HPTMT: Operator-Based Architecture for Scalable High-Performance Data-Intensive Frameworks

Supun Kamburugamuve[†], Chathura Widanage[†], Niranda Perera[*], Vibhatha Abeykoon[‡], Ahmet Uyar[†],
Thejaka Amila Kanewala[‡], Gregor von Laszewski [†] and Geoffrey Fox[§†]
[*]Luddy School of Informatics, Computing and Engineering, Bloomington, IN 47408, USA
dnperera@iu.edu
[†]Digital Science Center, Bloomington, IN 47408, USA
{skamburu, cdwidana, auyar}@iu.edu, laszewski@gmail.com
[‡]Indiana University Alumni, IN 47408, USA
{vibhatha,thejaka.amila}@gmail.com
[§]From August 2021, Biocomplexity Institute & Initiative and Computer Science Dept., University of Virginia
gcfexchange@gmail.com

***Abstract*—Data-intensive applications impact many domains, and their steadily increasing size and complexity demands high-performance, highly usable environments. We integrate a set of ideas developed in various data science and data engineering frameworks. They employ a set of operators on specific data abstractions that include vectors, matrices, tensors, graphs, and tables. Our key concepts are inspired from systems like MPI, HPF (High-Performance Fortran), NumPy, Pandas, Spark, Modin, PyTorch, TensorFlow, RAPIDS(NVIDIA), and OneAPI (Intel). Further, it is crucial to support different languages in everyday use in the Big Data arena, including Python, R, C++, and Java. We note the importance of Apache Arrow and Parquet for enabling language agnostic high performance and interoperability. In this paper, we propose *High-Performance Tensors, Matrices and Tables* (*HPTMT*), an operator-based architecture for data-intensive applications, and identify the fundamental principles needed for performance and usability success. We illustrate these principles by a discussion of examples using our software environments, Cylon and Twister2 that embody *HPTMT*.**

***Index Terms*—Data intensive applications, Operators, Vectors, Matrices, Tensors, Graphs, Tables, DataFrames, HPC**

I. Introduction

Data-intensive applications have evolved rapidly over the last two decades, and are now being widely used in industry and scientific research. Large-scale data-intensive applications became mainstream with the rise of the *map-reduce programming paradigm*. However, the data engineering community has come a long way in integrating the idea of *map-reduce*. There is a broader understanding on the data engineering application classes, and specialized frameworks that serve them. Modern systems can crunch data and learn from them using sophisticated algorithms that even make use of custom hardware solutions. We have seen different programming models and APIs being developed to make it easier to program data-intensive applications.

Due to the diverse nature of data-intensive applications, it is hard for one framework to support all classes of problems efficiently. For example, we may need to load data, curate them, utilize machine learning algorithms, conduct post-processing, and perform visualizations, all as parts of single application pipeline. Such an integration is done either as a custom-developed single program or by developing the pieces separately and combining them into a data-intensive application workflow. The complexities are vast, and having interoperable systems enhances usability significantly.

The various application classes require tailored abstract concepts. Vectors, matrices, tables, graphs, and tensors are widely used examples in data-intensive computations. For applications to benefit from these abstractions efficiently, operations around them must be implemented to provide solutions for general-purpose and problem-specific domains. Among these operations, matrix multiplication, vector addition, and table joins are some standard operations. We can represent these abstract objects in the main memory via data structures; vectors, matrices, and tensors are shown as arrays, graphs are represented either using matrices or as edge lists, and tables are configured as a set of columns or rows. One example is Apache Spark [1], which has a table abstraction originating from relational algebra. Deep learning systems such as PyTorch [2] are based on tensor abstractions originating from linear algebra.

When developing applications with modern frameworks, we often resort to data abstractions and their operators that may not fit the original problem because required data abstractions are not supported. This is due to lack of cohesiveness amongst systems and in-turn leaves a lot of performance on the table, even though they are capable of doing a top-notch job at their intended purpose. Ideally we would like different data abstractions and operators to work together to solve problems. In order to address this issue, we will analyze the fundamental designs of these systems while focusing on the elementary operators they provide and how they can work hand-in-hand.

We introduce the *HPTMT*(High Performance Tensors, Matrices and Tables) architecture in this paper, which defines an operator-centric interoperable design for data-intensive applications. The authors have developed two frameworks called

Cylon [3] and Twister2 [4] aimed at developing data-intensive applications. We will take these as examples to showcase the importance of designing operators and how the various systems implementing them can work together according to *HPTMT* architecture.

The rest of the paper is organized as follows. Section II gives a high-level introduction and motivation for *HPTMT*. Section III describes arrays and distributed operators around them, while Section IV focuses on tables. The next two sections, V and VI, talk about programming models and execution models around these data structures. Section VII presents the operator-based architecture, and Section VIII discusses how this architecture is realized in Twister2 and Cylon.

II. HPTMT Architecture

One of the most successful approaches to parallel computing is based on the use of runtime libraries of well-implemented parallel operations. This was for example a key part of High-Performance Fortran HPF [5] and related parallel environments (HPJava [6], HPC++ [7], Chapel [8], Fortress [9], X10 [10], Habanero-Java [11]). Such systems had limited success; maybe because the HPC community did not define sufficient operators to cover the sophisticated computational science simulations largely targeted by those languages with typically sparse or dense matrix operators. However data-intensive applications have used similar ideas with striking success.

Perhaps, the most dramatic event was the introduction of MapReduce [12] some 15 years ago, and its implementation in Hadoop which enabled parallel databases as in Apache Hive. MapReduce adds Group-By and key-value pairs to the Reduce operation common in the simulation applications of the previous HPF family. The powerful yet simple MapReduce operation was expanded in Big Data systems especially through the operators of Databases (union, join, etc.), Pandas (we identify 244 dataframe operators out of 4782 Pandas methods), and Spark, Flink [13], Twister2 (70). Deep Learning environments such as PyTorch, TensorFlow [14] (with Keras [15]) added further (over 700) operators to build deep learning components and execution.

