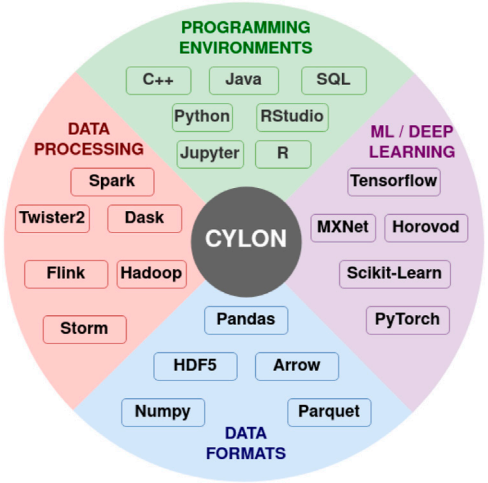

FIGURE 3 | Cylon for data engineering. Reproduced with permission from Widanage et al. (2020).

3 *HPTMT* FRAMEWORKS

Now let us look at Cylon and Deep Learning frameworks and see how they can work together according to the *HPTMT* architecture. First, we describe how Cylon is designed to support distributed data engineering on a dataframe abstraction. Then we discuss how Cylon can be coupled with state-of-the-art deep learning frameworks to organise end-to-end data analytics workloads.

3.1 Cylon

Cylon (Abeykoon et al., 2020; Widanage et al., 2020) provides a distributed memory DataFrame API on Python for processing data using a tabular format. Cylon provides a Python API around high-performance compute kernels in C++. These kernels are written on top of the Apache Arrow based efficient in-memory table representation. It can be deployed with MPI for distributed memory computations processing large datasets in HPC clusters. Operators in Cylon are based on relational algebra and closely resemble the operators in Pandas DataFrame to provide a consistent experience. The user can program with a global view of data by applying operations on them. Also, they can convert the data to local parallel processes and do in-memory operations as well. Cylon can be thought of as a framework that can work across different frameworks, data formats to connect various applications, as shown in **Figure 3**.

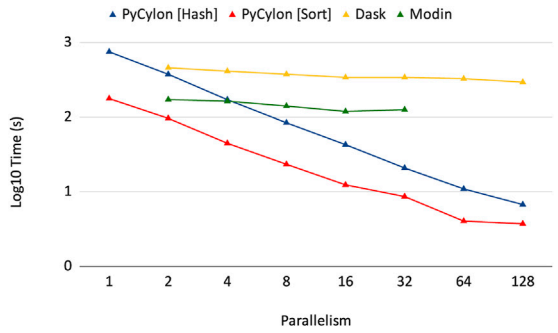

FIGURE 4 | Distributed join performance.

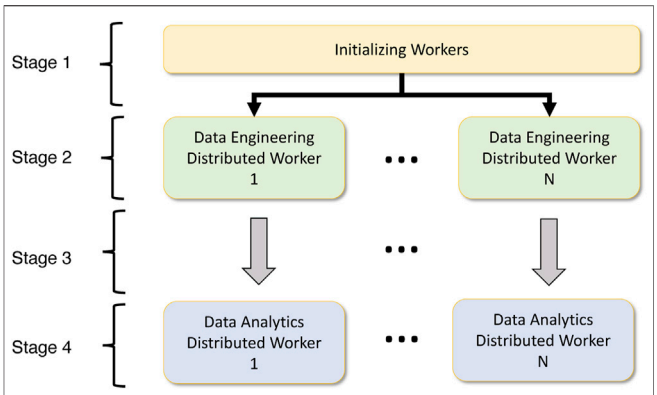

FIGURE 5 | Integrating data engineering workload with data analytics workload.

Cylon is different from other table abstractions such as Modin (Petersohn et al., 2020), Dask (Rocklin, 2015) and Spark Zaharia et al. (2010) because it supports an efficient loosely synchronous execution model. These other frameworks use the asynchronous execution model, which relies on a central scheduler and a coordinator and does not conform to the *HPTMT* architecture. **Figure 4** shows how the Cylon Join operator performs compared to other frameworks. This experiment used 200M records per relation (for both left and right tables in a join) and scaled up to 128 processes. Random data were generated by considering the uniqueness of data to be 10% such that the join performs under higher stress feeling hash functions and hash-based shuffles. In the parallel experiments, each process will be loading an equal amount of data such that the total amount is limited to 200M records. The results from **Figure 4** show that our distributed join implementation is faster than Dask and Modin implementations. Also, the scalability in Dask and Modin is not very strong compared to the scaling provided by PyCylon. Also, the Modin couldn't be scaled up beyond a single machine and failed in the execution.

3.2 Deep Learning Frameworks

Deep learning workloads are compute-intensive. Most of the existing deep learning frameworks can run codes in a distributed manner. Here, the widely used approach is the distributed data-parallel model. Distributed data-parallel model deals with the distributed memory architecture and has the loosely synchronous execution capability.

PyTorch offers a distributed data-parallel (DDP) model, which allows the user to train large models using many GPUs. It can use distributed frameworks such as MPI, NCCL, or GLOO for the necessary communication operations for deep learning training with multiple GPUs. Tensorflow does loosely synchronous distributed execution *via* frameworks like Horovod. Due to these reasons, we can think of these systems as *HPTMT* when running data-parallel training using the loosely synchronous execution model.

3.3 Deep Learning and Data Engineering

Because the distributed execution of Cylon and deep learning systems such as PyTorch and Tensor conform to the *HPTMT* architecture, they can work together in a single parallel program. This improves productivity and usability in dealing with end-to-end analytical problems. In a data analytics-aware data engineering workload, three main factors govern usability and performance.

- Single source, including data engineering and data analytics
- Simple execution mode for sequential and distributed computing
- Support for CPUs and GPUs for distributed execution

The single source refers to writing the data engineering and analytics code in a single script and executing with a single command. This is a beneficial and efficient method to do data exploration based data analytics. For such workloads, feature engineering and data engineering components are extensively modified to see how the data analytics workload performs for different settings. In such cases, the data scientist must have room to write the usual Python script and run the data analytics workload efficiently, not only in a single node but also across multiple nodes. Simple execution mode refers to running the workload with a simple method to spawn the processes to run in parallel.

Data analytics frameworks provide different methods to spawn parallel jobs. For instance, Dask requires that the user start the workers and schedulers on each node and provide host information for distributed communication. MPI allows for a single execution command *mpirun* to spawn all the processes. Such factors are essential in providing a unified interface to do deep learning easily. Also, the execution mode on various accelerators for deep understanding is a vital component. The majority of the frameworks support both CPU and GPU execution, so it is essential to provide the means to seamlessly integrate with these execution models to support data analytics workloads. Figure ?? highlights the high-level component overlay of a data analytics-aware data engineering workload. We have partitioned the workflow into four stages.

