Integrating Bulk-Synchronous Parallel and Distributed Asynchronous Models for Efficient High Performance Data Engineering

Niranda Perera
Intelligent Systems Engineering
Indiana University Bloomington

March 30, 2021

Abstract

The emergence of Big Data, Artificial Intelligence (AI), and Machine Learning (ML) demand more efficient and high performant data engineering frameworks. The Bulk Synchronous Parallel (BSP) model of execution has been around for more than four decades and has been widely adopted by the HPC community. Vis-a-vis, recent Big Data analytics frameworks have adopted a distributed asynchronous computing approach, given its advantages in commodity cloud infrastructure. This report attempts to exploit opportunities to bridge these computing models together and creating a more efficient high-performance data engineering pipeline.

1 Introduction

Data engineering is becoming (have already become) an increasingly important element in both scientific enterprise and research communities. The exponential growth of data generation in every aspect of society is a major contributor to this *Big Data* revolution. The emergence of Artificial Intelligence (AI) and Machine Learning (ML), has further exacerbated the need to come up with efficient and high performant data engineering approaches.

Extract-Transform-Load (ETL) logic is at the heart of data engineering. Data processing systems can be broadly categorized into 1. batch, 2. streaming, 3. graph, and 4. AI/ML. Each system comes with its own data formats, storage, and transformation/ analysis routines. Modern applications require resources beyond a single node's ability to provide. However, this is just a small part of the issues facing the overall data processing environment, which

must also support a raft of data engineering for pre- and post-data processing, communication, and system integration.

This report aims at examining a decades-old execution paradigm, "Bulk Synchronous Parallel (BSP)" and integrate it with the widely used "Distributed Asynchronous" execution, to create efficient high-performance data engineering pipelines. While BSP is still being heavily used in high-performance computing applications (MPI), there exists a presumption that BSP-style programming is not so convenient for the emerging data science community. Similarly, the HPC community has not welcomed fully asynchronous execution, citing performance implications. Hence, the report also aims at providing a specification for distributed data abstraction for BSP-like execution environments, that could provide modern user-friendly APIs without affecting performance aspects. The report also investigates the workflow management (WFM) frameworks, that have been mixing BSP and distributed asynchronous execution. This has been successfully adopted even in HPC WFM frameworks. But, as of current literature available, there has been little effort gone into bridging these models together in data engineering.

The report is organized as follows. Section 2 discusses both the execution paradigms together with workflow management, which is closely related to the same domain. Section 3 discusses commonly used data abstractions that are being currently used. Next, Section 4 looks at emerging technology trends that could encourage integrating the said paradigms. Given such advancements in the domain, Section 5 discusses the opportunity for such an integration. And lastly, Section 6 presents a survey of existing frameworks that covers the latest developments in the said domains.

2 Execution Models

Parallel problem classes can be categorized into the following.

- Synchronous - Tightly coupled. Software exploits features of the problem structure to get good scaling.
- Loosely synchronous - Similar to synchronous, but performs on non-identical data elements
- Asynchronous - Functional parallelism that irregular in space and time.
- Embarrassingly parallel - Independent execution of disconnected components.

There have been various computation frameworks designed to handle these problem classes. Bulk Synchronous Parallel and Distributed Asynchronous models have been the most widely used models thus far.

2.1 Bulk Synchronous Parallel Model (BSPM)

Many real-life problems can be categorized into *Loosely synchronous parallel* computations. 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" [1]. Later, a similar idea was published in an article by Valiant, L [2] in 1990 which introduced the term *"Bulk Synchronous Parallel"*. It was proposed as a *bridging model* for designing parallel algorithms. (Eventually, the scientific community has adopted the latter term "BSP" and it will be used in this report throughout).

In the loosely / bulk synchronous parallel model, processors work on independent data elements and communicate in parallel, but they arrive at a *barrier* synchronization at some point in the application. This model is widely adopted in the High-Performance Computing (HPC) community. The most popular specification that champions this idea is the Message Passing Interface (MPI).

Almost all the HPC workloads adopt BSPM, and lately, it has been the same model used by data-parallel distributed deep learning (DL) applications. For an example, Horovod DL framework[3] for distributed TensorFlow, and PyTorch Distributed package [4]. Additionally, the same approach is used in big data analytics tools such as Twister2 [5], and Cylon [6].

2.2 Distributed Asynchronous Model (DAM)

The Distributed Asynchronous/ fully distributed model also became very popular with the popularization of web services and distributed systems architecture. It was powered by the commodity cloud and networking infrastructure, which brought massively parallel computing to enterprise applications, a capability that was previously limited to supercomputers.

In a distributed asynchronous model, there is a separate client process that will be submitting *tasks* to a cluster of processes. These *tasks* could be an independent piece of computation or communication, and usually, there is a scheduler process (centralized or distributed) that will be managing the cluster and its tasks.

This model can be considered the cornerstone of Big Data analytics infrastructure and distributed database systems (RDBMS and NoSQL). It paved way to many data analytics frameworks, such as Hadoop [7] and MapReduce [8], Apache Spark [9], Apache Flink [10], etc. Furthermore, DAM has been widely adopted by distributed execution frameworks such as Dask [11], Ray [12], and also by distributed actor frameworks such as CAF [13], Akka [14].