The powerful array libraries of Numpy [16] are built on a large (at least 1085) set of array operations used in the original scientific simulation applications. Oversimplifying HPF as built around matrix or array operators, we suggest today that the natural approach is HPTMT or High-Performance Tables, Matrices, and Tensors. Operator based methods are not just used to support parallelism but have several other useful 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.

HPTMT is supported by many libraries including those like ScaLAPACK [17] (320 functions with a factor 4 more counting different precisions) and its follow-ons such as Intel's MKL (oneAPI and Data Parallel C++) [18] originally motivated by HPF simulation goals but equally important for Big Data. The NVIDIA RAPIDS [19] project is building a GPU library covering much of *HPTMT* requirements as is our Cylon project for CPUs. Modin [20] is using Dask [21] and Ray [22] for parallel Pandas operators.

Recently Apache Arrow [23] and Parquet [24] have been developed providing important tools supporting *HPTMT*. 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 large amount of data so that each processor has a big-enough workset to processes efficiently. It is a well-established principle that the problem needs to be large enough for the success of parallel computing [25], which the latest Big Data trends also follow. Note that in scientific computing, the most effective parallel algorithms use block (i.e. row and column) decompositions to minimize communication/compute ratios. Such block decompositions can be used in Big Data [26] (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 that one can 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 (Parsl [27], Swift [28], Pegasus [29], Argo [30], Kubeflow [31], Kubernetes [32]) or dataflow (Spark, Flink, Twister2) preserving the parallelism of HPTMT.

III. Arrays

Arrays are a fundamental data structure of scientific computing, machine learning, and deep learning applications because they represent vectors, matrices, and tensors. An array consists of elements from the same data type, and an index can address each value. An array is stored in contiguous memory of a computer. The size of an element and the index can efficiently compute the memory location of an array element. A single value array of a type is equivalent to a scalar of that type. As such, arrays can represent all primitive types. Variable width data types such as Strings would require a composite arrays that represent data and offsets/ strides.

TABLE I
ARRAY-BASED DISTRIBUTED OPERATIONS AS SPECIFIED BY MPI

Operation	Description
Broadcast	Broadcast an array from one process to many other processes.
Gather/AllGather	Collects arrays from different processes and creates a larger array in a single process or many processes.
Scatter/AllToAll	Redistributes the parts of an array to different processes.
Reduce/AllReduce	Element-wise reduction of arrays. Popular operations include SUM, MIN, MAX, PROD.

A. Vectors and Matrices

We can represent a vector directly using an array. A matrix is a 2-dimensional grid, and each value can be addressed using a row index and a column index. We need to use 2-dimensional arrays or 1-dimensional arrays to represent a matrix. We can store a matrix in row-major format or column-major format. In row-major format, the values of a row are in the contiguous memory of the array. In contrast, the column values are found in the array's contiguous memory with the column-major format.. These formats are designed to match the access patterns of matrices when doing calculations.

1) Sparse Matrices: Sparse matrices are defined as matrices where the majority of elements are zero. To store these as regular dense matrices wastes both memory and CPU cycles during computations. As an alternative, there are efficient layouts such as Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), and Doubly Compressed Sparse Column (DCSC) for sparse matrices. All these formats use arrays to store the matrix in memory.

B. Tensor

A tensor can be interpreted of as a more generic view on a collection scalars or vectors or both to represent a mathematical model or data structure. A matrix is also a tensor by definition, and it is the most generic abstraction for mathematical computations. Similar to matrices, tensors can be stored according to the access patterns using arrays.

C. Operations

Distributed operations around arrays are defined in the MPI (Message Passing Interface) [33] standard as collective operations. The table I describes some of these operations, which are derived from common communication patterns involved when dealing with vectors and matrices in the form of arrays. They are optimized to work on thousands of computers using data transfer algorithms [34] that can minimize the latency and utilize the available network bandwidth to the fullest.

IV. TABLES

A table is an ordered arrangement of data into a 2-dimensional rectangular grid with rows and columns. A single column of a table has data of the same type, while different columns can have different data types (heterogeneous data).

A. Tables in Memory

We can store table data in main memory using a simple technique such as a list of records or compact formats that keep the values in contiguous memory. Compact memory layouts can store tables similar to row-major and column-major representations in matrices. Having a list of records can lead to inefficient use of memory and degrade the performance of applications due to cache misses, TLB (Translation Lookaside Buffer) being misused, and serialization/deserialization costs.

In a column-major representation, the table columns are stored in contiguous memory. Table representations are complex compared to matrices because they can have variable-length data. In such cases, length information needs to be stored along with the columns. Thus, sparsity can be embedded or integrated into separate arrays. Also, information about *null* values needs to be stored in a table. One benefit of a column-major representation is that the values of the column are of the same data type. In a row-major definition, the table rows can be stored in contiguous memory. This means that different types of values are stored in contiguous memory with different bit widths. So we need to keep track of lengths and NULL values.

B. Operations

We can use row-based partitioning, column-based partitioning, or a hybrid version to divide table data into multiple processes. Most of the time, data processing systems work on tables distributed with row-based partitioning. Relational algebra defines five base operations around tables described in Table II. Table III lists some commonly used auxiliary operations around tables. Popular table abstractions like Pandas [35] extend these to hundreds of operators.

Figure 1 shows two tables in two processes and a distributed union operation that removes duplicates. This operation needs to redistribute the records of the tables so that the identical records go to the same table. Such redistribution is common in table-based distributed operations and is commonly referred to as *shuffle* operation.