- Stage 1: In the first stage, the processes must be spawned depending on the parallelism. A unified process spawning mechanism that identifies worker information such as host

IP addresses for each machine or network information is identified at this stage.

- Stage 2: Worker information is extracted, and data engineering operators will run in distributed mode on top of the data engineering platform, which depends on the worker initialization component. Here the operations can be distributed or pleasingly parallel.
- Stage 3: For data analytics workloads, the worker information, network information, chosen accelerator, and data must be provided from the corresponding data engineering process. This mapping is 1:1 for data engineering workers to data analytics workers. But this can also be a many-to-many relationship.
- Stage 4: The worker information, network information and data will be used to execute the data analytics workload is distributed or pleasingly parallel mode.

Considering this generic overview on deploying deep learning workloads with data engineering workloads, we have integrated PyCylon with distributed data-parallel models for PyTorch, Horovod-PyTorch, and Horovod-Tensorflow. Horovod is a distributed deep learning framework that supports a unified API for handling distributed deep learning on multiple frameworks. Horovod supports PyTorch, Tensorflow, and MXNet. In our research, we paid close attention to PyTorch and Tensorflow. Horovod internally uses mpirun to spawn the processes, and this model fits very well with PyCylon internals as we relied on mpirun to spawn the processes. This makes PyCylon uniquely qualified as a supportive data engineering framework for Horovod.

The first step is to initialize the runtime. Here either PyTorch distributed initialization, or PyCylon distributed initialization can be called. But especially on CPUs, the PyTorch initialization must be called since PyTorch internally does not handle the MPI initialization check. But if we use NCCL as the back-end, this constraint does not exist. This is one of the bugs we discovered from our previous research. For the PyTorch DDP, the master address and port must be provided because the NCCL back-end needs to identify which work will be designated as the master-worker to coordinate the communication. In addition, the initialization method has to be set. After the distributed initialization in PyTorch, the PyCylon context must be initialized to set to distributed mode. After this stage, we complete the requirements for stage 1 and partial requirements for stage 3 (network information is also passed along with data in stage 3, which is initialized in this step). **Figure 1** is a sample code snippet related to the initialization step.

Listing 1. Stage 1: Initialization for PyTorch With PyCylon

```
os.environ['MASTER_ADDR'] = master_address
os.environ['MASTER_PORT'] = port
os.environ["LOCAL_RANK"] = str(rank)
os.environ["RANK"] = str(rank)
os.environ["WORLD_SIZE"] = str(world_size)
dist.init_process_group(backend=backend,
init_method="env://")
```

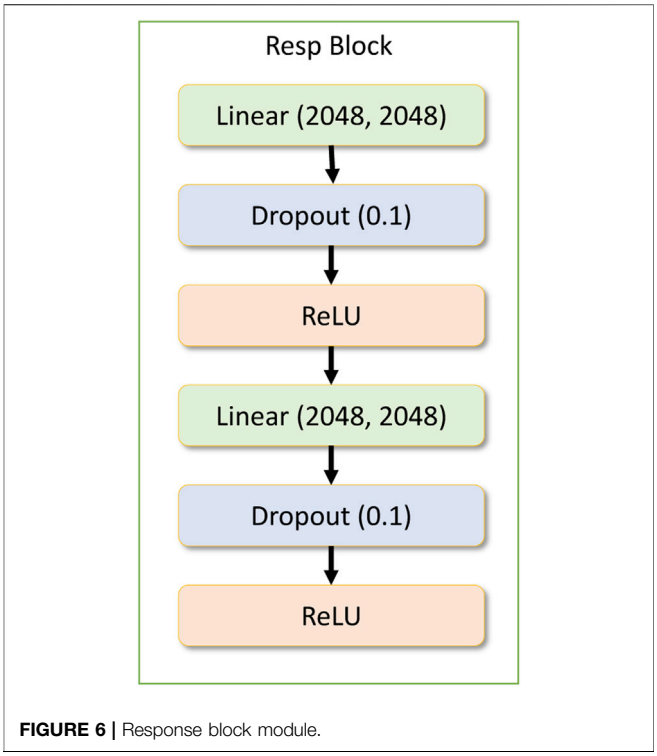

FIGURE 6 | Response block module.

```
mpi_config = MPIConfig()
env = CylonEnv(config=mpi_config, distributed
=True)
```

The data engineering workload is done in PyCylon, assuming the distributed mode initialization. We first join two tables and use the join response for a deep learning workload. The distributed join is called by providing the initialized context information to the join function. At the end of this stage, we create the resultant dataframe, and later on, in Stage 3, this dataframe can be used to generate the Numpy array required for deep learning. This stage is typical for any framework, including PyTorch, Tensorflow, etc. **Figure 2** details a sample data engineering workload for a data analytics problem.

Listing 2. Stage 2: PyCylon Data Engineering Workload

```
df1 = DataFrame(read_csv("..."))
df2 = DataFrame(read_csv("..."))
join_df = df1.merge(right=df2, left_on=[0],
right_on=[3], algorithm='hash')
```

In Stage 3, Stage 2 is used to create tensors required for the deep learning stage. We also perform the data partitioning for training and testing. This stage is different from framework to framework since the tensor creation and data partitioning steps can have various internal utils. We do not use data loaders or data samplers but note that these tools can be used to generate both. **Figure 3** is a sample code snippet for data movement from data engineering workload to data analytics workload.

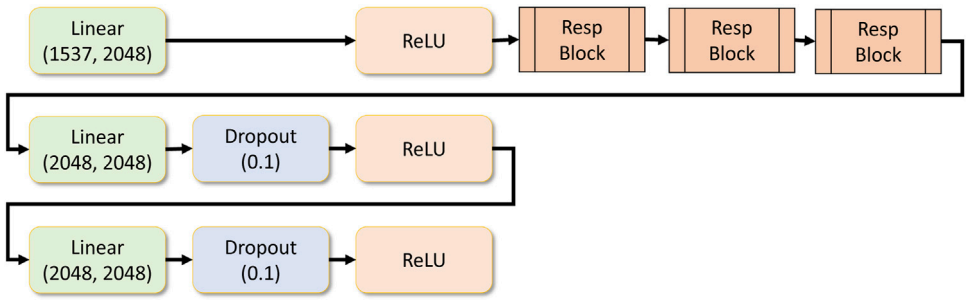

FIGURE 7 | Response network.