DAM model enables some important features that were challenging to implement by its predecessor models, the most important being fault tolerance. It also promotes the effective usage of shared computing resources in multi-tenant environments.

2.3 BSPM vs DAM

Following are the key differences between BSPM and DAM.

Area	BSPM	DAM
Focus	Computation, Synchronization focused	Communication, Concurrency focused
Synchronization	Loosely synchronous - communication operations can still be async but the resources would be dedicated to the job until completion	Fully asynchronous
Scheduling	Tightly coupled to hardware with fixed parallelism (no scheduler)	Loosely coupled with hardware. Schedulers have freedom to schedule tasks anywhere.
Data partitions	Data partitions are restricted to parallelism	Explicitly supports any number of data partitions
Handling messages	Efficient communications – Receiver immediately receives the message from the sender	Sent messages needs to be stored until the receiving task is scheduled (mailboxes)
Performance	High performance (time per workload is reduced)	High throughput (number of workloads per given time is increased)
Suitability	Suitable for workloads with high computation overheads	Suitable for workloads with high communication overhead
Fault tolerance	Does not support fault-tolerance inherently	Supports fault-tolerance
Hardware allocation	Requires fixed hardware allocation for a workload	Supports dynamic hardware allocation
Multi-tenancy	Multi-tenant workloads would require separate hardware / dedicated execution	Supports multi-tenant execution on the same cluster / shared execution
Hardware requirement	Suitable for dedicated hardware	Suitable for cloud environments, with preemptive resources

Table 1: BSPM vs DAM

2.4 Workflow Management (WFM)

Workflow management has also been an important component of computational and data-driven research practices, and some WFM predates the Big Data phenomenon. Some notable scientific workflow management systems are Pegasus, Kepler, Apache Airflow, Luigi, etc.

WFM frameworks primarily adopt the DAM for execution, and the most notable feature amongst all of these frameworks is that they arrange the workflow on a directed acyclic graph (DAG). DAG-based executions are also being heavily used in *dataflow programming paradigm*, which is an integral component in the data engineering frameworks (ex: Apache Spark, Twister2, Apache Flink, Apache Beam, TensorFlow, PyTorch).

Lately, data engineering pipelines have become extremely complex that they are required to manage a variety of data formats, storage, data extraction, transformation, and data movements. Hence, some data analytics frameworks have promoted themselves to orchestrate heterogeneous analytics workloads. For example, Spark 3.0 scheduler can now schedule

GPU-accelerated ML and DL applications on Spark clusters with GPUs, removing bottlenecks, increasing performance, and simplifying clusters [15].

3 Data Abstractions

Data abstractions play a vital role in data engineering. Each data processing workload (batch, streaming, graph, etc) has its own data abstraction. It is important to chosse the appropriate data abstraction while implementing a data engineering framework.

3.1 Arrays and Matrices

Arrays and Matrices are the core data abstraction used in linear algebraic tool-kits, such as Basic Linear Algebra Subroutines (BLAS), Linear Algebra Package (LAPACK), NumPy, etc. These are homogeneously typed, contiguous data structures. Matrices may be arranged in column- or row-major order in memory.

3.2 Tables, DataFrames, and DataSets

Tables are at the heart of data engineering. This is undoubtedly the most widely adopted data structure in the world, because it is the core data structure used in databases management systems (both relational and NoSQL). Tables are heterogeneously typed data structures which could contain a wide variety of data types. Frameworks may choose to employ row- or column-major memory representation based on the application, ex: a transaction processing framework may store data in row format, while analytical processing framework may use columnar format.

DataFrames were originally introduced in S programming language in Bell Labs, which is synonymous to tables. They were open-sourced by R language and then widely popularized by Python Pandas [16] for data analytics. *DataSets* are an extension to DataFrames, by adding a strongly-typed schema and operations. Both DataFrames and DataSets provide a *functional programming* flavor to data analytics compared to rigid SQL routines. Hence they has been widely used interactive data analytics.

Apache Spark [9] is the most popular implementation of distributed dataframes, while Dask [11] and Modin [17] are becoming increasingly popular due its compatibility with Python Pandas Dataframe. CuDF [18] and Dask-CuDF is a GPU based implementation of dataframes.

3.3 Tensors

Tensors are the main data abstraction for AI and ML frameworks. It is a multidimensional array with a uniform data type. Mathematically, scalars, arrays, matrices, all are tensors. Tensors are categorized differently because, the AI/ML frameworks such as TensorFlow, PyTorch, MXNet, etc, use implementation-specific memory representations. DLPack [19] is an open in-memory tensor structure to for sharing tensor among frameworks.

4 Emerging Trends

In the past few decades, processor and networking hardware have achieved significant improvements. Processor architectures have reached the end of Moore's Law for a single processor, but continuing to add more and more computing elements into a single processing unit. This is seen in both CPU and GPU architectures, as well as other hardware like FPGA's.

Computer network speeds have also improved significantly in the past decade. Specialized network hardware is being designed for accelerators such as NVLink and NVSwitch for GPUs, InfiniBand networks for compute nodes, and specialized software is also being developed to get the best out of these new network hardware (ex: Remote Direct Memory Access (RDMA)).

Analytic workloads are also becoming increasingly complex and resource-intensive. As a consequence, much more complex compute nodes are becoming publicly available such as Nvidia DGX systems with multi-CPU and GPU units.