1) Shuffle: Figure 2 displays a shuffle operation applied to four tables in four processes. The shuffle is done on attribute *K*. Shuffle is similar to the array *AllToAll* operation, which is equivalent to every process scattering values to other processes. Shuffle is similar in that it scatters records of a table to every other process. What makes these two operations different are the data structure, its representation in memory, and how we select which values are scattered to which processes. In *AllToAll*, scatter occurs by a range of indexes. In tables, the shuffle takes place based on a set of column values.

Large-scale data operations require careful use of memory and optimizations to scale to a large number of cores [36]. Because of the nature of these data structures like tables and arrays, we can use one structure to represent another. For example, we can have a table to represent a matrix. Also, we can have a set of arrays to represent a table. Therefore, we can sometimes use a programming API (data structures and operators) developed for a specific set of problems, to solve

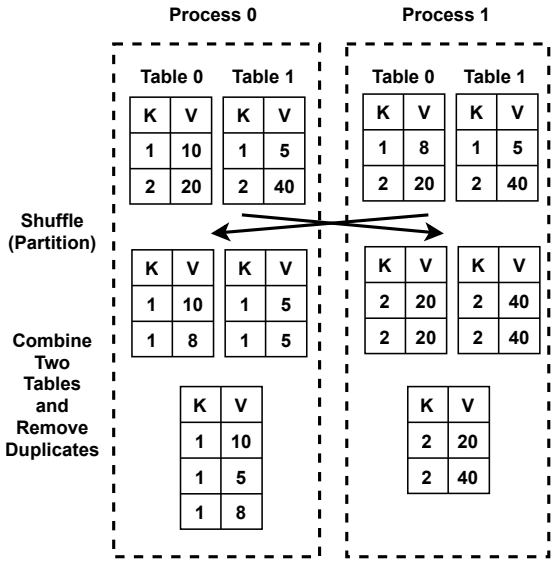

Fig. 1. Union of two tables distributed in two processes

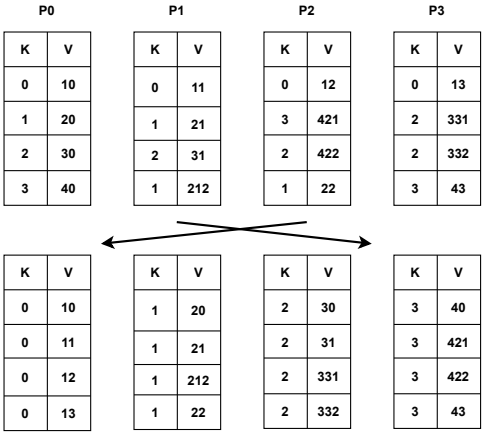

Fig. 2. Shuffle of 4 tables in 4 processes

problems in unrelated/ unintended domains. In practice, this leads to unnecessary inefficiencies in execution. For instance, say we have a table abstraction with GroupBy and aggregate operations implemented. Now we want to get AllReduce-sum semantics of a column in this table. We can do so by assigning a common key to each value in columns and doing a GroupBy on the key, followed by an aggregate operation. However, this is not an efficient method because it uses an additional column and a shuffle operation, which is more costly than the communication required for an AllReduce operation.

Arrays and tables have their own distributed operations. As seen in previous sections, the distributed operations on tables originate from relational algebra, and those on arrays derived to support linear algebra.

V. Programming Models and Operators

Data-intensive applications use both implicit and explicit parallel programming models. In an explicit model, the user is aware of the parallel nature of the program, writes the application according to a local view, and synchronizes the data distributed in multiple computers using distributed operations. MPI-based programming is the most popular explicit model. Most data-intensive applications use an implicit parallel model with a distributed data abstraction. These models have similarities to partitioned global address space models [8], [37].

TABLE II
Fundamental table operations

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 duplicates.
Cartesian Product	Applicable on two tables having similar schemas to keep only the records 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.

TABLE III
Auxiliary table operations

Operator	Description
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). Aggregations include summation and multiplication.
GroupBy	Groups the data using the given columns; GroupBy is usually followed by aggregate operations.

A. Local Data Model

In this model, the user programs a parallel process or task. Here, users only deal with the local data, and when they need to synchronize it with other processes, they invoke a distributed operator.

```
// every process loads its own data
LocalData A = readFiles()
// apply local operators
LocalData B = A.filter(Function filter)
// sort is a distributed operator, the data
// of B in the parallel processes will be sorted
LocalData C = sort(B)
// save C to disk
C.save()
```

Fig. 3. Eager execution

B. Global Data Model

In the distributed data API, the user defines an abstract object that acts as a global view for the data distributed across the cluster. This object represents a dataset such as a table or an array. The user applies operations to this global data that produces more distributed data objects. This is an implicit

parallel programming model that has been present for a while in various forms under the general umbrella of partitioned global address space (PGAS) programming model and is used by data-intensive frameworks extensively. Depending on the amount of data processed, we can have an eager model or a dataflow model using external storage for computations.

1) Eager Model: With an eager model, the operators work on in-memory data and can be executed immediately. Combining SPMD (Single Program Multiple Data)-style user code and distributed data-based API for the data structures, we can create powerful and efficient APIs for data-intensive applications. We depict the code in Figure 4. We assume 'A' is representing a partitioned table in multiple computers. Now 'B' and 'C' are also partitioned tables.

```
// load the data as partitions on
// multiple processes
DistributedData A = readFiles()
// user supplies a filter function
DistributedData B = A.filter(Function filter)
// sort is a distributed operator
// that requires network communication
DistributedData C = B.sort()
C.save()
```

Fig. 4. In-memory API

This code will run as SPMD code, but the data structures and operators can reduce the programmer's burden by providing a global data structure. The above code uses the distributed memory model with no threads.