Listing 3. Stage 3: Moving Data from Data Engineering Workload to Data Analytics Workload

```
data_ar: np.ndarray = feature_df.to_numpy()
df_ftrs: np.ndarray = data_ar[:, 0:3]
df_lrnr: np.ndarray = data_ar[:, 3:4]
x_train, y_train = df_ftrs[0:100], df_lrnr
[0:100]
x_test, y_test = df_ftrs[100:], df_lrnr[100:]
...
x_train = torch.from_numpy(x_train).to(device)
y_train = torch.from_numpy(y_train).to(device)
x_test = torch.from_numpy(x_test).to(device)
y_test = torch.from_numpy(y_test).to(device)
```

In Stage 4, we initialize the deep learning model and the DDP model using the sequential model. We pass along device information such that tensors and models are copied to the corresponding devices (if accelerators are involved) for training and testing. This initialization part varies from framework to framework depending on the requirements and APIs. **Figure 4** highlights the initialization of a DDP model with PyTorch.

Listing 4. Stage 4: PyTorch Distributed Data Analytics Workload

```
model = Network().to(device)
ddp_model = DDP(model, device_ids=[device])
loss_fn = nn.MSELoss()
optimizer = optim.SGD(ddp_model.parameters(),
lr=0.01)
optimizer.zero_grad()
for t in range(epochs):
  for x_batch, y_batch in zip(x_train,
  y_train):
      prediction = ddp_model(x_batch)
      loss = loss_fn(prediction, y_batch)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()
```

3.3.1 Horovod With PyTorch

Horovod PyTorch provides the ability to scale on both GPUs and CPUs with a unified API. This is significant because

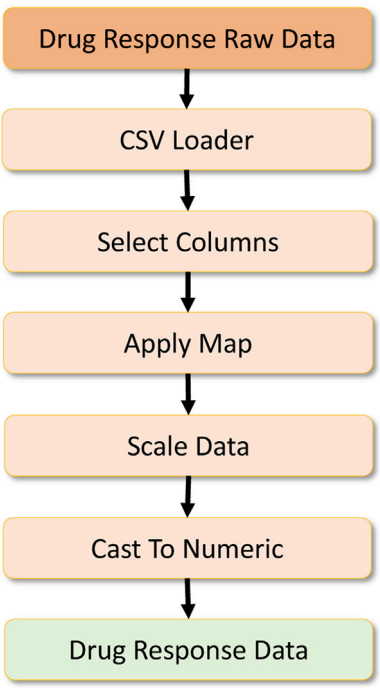

FIGURE 8 | Drug response data processing.

PyTorch does not need to be compiled from the source to get MPI capability. Horovod has already offloaded the distributed trainer, optimizer, and allreduce communication packages. The internal DDP mechanism that does this in PyTorch is offloaded.

In Stage 1, the Horovod init method must be called to initialize the environment. After that, the Cylon context can be initialized with distributed runtime true. If GPUs are used, the correct device must be set to PyTorch CUDA configs. To obtain the device IDs, we can either use the rank from Horovod initialization or PyCylon initialization. Still, at the moment, Horovod supports local rank as well, and it is more suitable in terms of effortlessly integrating with the distributed runtime for Horovod-PyTorch. **Figure 5** shows a sample code snippet demonstrating how this is accomplished.

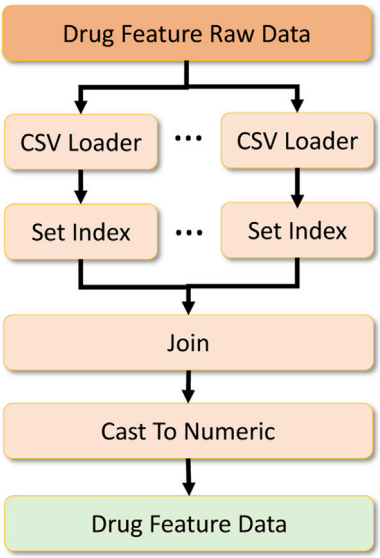

FIGURE 9 | Drug feature data processing.

Listing 5. Stage 1: Initialization for Horovod-PyTorch With PyCylon

```
hvd.init()
mpi_config = MPIConfig()
env = CylonEnv(config=mpi_config, distributed=
True)
rank = env.rank
cuda_available = torch.cuda.is_available()
device = 'cuda:' + str(rank) if cuda_
available else 'cpu'
if cuda_available:
   # Horovod: pin GPU to local rank.
   torch.cuda.set_device(hvd.local_rank())
   torch.cuda.manual_seed(42)
```

Another essential thing to note is that the data engineering code remains the same for any deep learning framework discussed in this context. Also, as with the PyTorch data engineering section, the output can be converted to a Numpy array using the endpoints from the PyCylon dataframe. Also, the tensors can be created by providing the device IDs obtained from the Horovod runtime, and data can be prepared for a deep learning workload.

In Stage 4, following the tensor creation step, the Horovod-related initialization must be done to prepare the optimizers, network and other utils for distributed training. PyTorch-Horovod integration, PyTorch's default neural network model, loss function, and optimizer can be used as input to the distributed computation-enabled Horovod components. First, the model parameters and optimizer must be broadcast using the Horovod broadcast method from 0^{th} rank. There are two method calls designated for initial network values and optimizer values. Also, Horovod provides a compression algorithm to select whether compression is required for distributed communication. After these steps, the distributed optimizer must be set by passing the initialized values. **Figure 6** includes a sample code snippet to initialize the Horovod components for distributed data-parallel deep learning with PyTorch.

Listing 6. Stage 4: Distributed Data Analytics PyTorch-Horovod Workload

```
optimizer = optim.SGD(...)
hvd.broadcast_parameters(model.state_dict(),
root_rank=0)
hvd.broadcast_optimizer_state(optimizer,
root_rank=0)
compression = hvd.Compression.fp16
model_ps = model.named_parameters()
optimizer = hvd.DistributedOptimizer(optimizer,
named_parameters=model_ps,
compression=compression,        op=hvd.Adasum,
gradient_predivide_factor=1.0)
```

3.3.2 Horovod With Tensorflow

Similar to PyTorch integration, Horovod also supports Tensorflow. Tensorflow has its own distributed training platform. It contains distributed mirrored strategy as the equivalent routine for distributed data-parallel training. To start this run, we initialize Horovod and PyCylon. As with PyTorch, we also need to decide how the device is selected depending on the accelerator. The Tensorflow config API provides a listing of GPUs, and this information is added to the Tensorflow configurations to make all the GPU devices available. **Figure 7** is a code snippet for the aforementioned initialization.

Listing 7. Stage 1: Initialization for Tensorflow With PyCylon

```
hvd.init()
assert hvd.mpi_threads_supported()
mpi_config = MPIConfig()
env       =       CylonEnv(config=mpi_config,
distributed=True)
rank = env.rank
world_size = env.world_size
gpus = tf.config.experimental.list_physical_
devices('GPU')
for gpu in gpus:
   tf.config.experimental.set_memory_growth
   (gpu, True)
if gpus:
  tf.config.experimental.set_visible_devices
  (gpus[hvd.local_rank()], 'GPU')
```

Similar to prior experience, the data engineering component also remains unchanged for Horovod-Tensorflow integration. The data analytics data structure creation is different from framework to framework. Tensorflow has its own set of APIs to make these steps simpler and more structured. The Tensorflow dataset API can be used to create tensors from Numpy arrays, and this API can be used to shuffle and create mini-batches, as expected by the deep learning workload. **Figure 8** contains a code snippet detailing this step.