Cloud infrastructure providers (ex: Amazon AWS, Google Cloud Platform, Microsoft Azure) are also providing dedicated hardware on-demand.

5 Opportunity For Improvement

As described in Section 4, these emerging technological advancements, open up new opportunities for improvement in the high-performance data engineering domain.

One key observation here is, that there is *no one-stop-shop model for data engineering applications*. For example,

- *Big company managing a company-wide Spark cluster and employees are submitting their queries to it* $\rightarrow$ *DAM*
- *Spawning a distributed Spark cluster in a dedicated DGX node/ cluster to perform a data preprocessing task* $\rightarrow$ *BSPM*
- *Running distributed hyper param search of a DNN in a multi-GPU node/ cluster* $\rightarrow$ *DAM + BSPM*

Another observation is, that distributed AI/ML/DL workloads are almost always following BSPM. But, most of the popular big data preprocessing frameworks are using a DAM. Hence, the data engineers would have to run separate distributed preprocessing and DL pipelines and orchestrate them with a workflow manager.

There are some inadvertent uses of DAM and BSPM together in processing pipelines. For example, a CPU process submits multiple GPU computation kernels via multiple 'streams'. Here, the CPU process is synonymous with DAM while the interaction of GPU streams is synonymous with BSPM.

5.1 Bridging BSPM and DAM

According to the current literature, no framework has been able to bring the BSPM and DAM models together. Each model has its own merits, and therefore, both paradigms are required for complex data analytics jobs and effective use of modern hardware (and advancements).

There is an opportunity to propose a framework that supports both BSPM and DAM execution. In such a system, a 'task' would be executed in BSPM fashion with dedicated resource allocation, while 'tasks' are scheduled in DAM fashion that would effectively schedule resources (similar to WFM).

Consequently, there are data engineering workloads (SQL-like workloads) that could benefit from BSPM execution than the existing DAM execution in data analytics frameworks (provided that there are sufficient resources available). This would be valuable for high-performance data engineering.

Furthermore, there is an opportunity to propose a *Dataframe specification for MPI-like environments*, which is currently not available. There are emerging dataframe specifications for DAM-based models (ex: Modin), but the BSPM execution is left behind because there is a perception in the data engineering community that *BSPM way of programming is inconvenient!*.

All in all, such a system would create a coherent ecosystem for distributed data engineering and deep learning workloads.

5.2 Possible Outcomes

- High performance data engineering framework that integrates BSPM and DAM
- Develop a distributed dataframe abstraction for MPI-like environments
- Extend data engineering to hardware platforms other than CPU and GPU by supporting OpenCL and Intel One API
- Use the proposed framework to support other hardware accelerators such as FPGA, TPU, etc.

5.3 Challenges

While the idea seems to be promising, several challenges need to be addressed.

- Load imbalance in partitions is a major challenge in the distributed data processing. BSPM is more susceptible to it because it cannot arbitrarily increase the number of partitions (which could be a way to mitigate the issue to a certain degree.)
- There are several papers published on server-less AI/ML/DL applications, and they claim that going serverless could be cheaper. The question remains if this would compel the DL community to move into the DAM approach.
- Recent researches have shown that there are irregular variations in inter-node network performance, especially in cloud service providers [20][21]. This is caused by the physical layout of servers, racks, etc. Therefore, implementing a *pure clean BSP execution* would not be a straightforward exercise.
- How to support computation and communication overlap in the BSPM model? This can be supported in BSPM using a task-like model without compromising the distributed-memory semantics
- Someone may argue that "Data preprocessing is a one-time task!, So it doesn't matter if it is inefficient or not!". (This has been partially addressed by the claim that, 80% of data scientist's time is spent on pre-processing data)
- Theoretically, it can be argued that BSPM is a subset of DAM. While this is a valid argument, the existing data analytics tools do not have explicit schedulers for BSPM

style workloads. But, it would be possible to extend current schedulers to provide this functionality.
- Data analytics workloads are mostly SQL-like workloads and big data frameworks have done a decent job in this space. There are also GPU accelerated extensions are being introduced to them. Therefore, it would be required to show a considerable upside by integrating BSPM and DAM.

6 Survey on Existing Frameworks

To make the most out of the opportunities described in Section 5, it is imperative to carry out a detailed survey of existing frameworks.

- Dask - DAM execution framework and a distributed dataframe abstraction
- Ray - DAM execution framework designed for Reinforcement Learning (RL)
- Horovod - BSPM execution framework for distributed DL
- Modin - distributed dataframe abstraction
- CuDF and Dask-CuDF - distributed dataframe abstraction for GPUs
- Apache Spark and Rapids Acceleration - WFM-like orchestration of both data processing and DL workloads
- Parsl - WFM framework for heterogeneous resources

6.1 Dask - Rocklin M. (2015) [11]

Dask is a specification that encodes task schedules with minimal incidental complexity using terms common to all Python projects, namely dicts, tuples, and callables. Ideally, this minimum solution is easy to adopt and understand by a broad community. It represents a computation as a directed acyclic graph of tasks with data dependencies. It attempts to provide a software solution to parallelize the scientific python software stack without triggering a full rewrite. Following are the main components in Dask.