2) DataFlow Model: Data-intensive applications constantly work with datasets that do not fit into the available random access memory of computing clusters. To work with large datasets, we need to support streaming computations that utilize external storage. The previous API with eager execution is not suitable here, and we can illustrate this with an example (see Figure 5). Here we need to save data multiple times to the disk to execute the program.

```
// A is a large dataset, so we need to
// store it in the disk
DistributedData A = readFiles()
// B is still large, so we need to store
// B in the disk
DistributedData B = A.filter(Function filter)
// to sort B, we need to use disk and
// then store C in disk
DistributedData C = B.sort()
// save C to disk
C.save()
```

Fig. 5. Eager execution

We can avoid such extensive use of external storage by executing the above program as a single graph with data streaming through it piece by piece. This is called the *dataflow model*. In this model, computations are data-driven (event-driven). The sources produce data, and this data activates the next connected computation. The middle nodes can produce more data until they reach a sink node without an output. The links represent distributed operators that carry data between nodes that can be within the same process or in different processes across machines. There are usually constraints applied to nodes that force them to be scheduled into computers in separate ways depending on the framework and the application.

The functionality of links depends on the operators and the data abstractions they represent. Each user-defined function runs on its own program scope without access to any state about other tasks. The only way to communicate between dataflow nodes is by messaging, as they can run in various places. This model is adopted by popular data processing engines such as Spark and Flink [13].

VI. Execution

Many parallel programs are instances of a single program running multiple instances working on different data, or multiple programs doing the same. This is called single program multiple data (SPMD) and multiple programs multiple data (MPMD) style parallel programs. SPMD programs are popular in batch programs where the same task instances (program) are executed in parallel processes simultaneously. For example, Hadoop map-reduce [38] programs are executed as SPMD programs where the map tasks are executed first, and then the reduce tasks are executed in parallel. MPMD programs are used in streaming applications and pipeline parallel batch programs. In an MPMD program, different tasks run on separate parallel processes. Apache Storm [39] and Flink run streaming programs in this style.

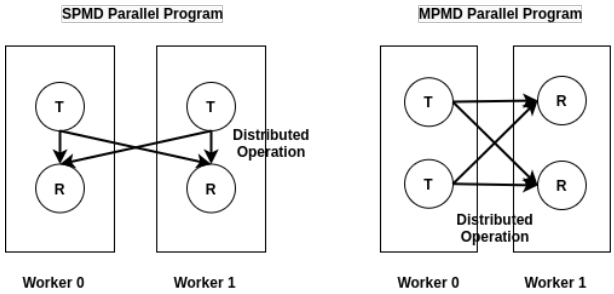

Fig. 6. SPMD and MPMD Programs

A. Memory Models

We can further divide parallel programs according to how parallel tasks share data. Here we have shared memory, distributed memory, and hybrid memory execution models. In a shared memory model, parallel instances of the program share the same memory address space. This model only allows parallel programs to run in a single computer that has many CPU cores. In a distributed memory model, every instance of the parallel program is executed on an isolated memory such as its own process. They can only access the data with processes using messaging. In hybrid memory model, some instances of parallel processes are run in the shared memory model. These groups of parallel instances need messaging to share data with other such groups.

B. Loosely Synchronous and Asynchronous Execution

We can also execute parallel applications in a loosely synchronous way or asynchronously. The loosely synchronous execution as shown in Figure 7 assumes all the tasks are executing concurrently, and the processes synchronize with each other by exchanging messages at certain points. The sections of code between communication synchronizations execute independently from other parallel processes.

In asynchronous execution, as shown in Figure 8, the tasks are decoupled in time. When a task sends a message, we can assume it is stored in a queue. Once the receiving task is ready, it picks up the message. This allows more flexible execution of the tasks independent of the programming model, but often needs a central server as a facilitator that notifies the receivers about pending messages. Also, storing a message, notifying the receiver about the message, and picking it up creates a middle step that can reduce the performance. Asynchronous execution has parallels to the preemptive scheduling we see in operating systems where it tries to simultaneously progress multiple programs to increase the responsiveness.

The asynchronous execution described here is an implementation detail because the synchronization comes from the operator. For example, in a distributed join operation, a single process requires data from all the other processes. So whether we insert messages into a queue or not, the program cannot continue until the operation is completed by receiving messages from other processes.

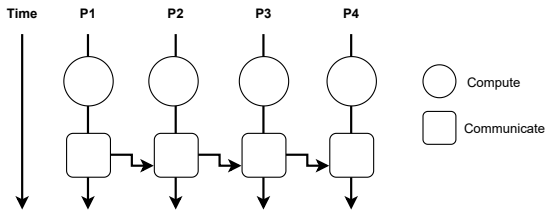

Fig. 7. Loosely Synchronous Execution

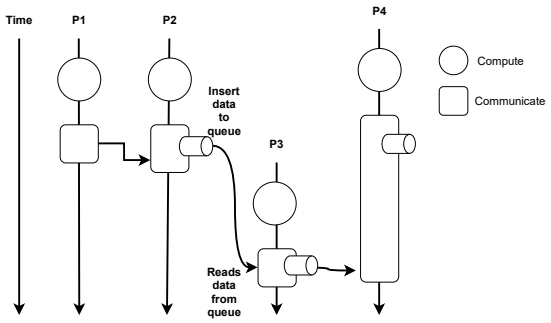

Fig. 8. Asynchronous Execution

VII. *HPTMT* Architecture Principals

We define *HPTMT* as an architecture where any combination of loosely synchronous operators built around different data abstractions working together to develop data-intensive applications. A high-level view of the architecture is shown in Figure 9, where dataflow operators and eager operators work together in a single parallel program.