Listing 8. Stage 3: Moving Data from Data Engineering Workload to Data Analytics Workload

```
...
train_dataset = tf.data.Dataset.from_tensor_
slices((x_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_
slices((x_test, y_test))
BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100
train_dataset = train_dataset.shuffle(SHUFFLE_
BUFFER_SIZE).batch(BATCH_SIZE)
test_dataset = test_dataset.batch(BATCH_SIZE)
...
```

Horovod-Tensorflow also requires a set of initialization steps to train a Tensorflow deep learning model. Like PyTorch, the Tensorflow loss function, optimization function and neural network model are compatible with Tensorflow-Horovod internals. The gradient tape from Tensorflow autograd can be used, and for this, Horovod provides a DistributedGradientTape operator, which takes the gradient tape instance as a parameter. In addition, before training, this DistributedGradientTape must be initialized with the model parameters and loss function, and the optimizer values must be set to initial values. Again, the model parameters and optimizer values must be broadcast using designated Horovod broadcast functions. **Figure 9** illustrates this.

Listing 9. Stage 4: Distributed Data Analytics Horovod-Tensorflow Workload

```
model = tf.keras.Sequential(...)
loss = tf.losses.MeanSquaredError()
opt = tf.optimizers.Adam(0.001 * hvd.size())
@tf.function
def training_step(images, labels, first_batch):
   with tf.GradientTape() as tape:
       probs = model(images, training=True)
       loss_value = loss(labels, probs)
   tape = hvd.DistributedGradientTape(tape)
   grads = tape.gradient(loss_value,
   model.trainable_variables)
   opt.apply_gradients(zip(grads, model.
   trainable_variables))
   if first_batch:
     hvd.broadcast_variables(model.
     variables, root_rank=0)
     hvd.broadcast_variables(opt.variables(),
     root_rank=0)
   return loss_value
```

4 UNOMT APPLICATION

To demonstrate an end-to-end *HPTMT* architecture, we implemented a scientific application with a workload containing data engineering and data science computations. Our objective is to showcase how a sequential workload can be designed in a distributed manner using PyCylon and run a deep learning workload seamlessly on only a single script with a unified runtime. For this, we selected an application that uses Pandas dataframe for data engineering and PyTorch for data analytics. The original application is sequentially executed, and we have implemented a parallel version of this application with PyCylon and distributed PyTorch.

4.1 Background

UNOMT application is part of CANDLE Wozniak et al. (2020), Xia et al. (2021) research conducted by Argonne National Laboratory, focusing on automated detection of tumour cells using a deep learning approach. The uniqueness of this approach is the composition of a data engineering workload followed by a deep learning workload written in PyTorch. This provides an ideal scientific experiment to showcase multiple systems working together to facilitate an efficient data pipeline. The goal of the UNOMT application is to give a cross-comparison of cancer studies and integrate it into a unified drug response model. Cell RNA sequences, drug descriptors and drug fingerprints are used as such responses to train the model.

In the deep learning component, multiple networks are involved working on small and large datasets in the training process. Our research focuses on the more extensive network designed to calculate the drug response based on the cell-line information.

4.2 Deep Learning Component

UNOMT refers to a unified deep learning model with multi-tasks to predict drug response as a function of tumour and drug features for personalized cancer treatment. Precision oncology focuses on providing medicines for specific characteristics of a patient's tumour. The drug sensitivity is quantified by drug dose-response values which measure the ratio of treated to untreated cells after treatment with a specific drug concentration. In this application, a set of drug data obtained from the NCI60 human tumour cell line database Shoemaker (2006) is used to predict the drug response by considering gene expression, protein and microRNA abundance. As per the contemplated scope, the UNOMT application we focus on in the study is conducted on single-drug response prediction using NCI60 and gCSI datasets. We used 1006 drugs from NCI60 database for this evaluation and gCSI for the cross-validation. The original application runs sequentially, and our contribution is providing a parallelized runtime for data engineering and running the deep learning workload alongside it.

The drug response model contains a dense input layer of shape 1537 to get the concatenated results of the gene network and the drug network response along with the concentration value. Within the drug response regression network, there is another residual block being used repeatedly. This layer is called the drug response block module, which contains two dense layers followed by a dropout layer and a ReLU activation layer. **Figure 6** depicts the response block module.

Residual blocks are stacked, and a set of dense layers are as well. Finally, the regression layer contains a single output dense layer. The number of response blocks can be customized

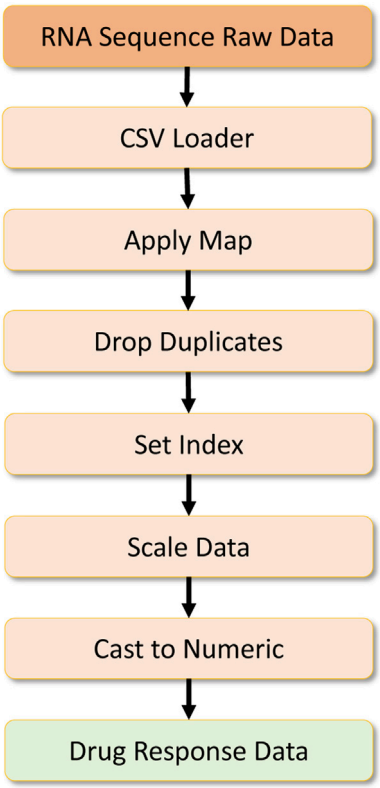

FIGURE 10 | RNA sequence data processing.

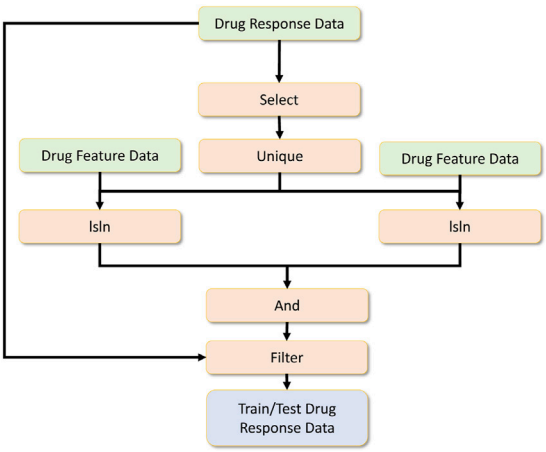

FIGURE 11 | Drug response overall data processing.

dynamically and the number of dense layers that follow it. All these parameters can be provided as a hyper-parameter in the application configuration file. **Figure 7** shows the drug response regression network.