- Dask graph - Dictionary mapping identifying keys to values (or tasks)
- Key - Any hashable value that is not a task
- Task - A tuple with a Python callable. Represents atomic units of work meant to be run by a single worker

Dynamic task scheduling considers graph creation and graph execution are separable problems. Currently, it provides *single-threaded, multi-threaded, multi-process, and distributed schedulers*. Schedulers determine execution order during execution rather than ahead of time through static analysis.

According to the authors, most dynamic task schedulers (Luigi, DAGuE, Spark, Dryad) are not suitable for blocked algorithms in shared memory computation. It requires *lightweight, easily installable in Python, low latency, and memory-efficient task scheduling*. Another important feature in Dask schedulers, is they are extensible.

Following Dask collections are currently available.

- Array - Implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays.

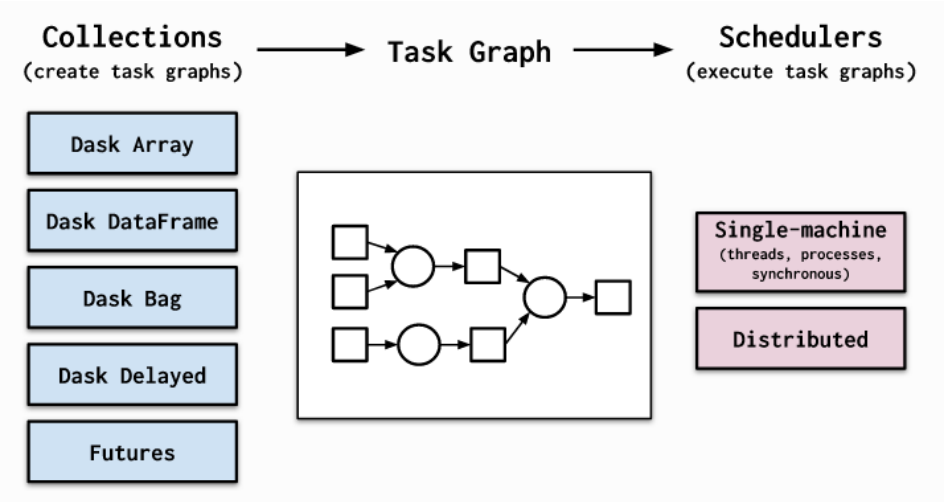


Figure 1: Dask execution

- Bag - Implements operations like map, filter, fold, and groupby on collections of generic Python objects.
- Dataframe - Large parallel DataFrame composed of many smaller Pandas DataFrames, split along the index.
- Delayed - Operates Python functions lazily, and arranges them in a DAG
- Futures - Extends Python's concurrent.futures interface for lazy operations

Dask Distributed

Dask distributed is a centrally managed, distributed, dynamic task scheduler. The central dask-scheduler process coordinates the actions of several dask-worker processes spread across multiple machines and the concurrent requests of several clients. It is asynchronous and event-driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers.

Workers communicate amongst each other for bulk data transfer over TCP. Internally the scheduler tracks all work as a constantly changing DAG of tasks. This graph of tasks grows as users submit more computations, fills out as workers complete tasks, and shrinks as users leave or become disinterested in previous results.

6.2 Ray - Berkeley (2018) [12]

Ray implements a unified interface that can express both task-parallel and actor-based computations, supported by a single dynamic execution engine. It employs a distributed scheduler and a distributed and fault-tolerant store to manage the system's control state and focuses on reinforced learning (RL).

A system for RL must support fine-grained computations, must support heterogeneity both in time and in resource usage, and must support dynamic execution, as results of simulations or interactions with the environment can change future computations. Existing frameworks that have been developed for Big Data workloads or for supervised learning

workloads fall short of satisfying these new requirements for RL. Hence the authors propose Ray, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications.

Tasks enable Ray to efficiently and dynamically load balance simulations, process large inputs and state spaces (e.g., images, video), and recover from failures. In contrast, actors enable Ray to efficiently support stateful computations, such as model training, and expose shared mutable states to clients, (e.g., a parameter server).

Ray implements the actor and the task abstractions on top of a single dynamic execution engine that is highly scalable and fault-tolerant. It distributes two components that are typically centralized in existing frameworks, 1. task scheduler; 2. Metadata store which maintains the computation lineage and a directory for data objects.

Ray provides lineage-based fault tolerance for tasks and actors and replication-based fault tolerance for the metadata store. To achieve scalability and fault tolerance, the authors propose a system in which the control state is stored in a sharded metadata store and all other system components are stateless. To achieve scalability, we propose a bottom-up distributed scheduling strategy.

RL applications must provide efficient support for,

- Training - Distributed SGD typically relies on an allreduce aggregation step or a parameter server
- Serving - Uses the trained policy to render an action based on the current state of the environment.
- Simulations - Evaluate the policy

Ray model

Ray implements a dynamic task graph computation model (models an application as a DAG of dependent tasks that evolves during execution)

Tasks (stateless)	Actors (stateful)
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

Figure 2: Ray tasks vs actors

Architecture

Major components - 1. Global control store (GCS); 2. Distributed scheduler, and 3. Distributed object store (All components are horizontally scalable and fault-tolerant).

Global control store (GCS) is a Redis KV store with replication. Object metadata is stored in the GCS rather than in the scheduler, fully decoupling task dispatch from task scheduling (makes the scheduler stateless). Provides debugging, profiling, and visualization tools.