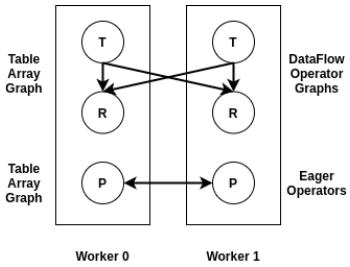

Fig. 9. Operator Architecture for Data-Intensive Applications

In general, we can categorize operators depending on whether they are designed to run on MPMD or SPMD executions. This is shown in Figure 10. Here SPMD-style programs can use dataflow-style operators for programs that demand external memory, or eager operators for applications that run in memory. MPMD-style operators tend to be dataflow operators because they are used in streaming and pipeline parallel programs.

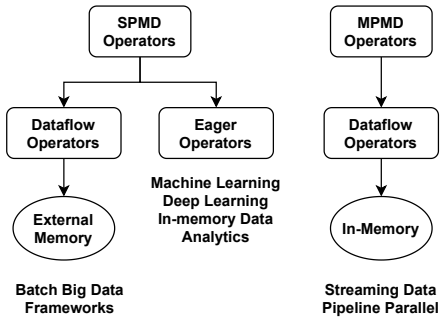

Fig. 10. Operators with SPMD and MPMD programs

A. Dataflow and Eager Operators

In general, a dataflow operator needs to take input piece-by-piece and produce output the same way. It can also take input at once and produce output individually, or take input piece-by-piece and produce output as a single object as well. This means it needs to be a non-blocking operator. Furthermore, the operators need to determine when to terminate by using a termination algorithm because inputs are not coordinated. If it is a streaming application, the operator may not terminate and continue consuming and producing data. Dataflow operators can use external storage to keep the intermediate data in order to do operations that cannot fit into the memory.

Eager operators work on in-memory data by taking input data at once and producing output all at once. This is the approach taken by operators in MPI. They can be deterministic in terms of execution because only one data input is given and one output is produced.

B. SPMD and MPMD Operators

In an SPMD-style program, the same processes participate in the operators as data producers and receivers. This can simplify the operator interfaces. For example, the collective operators in MPI standard are implemented in this fashion.

In an MPMD-style program, the operators can have data producers and receivers in different processes. This is a more general form of an operator, as it can represent SPMD-style operators as well. Twister:Net [40] is one such MPMD-style operator library. Whether it is SPMD or MPMD, we can have eager style operators or dataflow operators. Streaming systems are where we primarily see MPMD-style operators.

C. Operator Principles

Whether it is a dataflow operator or an eager operator, *HPTMT* architecture identifies several design principles for them to work together in different environments.

- Multiple data abstractions and operators - Discourage the use of data structures and operators suitable for one class of problems to be used in another class of problems. i.e. Do not use table operators for a problem that needs arrays.
- Efficient Loosely Synchronous Execution - In an asynchronous framework, operators and the scheduler are coupled. In such situations we may need to develop operators specifically targeting the framework, which is contrary to the *HPTMT* goals.
- Independence 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. We see this design in MPI-based operators where collectives are coupled to the MPI implementation's parallel process management. Frameworks such as UCX [41] are in the process of developing MPI-equivalent collective operators for arrays without process management. In other words we should be able to bootstrap operator implementation on various parallel environments.
- 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.

In some situations, we can get around certain design principles and make operators work together. For example, we can use MPI primitive based operators on different data abstractions as long as we are running within an MPI execution environment. So it partially satisfies the *HPTMT* requirements.

The general architecture of distributed operators is shown in Figure 11. At the bottom are the networking hardware and various software libraries that abstract out them. Then we have different execution units such as CPUs and GPUs. These are also abstracted by various programming APIs. On top of these we have our data structures in the memory defined according to various formats. We define the distributed and local operators on these data structures using the various execution and networking hardware.

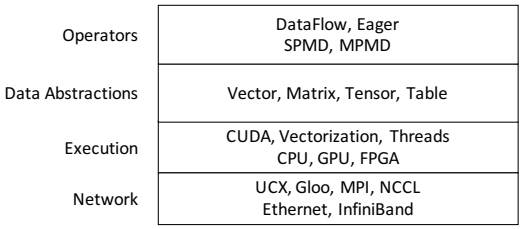

Fig. 11. Distributed operator implementation layers

D. Workflows

A workflow is a sequence of tasks connected by data dependencies. Given such a set of tasks, a workflow system orchestrates the execution of the tasks in the available resources preserving the data dependencies. Workflow systems for scientific computing applications [42] have been around for some time, with prominent scientific workflow systems such as Kepler [43] and Pegasus [29] leading the way. In addition, there are data-intensive application-specific workflow systems such as Kubeflow and Apache Airflow.

Workflow systems provide mechanisms for specific applications using domain-specific languages (DSLs) as well as using graphical user interfaces. Furthermore, we can use general-purpose programming languages to specify workflows as seen in Python-based systems such as Parsl [27] and Ray-Project [].

It is important to make the distinction between tasks of a parallel program and tasks of a workflow. A workflow task system is usually an application such as a machine learning algorithm. This algorithm may need to run on multiple computers, and it might run internally as a set of tasks. These internal tasks to the machine learning algorithm are fine-grained and usually developed using the programming methods we described earlier. It is not efficient to use workflows for finer-grained tasks because of the central coordination.

1) Remote Execution: Remote execution is a form of workflow adopted by current data systems. With this model, a data-intensive program is created at a central server and submitted to the cluster to execute. We believe having a clean separation between programs and workflow is important for the inseparability of frameworks. Mixing both can hinder development and make it difficult for programmers to think about applications.

E. Separation of Concerns

Separation of concerns is a design principle that states we should separate a program into distinct sections that address specific concerns. For example, from a parallel computing perspective, running computations on a cluster is a separate concern better addressed by a workflow engine designed precisely for that task. Developing and running a parallel application is another task that should be handled by frameworks

suited for those tasks. The remote execution methods adopted by current programs combine these two aspects into a single program.