This network is trained in a distributed data-parallel model since it contains a large dataset and a complex network compared to the other examples. The corresponding data engineering component is also distributed data-parallel, which is discussed in detail in **Section 4.3**.

4.3 Data Engineering Component

UNOMT application uses 2.5 million samples of cancer data across six research centres. This model analyses the study bias across these samples to design a unified drug response model. Before building this model, the application consists of a data engineering workload written in Pandas. The application consists of a few data engineering operators: concat (inner-join), to_csv, rename, read_csv, astype, set_index, map, isnull, drop, filter, add_prefix, reset_index, drop_duplicates, not_null, isin and dropna.

The existing data engineering workload is written in Pandas and does not run in parallel. We re-engineered this application to a parallel data engineering workload. We designed a seamless integration between data analysis and data engineering workload consuming state-of-the-art high-performance computing resources. We also integrated a Modin-based implementation to showcase the performance comparison with our implementation. The data engineering workload is executed in CPU-based distributed memory, and the data analytical workload can be either run in CPU or GPU. We use Pytorch for data analytics workload and extend it to PyTorch distributed data-parallel training. Our objective is to integrate an HPC-based full stack of data analytics-aware data engineering for scalability. PyCylon only supports this feature at the moment. Also, we stress the importance of designing a BSP-based model for deep learning workloads associated with data engineering components for better performance and scalability in HPC hardware.

The data analytics component requires a set of features to be engineered from the raw data. Here, three primary datasets are necessary to create the complete dataset for the drug response model. **Figure 8** refers to the primary dataset, which contains the drug response. The raw dataset possesses additional features, so the data is loaded in the initial stage, and a column filtering operation selects extract the expected features. Then a map operation is performed to preprocess a drug ID column to remove symbols from the columns and create a consistent drug ID. Once the data are cleaned, they are scaled with the Scikit-learn preprocessing library for scaling numerical values. After this, the data are fully converted into a numeric type to provide numeric tensors for the deep learning workload. In the parallel mode, we partition this dataset with the set parallelism, upon which it is passed to the corresponding operators.

To formulate the global dataset, we require two other datasets which act as metadata to filter and process the primary drug response dataset. The first is the drug feature raw dataset, which contains drug features required to be located in the drug response data. Two sub-datasets contribute to formulating the drug feature dataset. We merge them by performing an inner join on the dataset based on the index formed on the drug IDs. After that, we cast the data into numeric types and output them as a numeric array which is later converted to a numeric tensor for deep learning. This data processing workflow is shown in **Figure 9**.

The other dataset required is the RNA sequence dataset containing information about RNA sequences. Here the

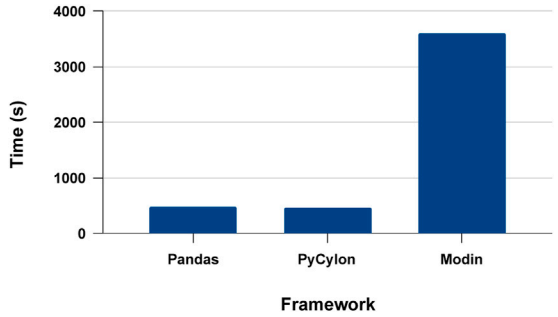

FIGURE 12 | Sequential data engineering.

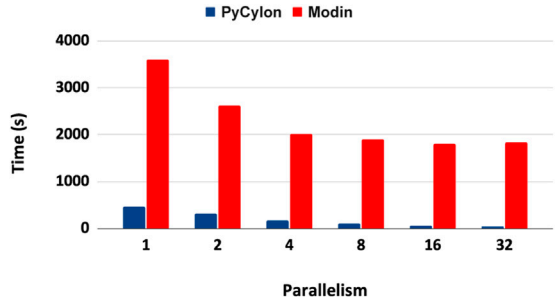

FIGURE 13 | Multi-core data parallel data engineering performance.

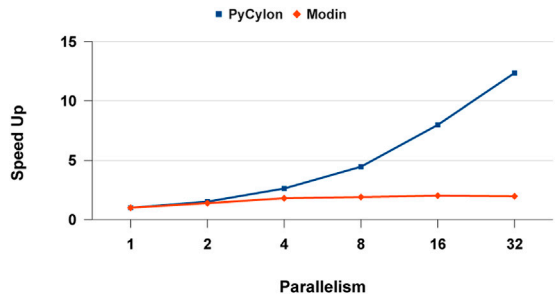

FIGURE 14 | Multi-core data parallel data engineering speed-up.

dataset is first processed to remove specific symbols by a map operation, and then duplicate records are dropped by a drop duplicate operator. Then an index is set for this dataset, and later on, scaling is done on the numeric data using the Scikit-Learn preprocessing library. Finally, the data is cast to a numeric type, and preprocessed RNA-sequence data are formulated as a Numpy array, which is later converted into a numeric tensor for the deep learning workload. This data processing pipeline can be found in **Figure 10**.

Once the drug response data, drug feature data and RNA-sequence data are preprocessed, the final data for the drug response model is engineered as shown in **Figure 11**. The processed drug response data are further feature-selected, and a unique operation is applied. Then the RNA sequence data is filtered by checking whether specific drug-related RNA sequences are present. The same is done for the drug feature dataset. These two operations are done by the isin operator. Afterwards, the common drug set is selected by performing an and operation, and later these common drug-related drug response data filters are used to get the final drug response data.

Among the operators applied, since we partitioned the data, each data engineering operator can work independently in a pleasingly parallel manner. But we can rely on the distributed unique operator to ensure no duplicate records are used for deep learning across all processes. Note that the data engineering component of this application is feature engineering metadata, and we use them to filter an extensive dataset converted to formulate the expected input for the drug response model.

5 PERFORMANCE EVALUATION

The original UNOMT application was a single-threaded application implemented on Pandas for data engineering and PyTorch for deep learning. Our first goal was to implement the sequential version of the application and improve the sequential performance. After the first stage, we conducted distributed experiments to see how we could scale our workload on CPUs for data engineering. We also extended the deep learning component of this application by integrating with PyTorch distributed execution framework on both CPUs and GPUs using MPI and NCCL, respectively. Our goal was to seamlessly incorporate a deep learning-aware data engineering workload using a single Python data engineering and deep learning script with a single runtime in this benchmark. Also, note that we used the drug response network-related more extensive data distribution for the application benchmark. At the same time, the smaller networks require a much shorter execution time than this larger model.