Distributed Scheduler is a two-level scheduler, global and per-node local. Tasks are first accepted by the local scheduler and if it decides not to schedule the task, it forwards to the

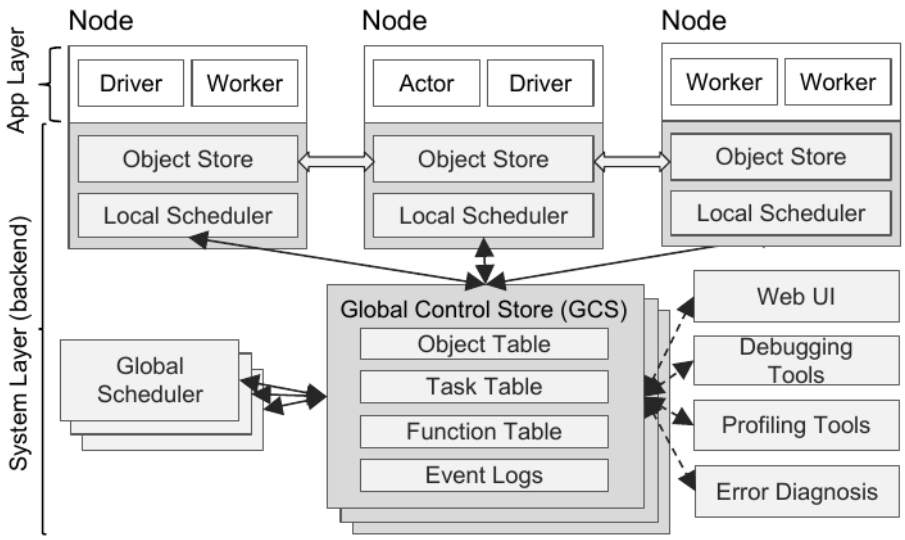


Figure 3: Ray major components

global scheduler. Then the global scheduler schedules it in some other node. Scheduler states are managed in GCS

In-Memory Distributed Object Store is an Apache Arrow Plasma store distributed storage system to store the inputs and outputs of every task, or stateless computation.

Ray is closely related to CIEL and Dask. All three support dynamic task graphs with nested tasks and implement the futures abstraction. It is not a framework for data analytics, but they have taken some preliminary efforts to provide a distributed Pandas Dataframe abstraction on top of Ray execution.

6.3 Horovod - Uber (2018) [3]

Horovod is an open-source library that improves on obstructions to scaling distributed TensorFlow applications by, 1. employing efficient inter-GPU communication via ring reduction, and 2. requiring only a few lines of modification to user code, enabling faster, easier distributed training.

The standard distributed DL TensorFlow package runs with a parameter server approach to averaging gradients. This entails the following issues: 1. identifying the right ratio of worker to parameter servers, 2. handling increased program complexity.

Ring-AllReduce

2017 Baidu published a new all-reduce approach, *Ring-allreduce* which is bandwidth-optimal. Horovod implements ring-allreduce with NCCL. The authors show that this new all reduce algorithm performs better than OpenMPI v1.1, but it would have been better if a later version of OpenMPI was used for the comparison.

Distributed operation is handled by MPI (BSPM). Adds support for models that fit inside a single server, potentially on multiple GPUs, whereas the original TF version only supported

models that fit on a single GPU.

Horovod also provides a timeline, a profiling tool, and Tensor Fusion, an algorithm that fuses tensors before calling the ring-allreduce.

Authors show that Horovod scales well on both plain TCP and RDMA-capable networks, and they have planned to implement very large models using the framework.

6.4 Modin (2020) [17]

Authors identify issues with dataframes as, 1. dataframes face performance issues even on moderately large datasets, and 2. significant ambiguity regarding dataframe semantics. Hence the authors propose Modin, a scaled-up implementation of Python pandas. They also introduce a simple data model and algebra for dataframes.

Dataframes provide a functional interface that is more tolerant of unknown data structure and well-suited to developer and data scientist workflows, including REPL-style imperative interfaces and data science notebooks. It is an intuitive data model that embraces an implicit ordering on both rows and columns and treats them symmetrically. It is also a query language that bridges relational (e.g., filter, join), linear algebra (e.g. transpose), and spreadsheet-like (e.g., pivot) operators. Also, it is an incrementally composable syntax that encourages easy and rapid validation and iterative evaluation of queries.

Padas has a rich set of APIs, but they have significant redundancies. There's no query plan, hence can not be optimized based on the entire application. Modin aims to overcome these by providing a distributed scalable dataframe system on large datasets. It rewrites pandas API calls into a sequence of operators in a new, compact dataframe algebra.

The authors propose a superset of operators called "dataframe algebra" that could span Pandas 240+ operators. Parallel execution is done using partitions. Each partition is then processed independently by the execution engine, with the results communicated across partitions as needed.