Going by these concerns, we propose operator-based data-intensive applications orchestrated by a workflow engine as the overall architecture of data intensive applications as shown in Figure 12.

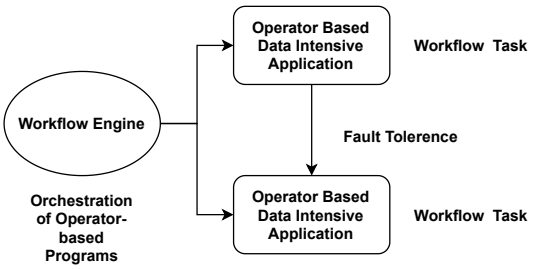

Fig. 12. Workflow for orchestrating operator-based data-intensive applications

We can combine distributed operator-based parallel programs with workflow engines to create rich data-intensive applications. Aspects such as fault handling can be moved to the workflow level to keep a balance between performance and fault tolerance overheads.

F. Fault Tolerance

Handling faults is an important aspect of large-scale data-intensive applications. Modern hardware is becoming increasingly reliable, and the chances of hardware failure during a computation are decreasing. But for applications that execute over a longer duration, handling hardware and software failure is still important. Streaming applications and long-running batch applications are some examples.

Handling faults at the operator level is inefficient as it adds additional synchronization to communication steps. Because of this, we can always handle the faults outside of the operator code. If an operator fails, we can go back to the previous state before starting the operator. But operators need to detect failures and notify the applications to handle them gracefully.

VIII. Frameworks

Let us look at Twister2 and Cylon, which are frameworks with DataFlow and eager operator APIs for data-intensive applications. We take these as examples for the proposed *HPTMT* architecture in combination with array based operators of MPI.

A. Twister2

Twister2 [4] is a data analytics framework for both batch and stream processing. The goal of Twister2 is to provide users with performance comparable to HPC systems while exposing a user-friendly dataflow abstraction for application development. TSet [44] is the distributed data abstraction of Twister2. Twister2 provides a table abstraction and array abstraction with DataFlow operators. This allows Twister2 to compute using external memory efficiently for problems that do not fit into the memory. Twister2 operators are implemented as MPMD and can support both streaming and batch applications.

Local and distributed operations can transform or combine TSets to produce new TSets with different schemas. The simplest model of TSet is modeling a distributed primitive array (series). At the same time, this can be extended to represent a table by making every element of an array a composite object, as shown in Figure 13. Although the worker level parallelism of Twister2 is set to four at line 5, TSet level parallelism can even be a different value since Twister2 internally models the dataflow as a task graph and evenly distributes tasks over the cluster to balance the load.

As shown in line 28, once the data transformation is performed, TSets can be converted to a different data format like NumPy such that it can feed to a different library to perform further processing. If Twister2 processes are bootstrapped using an MPI implementation, intermediate data can be directly processed using MPI API.

```
1  from mpi4py import MPI
2  import numpy as np
3  from twister2 import Twister2Environment
4
5  env = Twister2Environment(resources=[{"cpu": 4,
6              "ram": 4096, "instances": 4}])
7  class PersonSource(SourceFunc):
8      def next(self):
9          # generate person tuple
10         return ["id", "name", "address"]
11 class VaccinationSource(SourceFunc):
12     def next(self):
13         # generate vaccination info. tuple
14         return ["person_id", "doses"]
15 def join_logic(student, result, ctx):
16     return student["id"] == result["person_id"]
17                     and result["doses"] == 2
18
19 people = env.create_source(PersonSource(),
20             parallelism = 10)
21 vaccination = env.create_source(
       VaccinationSource(),
22             parallelism = 10)
23 # finding people who have received two doses.
24 # this involves entire population, might
25 # spill to the disk
26 fully_vaccinated = people.join(vaccination,
27             join_type=INNER, on=join_logic)
28 people_ids_split = fully_vaccinated.select("id")
       .toNumpy()
29
30 # switching to use mpi directly
31 this_worker_total = people_ids_split.size
32 global_total = MPI.COMM_WORLD.allreduce(total,
       op=MPI.SUM)
33 people_ids = numpy.zeros(global_total, dtype=np.
       integer)
34 MPI.COMM_WORLD.allgather(people_ids_split,
       people_ids)
35 env.finalize()
```

Fig. 13. Twister2 TSet on MPI

1) Multidimensional Scaling: Multidimensional scaling (MDS) is a valuable tool in data scientists toolbox. Authors have previously developed an MPI based MDS algorithm [45]–[47] that can scale to large number of cores. The MDS algorithm expects a distance matrix and we need to calculate this matrix from an input dataset which can be large. This distance

matrix is partitioned row-wise and feed into the algorithm. We developed an application that combines the MPI based MDS algorithm and Twister2 based data processing to create the partitioned distance matrix as shown in Figure 14. This program runs executing table operators for data prepossessing and matrix operators for MDS algorithm. Further, Figure 15 shows the strong scaling performance of MDS algorithm (only the algorithm) on varying number of nodes with 32000 points. Spark and Flink implementations of the MDS algorithm are developed on their table abstractions and MPI version is developed with the array abstractions and operators.

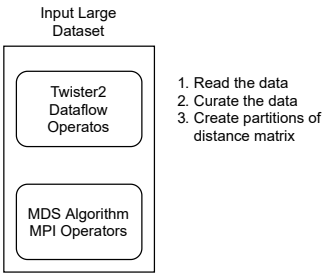

Fig. 14. Dataflow operators to prepossess data and MPI operators for Matrix manipulations in MDS algorithm.