5.1 Setup

For the experiments, we had two sets of clusters for CPUs and GPUs. Victor cluster of Future Systems was used with six nodes and 16 processes per each on the maximum parallelism for CPUs. This cluster contains Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10 GHz machine per node. GPU experiments had Tesla K80s with 8 GPU devices on Google Cloud Platform. For single-node single-process executions, we used the same Victor nodes. Pandas, PyCylon (single-core) and Modin (single-core) were deployed for the sequential performance comparisons. Finally, for the distributed performance comparisons, we used PyCylon and Modin on single node multi-core scaling. We selected Modin instead of Dask because it is closer to the data engineering stack proposed by PyCylon due to eager execution and the ability to convert an existing Pandas data engineering workload in a straightforward manner.

5.2 Sequential Execution Performance

We first conducted experiments to evaluate the proposed systems' single process execution, PyCylon, Modin, and Pandas. Modin

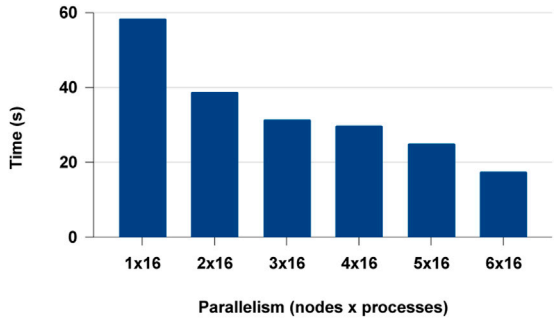

FIGURE 15 | PyCylon multi-node multi-core distributed data parallel data engineering.

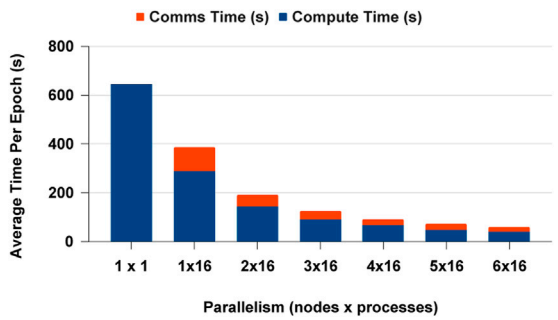

FIGURE 16 | Distributed data parallel deep learning on CPU.

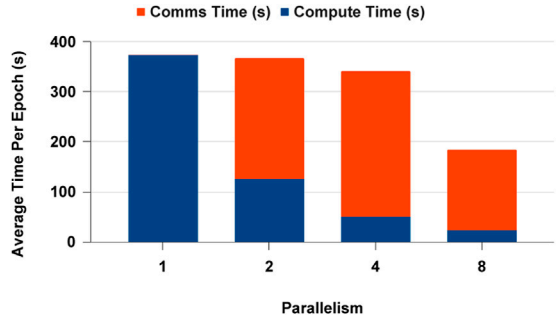

FIGURE 17 | Distributed data parallel deep learning on GPU

provides the ability to convert a Pandas data engineering workload utilizing a single line of code. In contrast, PyCylon offers a dynamic API allowing the user to dynamically decide the nature of sequential and parallel operators. We evaluated the data engineering performance for the drug response data preprocessing workload used for the drug response regression network. **Figure 12** has the single-core performance for the aforementioned data engineering workload. We observe that the sequential performance of PyCylon and Pandas are very similar, while Modin is much slower. These measurements include data loading efficiency plus overall operator performance improvements. But in a general way, Pandas and PyCylon have almost similar performance in most operators except for data loading, duplicate handling, null handling and search operations involved in this application. Note that both PyCylon and Modin are evolving data engineering frameworks to support data engineering on tabular data.

This section of the application entails data normalization, parameterized data partition, and statistical data processing with third-party Python libraries like Scikit-Learn. These libraries integrate with Pandas dataframe seamlessly. Since PyCylon employs zero-copy conversion to and from Pandas, such third-party libraries can be easily integrated without a performance penalty. But for Modin, it cannot go back-and-forth between the Pandas data structure. This caused some of these operations to be relatively slower for Modin, compared to Pandas and PyCylon. This shows that we have to go beyond the dataframe construct and integrate with third-party libraries in implementing real-world applications. And to integrate with such libraries, data engineering frameworks must be very well designed with widely used data structures used by data scientists.

5.3 Distributed Execution Performance

UNOMT computation can be distributed in a data-parallel setting. For the distributed implementation, the sequential scripts were ported to distributed *HPTMT* (PyCylon) operators. We initially evaluated the performance for a single-node multi-core setup. **Figure 13** shows the results for that application. These results show that the PyCylon is scaling well compared to Modin in the distributed data-data parallel setting.

Figure 14 depicts the relative speed-up for each framework. We observe that PyCylon has a much relative speed-up compared to Modin. We observed the similar scaling results when we comparing the performance for distributed-join operation. A reasonable conclusion drawn from these results is that data engineering applications could greatly benefit from employing *HPTMT* architecture.

We extended the distributed experiments further for multi-node multi-core. We observed that Modin failed to scale beyond a single node and failed in the cluster set-up. This could be a lack of documentation or an issue with the distributed framework Modin uses. Modin doesn't contain its own distributed runtime but relies on Ray or Dask. But with PyCylon, conveniently gets distributed in multi-node BSP (ex: MPI) execution environment. The distributed data engineering performance for PyCylon is shown in **Figure 15**.

5.4 Deep Learning Execution

The deep learning experiments also extend data engineering runs on CPUs but both CPU and GPUs. As indicated previously, deep learning computation also built on distributed data-parallel (DDP) setting. For these experiments, we used the same number of processes for both data engineering and deep learning. But PyCylon can be further improved to run in many-to-many process mapping for more complex data-parallel executions.

We selected PyTorch distributed communication framework with MPI for CPUs and NCCL for GPUs for the data analytics scaling experiments. The single process experiment results are the

similar for PyCylon and Pandas, and both have the same PyTorch codebase. Furthermore, all the data were in memory before the deep learning workload, so there was no overhead in loading data to create mini-batches. The CPU-based DDP experiments scaled well across multi-nodes, but we observed a slight memory overhead, causing the application to scale below the ideal point. We completed more experiments to evaluate an overhead from the data engineering framework, but we observed no significant overheads hindering the scalability on CPUs. **Figure 16** highlights the average computation and communication time spent per epoch as we add more resources to the setup. We used a locally built PyTorch binary with MPI execution. One significant factor is that PyTorch becomes an ideal distributed computation deep learning framework for PyCylon since PyCylon also supports an MPI backend for distributed computation.

The GPU-based DDP experiments were handled with a single-node multi-GPU experiment setting to see how the data analytics workload could be scaled on the NCCL execution framework with PyTorch. **Figure 17** displays the results for single GPU and multi-GPU experiments. We observed that the execution time was dominated by the communication time. With the increase of parallelism, the number of communications across devices increases, but the number of batches that has to be sent across devices decreases. This gives an advantage in scaling. When we consider the computation time, we saw that scaling happens closer to the ideal scaling point in all parallel settings. In addition, the computation is much faster in Parallelism 2 than in Parallelism 1, where the memory overhead is 50% less than the sequential execution. When considering CPU vs GPU performance for the deep learning workload, the speed-up from GPUs is 2x compared to CPUs in this network.