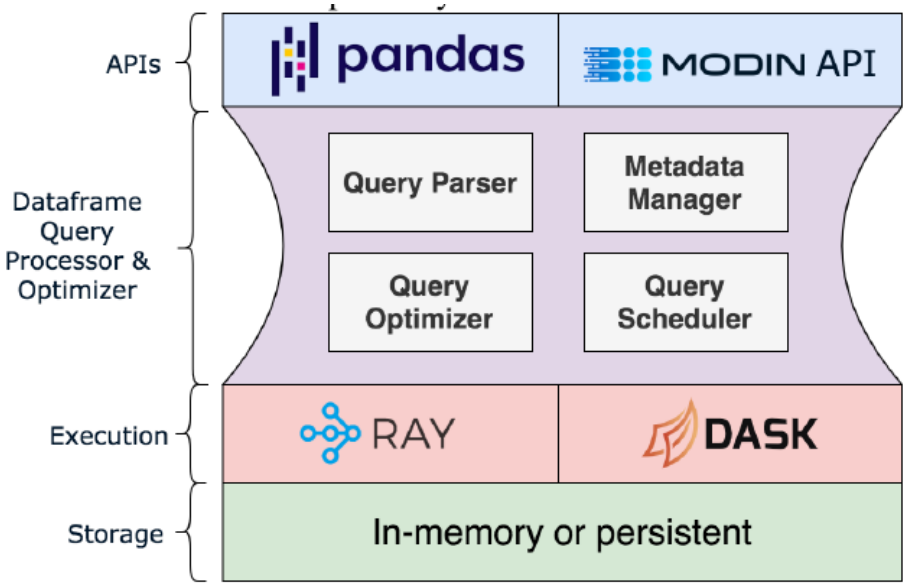


Figure 4: Modin Architecture

Dataframe Characteristic	Relational Characteristic
Ordered table	Unordered table
Named rows labels	No naming of rows
A lazily-induced schema	Rigid schema
Column names from $d \in Dom$	Column names from att [17]
Column/row symmetry	Columns and rows are distinct
Support for linear alg. operators	No native support

Figure 5: Dataframes vs RDBMS table

Dataframe Characteristic	Matrix Characteristic
Heterogeneously typed	Homogeneously typed
Both numeric and non-numeric types	Only numeric types
Explicit row and column labels	No row or column labels
Support for rel. algebra operators	No native support

Figure 6: Dataframes vs Matrices

Dataframe Algebra

Modin presents an extensive definition for dataframe operations - "dataframe algebra". The authors also analyze Dataframe usage stats of 1 million Jupyter notebooks from GitHub (Rule et al) to understand the usages of dataframes.

Operator	(Meta)data		Schema	Origin	Order	Description
SELECTION		×	static	REL	Parent	Eliminate rows
PROJECTION		×	static	REL	Parent	Eliminate columns
UNION		×	static	REL	Parent†	Set union of two dataframes
DIFFERENCE		×	static	REL	Parent†	Set difference of two dataframes
CROSS PRODUCT / JOIN		×	static	REL	Parent†	Combine two dataframes by element
DROP DUPLICATES		×	static	REL	Parent	Remove duplicate rows
GROUPBY		×	static	REL	New	Group identical attribute values for a given (set of) attribute(s)
SORT		×	static	REL	New	Lexicographically order rows
RENAME	(×)		static	REL	Parent	Change the name of a column
WINDOW		×	static	SQL	Parent	Apply a function via a sliding-window (either direction)
TRANSPOSE	(×)	×	dynamic	DF	Parent◇	Swap data and metadata between rows and columns
MAP	(×)	×	dynamic	DF	Parent	Apply a function uniformly to every row
TOLABELS	(×)	×	dynamic	DF	Parent	Set a data column as the row labels column
FROMLABELS	(×)	×	dynamic	DF	Parent	Convert the row labels column into a data column

Figure 7: Modin Dataframe Algebra - Operation Super-set

Data analysis workflow is broken into,

- Operator - single dataframe operation
- Statement - a set of operators
- Query - a set of statements arranged as a DAG
- Session - a complete end-to-end application

Modin Dataframes are maintaining a strict order but attempts to implement "physical data independence" by altering metadata rather than changing physical data. It also proposes to evaluate queries and return results in futures. Authors believe that this approach could use user "think time" for query evaluation.

Modin treats the dataframe data model and algebra as first-class citizens, as opposed to a means to enable distributed processing, addressing challenges in dataframe processing in systems like pandas and R at scale, while not sacrificing the convenient functionalities that have made dataframes so popular.

6.5 CuDF/ Dask-CuDF [18]

Built based on the Apache Arrow columnar memory format, cuDF is a GPU DataFrame library for loading, joining, aggregating, filtering, and otherwise manipulating data.

CuDF provides a pandas-like API that will be familiar to data engineers and data scientists, so they can use it to easily accelerate their workflows without going into the details of CUDA programming.

Dask-CuDF

cuDF is a single-GPU library. For Multi-GPU cuDF solutions Rapids developers use Dask and the dask-cudf package, which is able to scale cuDF across multiple GPUs on a single machine, or multiple GPUs across many machines in a cluster.

As discussed in the Section 6.1, Dask DataFrame was originally designed to scale Pandas, orchestrating many Pandas DataFrames spread across many CPUs into a cohesive parallel DataFrame. Because cuDF currently implements only a subset of Pandas's API, not all Dask DataFrame operations work with cuDF.

6.6 Apache Spark and Rapids Acceleration (2020) [15]

NVIDIA has worked with the Apache Spark community to implement GPU acceleration through the release of Spark 3.0 and the open-source RAPIDS Accelerator for Spark. It is believed that 80% of a data scientist's time is spent on data preprocessing and hence improving data preprocessing workflows is vital.