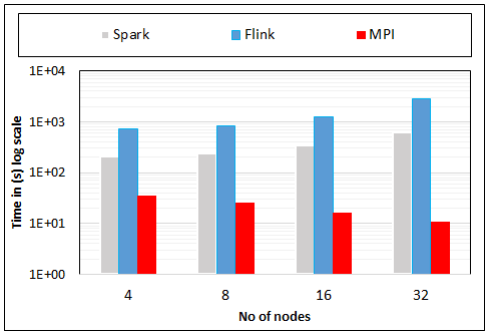

Fig. 15. MDS execution time with 32000 points on varying number of nodes. Each node runs 20 parallel tasks.

B. Cylon

Cylon [3], [48] provides a distributed memory DataFrame API on Python for processing data using a tabular format. Unlike existing state-of-the-art data engineering tools written purely in Python, Cylon adopts high performance compute kernels in C++, with an in-memory table representation. Cylon uses the Apache Arrow memory specification for storing table data in the memory. 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 uniform experience for the user. They are implemented as eager operators. The user can program with a global view of data by applying operations to them. Also, they can convert the data to local parallel processes and do in-memory operations as well. Cylon programs are SPMD-style programs and operators

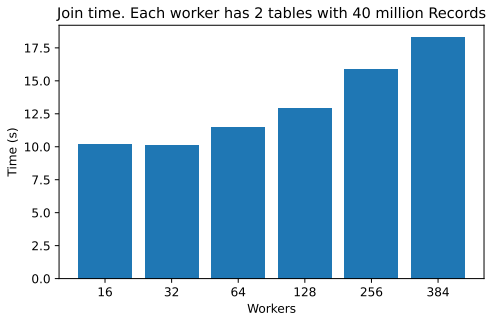

Fig. 16. Cylon join operator. Each worker is assigned two tables with 40 million records each. Each table has two 64bit integers columns.

are designed to work in that fashion. Figure 16 shows how Cylon Join operator can scale to large number of cores with increased load without sacrificing the performance. This test was done on a cluster with 8 nodes each with two Intel(R) Xeon(R) Platinum 8160 CPUs and 256GB of memory.

In Figure 17, transformed data can directly pipe to a different framework like PyTorch [49] to train ML/DL models or extract more valuable insights from the data. Line 18 of Figure 17 transforms Cylon DataFrame into a NumPy array. Similarly, Cylon can convert DataFrames to multiple other data formats, including Pandas DataFrames, Arrow Tables, and even copy/move to a different device such as GPUs. Cylon tries to handle such transformations as efficiently as possible by avoiding additional data copies or wasting CPU cycles solely for data format transformation. Cylon owes this capability to its high performance core written in C++ and the columnar data representation it internally uses to hold data in memory. All these capabilities integrate Cylon operators seamlessly with existing frameworks and libraries without compromising the performance or adding additional memory pressure.

When Cylon processes are bootstrapped with an MPI implementation, the application gets all the capabilities of the underlying MPI implementation, including access to the BSP-style collective communication API. As shown in line 39, the programmer can use AllReduce operation of MPI to synchronize the model across the distributed set of workers. In addition to writing such custom code to handle synchronization, if the integrated library has the native capability to use MPI (PyTorch, Horovod [50], TensorFlow [51], Keras [52] etc.), the programmer can use such capabilities since the process (Cylon Worker) already belongs to an MPI world. In addition, NVIDIA NCCL [53] provides a BSP mode of execution for model synchronization at scale for GPU devices for accelerated deep learning. Functionality in NCCL is similar to MPI. Distributed interfaces like that of PyTorch have been specifically designed to provide a unified interface for accelerated deep learning on various accelerators. Additionally, Horovod also extends to GPU-based AllReduce for deep learning models. Thus Cylon can be seamlessly integrated not only for CPU-based MPI model synchronization, but GPU-

based model synchronization as well.

```
import numpy as np
from mpi4py import MPI
from pycylon import DataFrame, read_csv, CylonEnv
from pycylon.net import MPIConfig

# preprocessing data using Cylon
env = CylonEnv(config=MPIConfig())
# load people
people = read_csv("people.csv", slice=True, env=env) # [id, severity]
# load vitals
vitals = read_csv("vitals.csv", slice=True, env=env) # [id, type, value]
# consider only temperature
temp_of_people = vitals.where(vitals["type"] == "TEMP")
people = people.set_index(["id"])
temp_of_people = temp_of_people.set_index(["id"])
# join temperature to people
joined = temp_of_people.join(people, how="left")
numpy_arr = joined.to_numpy()
# Create random input and output data
x = numpy_arr.T[1] # temperature
y = numpy_arr.T[3] # severity
# Randomly initialize weights
w = np.random.rand(4)

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = w[0] + w[1] * x + w[2] * x ** 2 + w[3] * x ** 3
    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grads = mp.array([grad_y_pred.sum(), (grad_y_pred * x).sum(),
        (grad_y_pred * x ** 2).sum(), (grad_y_pred * x ** 3).sum()])
    # Update weights
    w -= grads * learning_rate
    # synchronizing the model parameters using MPI
    w = np.array(MPI.COMM_WORLD.allreduce(w, op = MPI.SUM))/MPI.COMM_WORLD.Get_size()

env.finalize()
```

Fig. 17. Cylon interoperability with Pytorch and MPI

C. Global Data & Asynchronous Operators

The data model in Apache Spark [1] is based on Resilient Distributed Datasets (RDDs). RDD is an abstraction to represent a large dataset distributed over the cluster nodes. The logical execution model is expressed through a chain of transformations on RDDs by the user. A graph created by these transformations is termed the *lineage graph*. It also plays an important role in supporting fault tolerance.

Dask [21] is similar to Spark in its execution and data model. The major difference is that Dask is implemented on top of Python as opposed to Java in Spark. Modin [20] is a framework designed for scaling Pandas DataFrame-based applications. Modin allows Pandas DataFrame-based transformations to scale to many cores.