6 RELATED WORK

There are many efforts to build efficient distributed operators for data science and data engineering. Frameworks like Apache Spark (Zaharia et al., 2016), Apache Flink (Carbone et al., 2015) and Map Reduce (Dean and Ghemawat, 2008) are legacy systems created for data engineering. And many programming models have been developed on top of these big data systems to facilitate data analysis (Belcastro et al., 2019). Later on, these systems adopted the data analytics domain under their umbrella of big data problems. But with the emerging requirement for high-performance computing for data science and data engineering, the existing parallel operators in these frameworks don't provide adequate performance or flexibility (Elshawi et al., 2018). Frameworks like Pandas McKinney (2011) gained more popularity in the data science community because of their usability. Pandas only provide serial execution, and Dask (Rocklin, 2015) uses it internally (parallel Pandas) to provide parallel operators. Also, it was re-engineered as Modin (Petersohn et al., 2020) to run the dataframe operators in parallel. But these efforts are mainly focused on a driver-based asynchronous execution model, a well-known bottleneck for distributed applications.

The majority of the data analytics workloads tend to use data-parallel execution or bulk synchronous parallel (loosely synchronous) mode. This idea originated in 1987 from Fox, G.C. in the article "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems" Fox (1989). Later, a similar idea was published in an article by Valiant, L Valiant (1990) in 1990 which introduced the term "*Bulk Synchronous Parallel*". Frameworks like PyTorch (Paszke et al., 2019) adopted this HPC philosophy, and distributed runtimes like Horovod (Sergeev and Del Balso, 2018) generalized this practice for most of the existing deep learning frameworks. They were adopting this philosophy along the same time HPC-driven big data systems like Twister2 (Fox, 2017; Abeykoon et al., 2019; Wickramasinghe et al., 2019) were created to bridge the gap between data engineering and deep learning. But with the language boundaries of Java (Ekanayake et al., 2016) and usability with native-C++ based Python implementations were favoured over JVM-based systems. PyCylon (Abeykoon et al., 2020) dataframes for distributed CPU computation and Cudf (Hernández et al., 2020) dataframes for distributed GPU computation were designed. The seamless integration of data engineering and deep learning was a possibility with such frameworks and nowadays are being widely used in the data science and data engineering sphere to do rapid prototyping and design production-friendly applications.

7 LIMITATIONS AND FUTURE WORK

As showcased in the **Section 5**, *HPTMT* Model scales well in a distributed environment using BSP execution. This requires dedicated resource allocation. Thus it does not support dynamic auto-scaling, which may be an important aspect in a multi-tenant cloud environment. In most of the client-server (fully asynchronous) frameworks such as Dask, Spark, etc. provide the ability to allocate new resources without interrupting current job. Even with a process-memory snapshot mechanism, the system comes to a pause and will be restarted with the new processes. Furthermore, fault-tolerance is another useful aspect in a cloud setup. Even though the cloud hardware are becoming increasingly cheaper, more reliable, and widely available, we believe that dynamic scaling and fault tolerance would be important, and those would be incorporated with *HPTMT* and Cylon in the future.

Our next focus is to provide an enhanced set of collective communication operators on the tabular level to data science/engineering application developers. The main future objective is to provide a set of advanced APIs for the data science application developers to design complex data engineering applications with a high performance toolkit in hand with much better usability.

Furthermore, we believe that both BSP and fully asynchronous execution models are important for complex data engineering pipelines. We are currently integrating an asynchronous execution model into *HPTMT* and Cylon, using workflow management concepts. This would enable creating individual data engineering workflows that runs on BSP, while each of these workflows be scheduled asynchronously. We believe such a

system would optimize resource allocation without hindering the overall performance.

8 CONCLUSION

In this paper, we proposed the *HPTMT* architecture that defines an operator and execution model for scaling data-intensive applications. We showcased the applicability of this architecture in an en end-to-end application using Cylon framework, where data engineering and deep learning operators working together in a single distributed program. We believe that it is important to formulate and clearly define the core concepts used in developing Cylon, which could help in building highly scalable big-data applications in the future. *HPTMT* multi-process experiments' results show how well the proposed system architecture can scale compared to the existing systems with the non-synchronous mode of computation. Also, the parallel performance gain ratio is 6:1 in favor of the proposed system. This highlighted the importance of *HPTMT* based distributed and local operators on a different data structure that can work together in a single program. Further, the *HPTMT* style operators are more efficient in executing at scale, due to their loosely synchronous nature and low scheduling/coordination overhead. With the future work proposed for the architecture, we believe that we can elevate Cylon to be a truly high-performance data engineering framework built for the future.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/cylondata/cylon, https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/UnoMT.

AUTHOR CONTRIBUTIONS

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

FUNDING

This work is partially supported by the National Science Foundation (NSF) through awards CIF21 DIBBS 1443054, SciDatBench 2038007, CINES 1835598 and Global Pervasive Computational Epidemiology 1918626.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fdata.2021.756041/full#supplementary-material

REFERENCES

Abeykoon, V., Kamburugamuve, S., Govindrarajan, K., Wickramasinghe, P., Widanage, C., Perera, N., et al. (2019). "Streaming Machine Learning Algorithms with Big Data Systems," in 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA (IEEE), 5661–5666. doi:10.1109/bigdata47090.2019.9006337

Abeykoon, V., Perera, N., Widanage, C., Kamburugamuve, S., Kanewala, T. A., Maithree, H., et al. (2020). "Data Engineering for Hpc with python," in 2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC), Atlanta, GA, USA (IEEE), 13–21. doi:10.1109/pyhpc51966.2020.00007

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., et al. (2005). The Fortress Language Specification. *Sun Microsystems* 139, 116.

Apache Arrow (2021). Apache Software Foundation (Accessed 2021/Aug). Apache arrow. Available at: https://arrow.apache.org/(Accessed Aug 08, 2021).

Apache Parquet (2021). Apache Software Foundation (Accessed 2021/Aug). Apache parquet project. Available at: https://parquet.apache.org/(Accessed Aug 08, 2021).

Argo Home Page (2021). Available at: https://argoproj.github.io/argo-workflows/ (Accessed Aug 08, 2021).