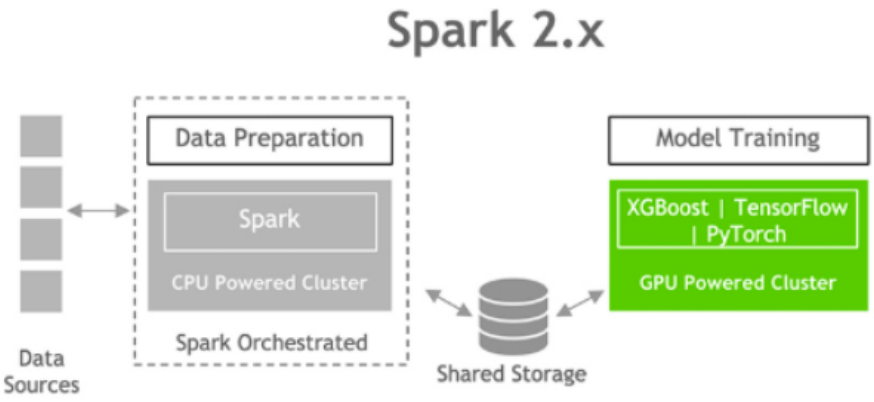


Figure 1. In Spark 2.x, separate clusters were needed for ETL on CPUs, and model training on GPUs.

Figure 8: Spark 2.x Execution

The Apache Spark community has been focused on bringing both phases of this end-to-end pipeline together so that data scientists can work with a single Spark cluster and avoid the penalty of moving data between phases. Spark 3.0 represents a key milestone, as Spark can now schedule GPU-accelerated ML and DL applications on Spark clusters with GPUs, removing bottlenecks, increasing performance, and simplifying clusters.

The RAPIDS Accelerator for Apache Spark enables applications to take advantage of GPU parallelism and high-bandwidth memory speed with no code changes, through the Spark SQL and DataFrame APIs and a new Spark shuffle implementation.

With the RAPIDS accelerator, the Catalyst query optimizer has been modified to identify operators within a query plan that can be accelerated with the RAPIDS API, mostly a one-to-one mapping. The new Spark shuffle implementation is built upon the GPU-accelerated Unified Communication X (UCX) library to dramatically optimize the data transfer between Spark processes. UCX exposes a set of abstract communication primitives which utilize the best of available hardware resources and offloads, including RDMA, TCP, GPUs, shared memory, and network atomic operations.

The Criteo Terabyte click logs public dataset, one of the largest public datasets for recommendation tasks, was used to demonstrate the efficiency of a GPU-optimized DLRM training pipeline. With eight V100 32-GB GPUs, processing time was sped up by a factor of up to 43X compared to an equivalent Spark-CPU pipeline.

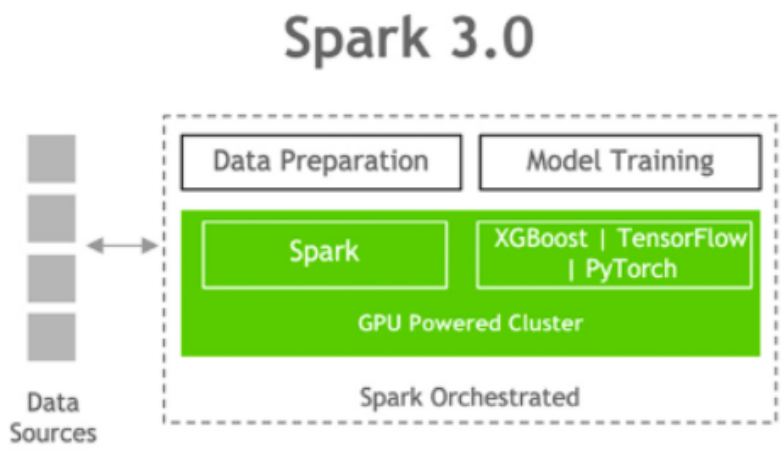


Figure 2. In Apache Spark 3.0, you can now have a single pipeline, from data ingest to data preparation to model training on a GPU powered cluster.

Figure 9: Spark 3.0 Execution

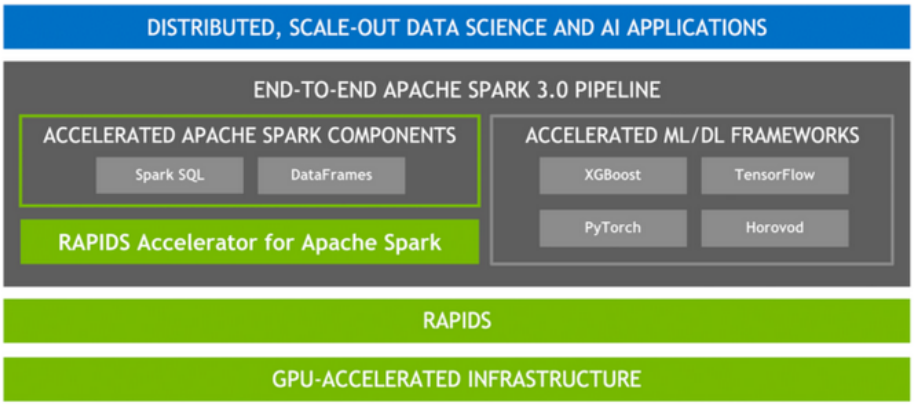


Figure 10: Spark Technology Stack

6.7 Parsl - Argonne NL (2018) [22]

Parsl (Parallel Scripting Library), is a Python library for programming and executing data-oriented workflows in parallel. It addresses the challenges of integrating workflow systems in science gateways.