Apache Spark, Dask and Modin all provide a global view of data. The user cannot access a parallel process and must program functions that are applied to the partitions of the global data. Also, they are executed with the asynchronous execution model we showed earlier. This makes these systems incompatible with the *HPTMT* architecture.

IX. Conclusions

This paper exploits the HPTMT principle that efficient distributed operators designed around a collection of key data abstractions enables an environment supporting high performance data-intensive applications at scale. We noted that many successful systems have used this principle but it is typically applied to a subset of available operators and data abstractions. We gave details of the HPTMT architecture for developing distributed operators independent of any framework that can work together using many orchestration (workflow) systems. We introduced the Cylon project, which provides eager operators on tables and the capability to use MPI's array (matrix)-based operators. This also provides an efficient bridge between different languages (C++, Python, Java) and links to the Java Twister2 project that provides DataFlow operators supporting this architecture. We believe that we have implemented enough operators across a range of application domains to show that one can achieve efficient parallel HPTMT across the thousands of operators found by aggregating scientific computing, Spark and Flink Big Data, NumPy, Pandas, Database, and Deep Learning. Our work focused on traditional CPU hardware but NVIDIA, Intel, and others are applying the HPTMT architecture to GPUs and accelerators. In future work, we will continue to explore more operators and their use across a variety of important hardware platforms. We will test these ideas on applications to verify end-to-end performance and that the operator-based programming model is indeed expressive enough. We can expect that we and others will discover the need for further operators as application experience develops. Apache Arrow and Parquet provide important tools for HPTMT and we are exchanging ideas with them.

Acknowledgments

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. We thank the FutureSystems team for their infrastructure support.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[3] C. Widanage, N. Perera, V. Abeykoon, S. Kamburugamuve, T. A. Kanewala, H. Maithree, P. Wickramasinghe, A. Uyar, G. Gunduz, and G. Fox, "High performance data engineering everywhere," in *2020 IEEE International Conference on Smart Data Services (SMDS)*. IEEE, 2020, pp. 122–132.

[4] G. Fox, "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*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/3147213.3155012

[5] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *Sourcebook of parallel computing*. Morgan Kaufmann Publishers San Francisco^ eCA CA, 2003, vol. 3003.

[6] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen, "Hpjava: data parallel extensions to java," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 873–877, 1998.

[7] E. Johnson and D. Gannon, "Hpc++ experiments with the parallel standard template library," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 124–131.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[9] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund *et al.*, "The fortress language specification," *Sun Microsystems*, vol. 139, no. 140, p. 116, 2005.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[11] S. Imam and V. Sarkar, "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*, 2014, pp. 75–86.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[15] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[16] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[17] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[18] "Accelerate Fast Math with Intel® oneAPI Math Kernel Library." [Online]. Available: https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

[19] "Open GPU Data Science." [Online]. Available: https://rapids.ai/

[20] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems," *arXiv preprint arXiv:2001.00888*, 2020.

[21] M. Rocklin, ""dask: Parallel computation with blocked algorithms and task scheduling"," in *Proceedings of the 14th python in science conference*, no. 130-136. Citeseer, 2015.

[22] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. 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.

[23] "Apache Arrow." [Online]. Available: https://arrow.apache.org/

[24] "Apache Parquet." [Online]. Available: https://parquet.apache.org/

[25] G. C. Fox, R. D. Williams, and G. C. Messina, *Parallel computing works!* Elsevier, 2014.

[26] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1235–1246.

[27] Y. N. Babuji, K. Chard, I. T. Foster, D. S. Katz, M. Wilde, A. Woodard, and J. M. Wozniak, "Parsl: Scalable parallel scripting in python." in *IWSG*, 2018.

[28] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[29] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[30] [Online]. Available: https://argoproj.github.io/argo-workflows/

[31] [Online]. Available: https://www.kubeflow.org/

[32] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 10:70–10:93, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2898442.2898444

[33] "MPI: A Message-Passing Interface Standard Version 3.0," 2012, Technical Report. [Online]. Available: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[34] U. Wickramasinghe and A. Lumsdaine, "A survey of methods for collective communication optimization and tuning," *arXiv preprint arXiv:1611.06334*, 2016.

[35] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, no. 9, pp. 1–9, 2011.

[36] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 517–528, 2017.

[37] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[38] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[39] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[40] S. Kamburugamuve, P. Wickramasinghe, K. Govindarajan, A. Uyar, G. Gunduz, V. Abeykoon, and G. Fox, "Twister:net - communication library for big data processing in hpc and cloud environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 383–391. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00055

[41] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. 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.

[42] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *ACM Sigmod Record*, vol. 34, no. 3, pp. 44–49, 2005.

[43] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[44] P. Wickramasinghe, S. Kamburugamuve, K. Govindarajan, V. Abeykoon, C. Widanage, N. Perera, A. Uyar, G. Gunduz, S. Akkas, and G. Fox, "Twister2: Tset high-performance iterative dataflow," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE, 2019, pp. 55–60.

[45] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 1–8.

[46] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox, "Java thread and process performance for parallel machine learning on multicore hpc clusters," in *2016 IEEE international conference on big data (Big Data)*. IEEE, 2016, pp. 347–354.

[47] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake, and G. C. Fox, "Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 61–73, 2018.

[48] V. Abeykoon, N. Perera, C. Widanage, S. Kamburugamuve, T. A. Kanewala, H. Maithree, P. Wickramasinghe, A. Uyar, and G. Fox, "Data engineering for hpc with python," in *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, 2020, pp. 13–21.

[49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[50] A. Sergeev and M. Del Balso, ""horovod: fast and easy distributed deep learning in tensorflow"," *arXiv preprint arXiv:1802.05799*, 2018.

[51] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[52] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[53] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, 2017.