Babuji, Y. N., Chard, K., Foster, I. T., Katz, D. S., Wilde, M., Woodard, A., et al. (2019). "Scalable Parallel Programming in Python with Parsl," in Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (PEARC '19), Chicago, IL, USA, (New York, NY, USA: Association for Computing Machinery), 1–8. doi:10.1145/3332186.3332231

Belcastro, L., Marozzo, F., and Talia, D. (2019). Programming Models and Systems for Big Data Analysis. *Int. J. Parallel, Emergent Distributed Syst.* 34, 632–652. doi:10.1080/17445760.2017.1422501

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and Kubernetes. *Queue* 14, 70. doi:10.1145/2898442.2898444

CarboneEwen, S., Haridi, S., Katsifodimos, A., Markl, V., and Tzoumas, K. (2015). "Apache Flink: Stream and Batch Processing in a Single Engine Paris," in Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society Special Issue on Next-Generation Stream Processing, December 2015 38 (4). Available at: http://sites.computer.org/debull/A15dec/p28.pdf.

Carpenter, B., Zhang, G., Fox, G., Li, X., and Wen, Y. (1998). Hpjava: Data Parallel Extensions to Java. *Concurrency: Pract. Exper.* 10, 873–877. doi:10.1002/(sici)1096-9128(199809/11)10:11/13<873:aid-cpe402>3.0.co;2-q

Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 291–312. doi:10.1177/1094342007078442

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., et al. (2005). X10: an Object-Oriented Approach to Non-uniform Cluster Computing. *Acm Sigplan Notices* 40, 519–538. doi:10.1145/1103845.1094852

Dean, J., and Ghemawat, S. (2008). MapReduce. *Commun. ACM* 51, 107–113. doi:10.1145/1327452.1327492

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., et al. (2015). Pegasus, a Workflow Management System for Science Automation. *Future Generation Comput. Syst.* 46, 17–35. doi:10.1016/j.future.2014.10.008

Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., et al. (2003). *Sourcebook of Parallel Computing, 3003*. San Francisco, CA: Morgan Kaufmann Publishers.

Ekanayake, S., Kamburugamuve, S., Wickramasinghe, P., and Fox, G. C. (2016). "Java Thread and Process Performance for Parallel Machine Learning on Multicore Hpc Clusters," in 2016 IEEE international conference on big data (Big Data), Washington, DC, USA (IEEE), 347–354. doi:10.1109/bigdata.2016.7840622

Elshawi, R., Sakr, S., Talia, D., and Trunfio, P. (2018). Big Data Systems Meet Machine Learning Challenges: Towards Big Data Science as a Service. *Big Data Res.* 14, 1–11. doi:10.1016/j.bdr.2018.04.004

Fox, G. (2017). "Components and Rationale of a Big Data Toolkit Spanning Hpc, Grid, Edge and Cloud Computing," in Proceedings of the10th International Conference on Utility and Cloud Computing, Austin, TX, USA (New York, NY, USA: ACM), 1, 2017. UCC '17. doi:10.1145/3147213.3155012

Fox, G. C. (1989). "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems," in Proceedings of the third conference on Hypercube concurrent computers and applications-Volume 2, Pasadena, CA, USA, 897–955. doi:10.1145/63047.63048

Fox, G. C., Williams, R. D., and Messina, G. C. (1994). *Parallel Computing Works!*. San Francisco, CA: Morgan Kaufmann Publishers.

Hernández, B., Somnath, S., Yin, J., Lu, H., Eaton, J., Entschev, P., et al. (2020). "Performance Evaluation of python Based Data Analytics Frameworks in summit: Early Experiences," in Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI, Oak Ridge, TN, SMC 2020. Communications in Computer and Information Science. Editors J. Nichols, B. Verastegui, A. Maccabe, O. Hernandez, S. Parete-Koon, and T. Ahearn (Cham: Springer) Vol. 1315. doi:10.1007/978-3-030-63393-6_24

Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E. N., O'Malley, O., et al. (2014). "Major Technical Advancements in Apache Hive," in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, Snowbird, UT, USA, 1235–1246. doi:10.1145/2588555.2595630

Imam, S., and Sarkar, V. (2014). "Habanero-java Library: a Java 8 Framework for Multicore Programming," in Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools, Cracow, Poland, 75–86. doi:10.1145/2647508.2647514

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), Chicago, IL, USA. doi:10.1109/cloud53861.2021.00036

Kubeflow (2021). Kubeflow home page. Available at: https://www.kubeflow.org/.

McKinney, W. (2011). "Pandas: A Foundational python Library for Data Analysis and Statistics," in Workshop collocated with the 24rd International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), Seattle, WA, USA, November 18, 2011 14, 2011. Available at: https://www.dlr.de/sc/en/Portaldata/15/Resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf (Accessed Dec 23, 2021).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). "Pytorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. Editors H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Vancouver, Canada), 8026–8037.

Petersohn, D., Macke, S., Xin, D., Ma, W., Lee, D., Mo, X., et al. (2020). Towards Scalable Dataframe Systems. arXiv preprint arXiv:2001.00888.

Rocklin, M. (2015). "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling," in Proceedings of the 14th python in science conference (Citeseer), Austin, TX, USA, 130, 136. doi:10.25080/majora-7b98e3ed-013

Sergeev, A., and Del Balso, M. (2018). Horovod: Fast and Easy Distributed Deep Learning in Tensorflow. arXiv preprint arXiv:1802.05799.

Shoemaker, R. H. (2006). The Nci60 Human Tumour Cell Line Anticancer Drug Screen. *Nat. Rev. Cancer* 6, 813–823. doi:10.1038/nrc1951

Valiant, L. G. (1990). A Bridging Model for Parallel Computation. *Commun. ACM* 33, 103–111. doi:10.1145/79173.79181

Wickramasinghe, P., Kamburugamuve, S., Govindarajan, K., Abeykoon, V., Widanage, C., Perera, N., et al. (2019). "Twister2: Tset High-Performance Iterative Dataflow," in 2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS), Shenzhen, China (IEEE), 55–60. doi:10.1109/hpbdis.2019.8735495

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, China (IEEE), 122–132. doi:10.1109/smds49396.2020.00022

Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S., and Foster, I. (2011). Swift: A Language for Distributed Parallel Scripting. *Parallel Comput.* 37, 633–652. doi:10.1016/j.parco.2011.05.005

Wozniak, J. M., Yoo, H., Mohd-Yusof, J., Nicolae, B., Collier, N., Ozik, J., et al. (2020). "High-bypass Learning: Automated Detection of Tumor Cells that Significantly Impact Drug Response," in 2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S), Virtual location, 1–10. doi:10.1109/MLHPCAI4S51975.2020.00012

Xia, F., Allen, J., Balaprakash, P., Brettin, T., Garcia-Cardona, C., Clyde, A., et al. (2021). A Cross-Study Analysis of Drug Response Prediction in Cancer Cell Lines. arXiv preprint arXiv:2104.08961. doi:10.1093/bib/bbab356

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). "Spark: Cluster Computing with Working Sets," in Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (Berkeley, CA, USA: USENIX Association), 10, 2010. HotCloud'10.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., et al. (2016). Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 56–65. doi:10.1145/2934664

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.