Parsl identifies challenges associated with current scientific gateway approaches as, 1. many workflow engines are focused on many task applications rather than interactive, online, or machine learning analyses, 2. workflow engines are not easily integrated into external services (e.g., gateways) due to issues such as language mismatch and the need for intermediate workflow representations.

Parsl is built on the Swift workflow language model and brings parallel workflow capabilities to scripts, applications, and gateways implemented in Python. When a Parsl script is executed, the Parsl library causes annotated functions (Apps) to be intercepted by the Parsl execution fabric, which captures and serializes their parameters, analyzes their dependencies, and runs them on selected resources.

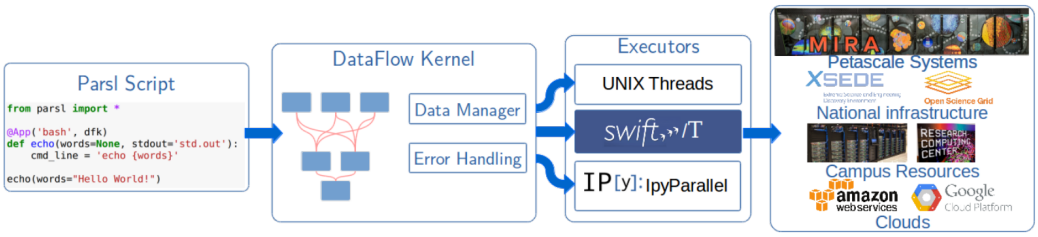
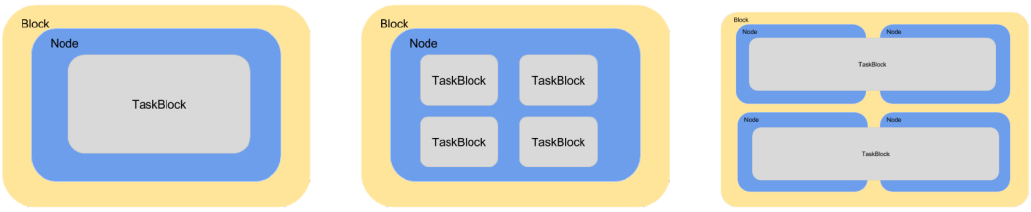


Figure 11: Parsl Architecture

Parsl scripts are decomposed into a simple dependency DAG by the DataFlow Kernel (DFK). The DFK provides a single lightweight abstraction on top of different execution resources. A Parsl script consists of standard Python code plus a number of Apps—annotated units of Python code or external applications that specify their input and output characteristics and that may be run in parallel.

Parsl Apps are completely asynchronous. When an App is invoked, there is no guarantee of when the result will be returned. Instead of directly returning a result, Parsl returns an *AppFuture*: a construct that includes the real result as well as the status and exceptions for that asynchronous function invocation.

When instantiating the DFK, developers specify the specific execution providers and executors that will be used for executing the parallel components of the script. Execution providers are simple abstractions over computational resources and executors provide an abstraction layer for executing tasks.



(a) A block comprised of a node with one TaskBlock.

(b) A block comprised of a node with several TaskBlocks.

(c) A block comprised of four nodes with two TaskBlocks.

Figure 12: Parsl Block Model - Resource allocation

7 Conclusion

This report discusses two main execution paradigms used in current distributed systems, Bulk Synchronous Parallel Model (BSPM) and Distributed Asynchronous Model (DAM). It proposes the idea of bridging these two models together to create efficient high-performance data engineering pipelines. It also recognizes that current Workflow Management (WFM) systems have been successful in using BSPM within a DAM environment. Furthermore, the report aims at providing a specification for distributed data abstraction for BSP-like execution environments that could provide modern and user-friendly APIs without losing performance. The report presents a survey of existing frameworks that encompasses the latest developments in the said domains to evaluate this idea.

References

[1] G. C. Fox, "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*, 1989, pp. 897–955.

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

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

[4] Pytorch distributed overview. [Online]. Available: https://pytorch.org/tutorials/beginner/dist_overview.html

[5] S. Kamburugamuve, K. Govindarajan, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Twister2: Design of a big data toolkit," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e5189, 2020.

[6] 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," *arXiv preprint arXiv:2007.09589*, 2020.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

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

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[10] 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.

[11] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, vol. 126. Citeseer, 2015.

[12] 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.

[13] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Caf-the c++ actor framework for scalable and resource-efficient applications," in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, 2014, pp. 15–28.

[14] Akka framework. [Online]. Available: https://akka.io/

[15] Accelerating apache spark 3.0 with gpus and rapids. [Online]. Available: https://developer.nvidia.com/blog/accelerating-apache-spark-3-0-with-gpus-and-rapids/

[16] 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, 2011.

[17] 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.

[18] Cudf gpu dataframes. [Online]. Available: https://docs.rapids.ai/api/cudf/stable/

[19] Dlpack: Open in memory tensor structure. [Online]. Available: https://github.com/dmlc/dlpack

[20] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," *arXiv preprint arXiv:1910.04940*, 2019.

[21] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training," 2020.

[22] 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.