Indiana University Bloomington

Indiana, United States

TOWARDS DATA ANALYTICS-AWARE HIGH PERFORMANCE DATA ENGINEERING AND BENCHMARKING

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

INTELLIGENT SYSTEMS ENGINEERING

by

Vibhatha Lakmal Abeykoon

2021

To: Martin Swany
Luddy School of Informatics, Computing and Engineering

This dissertation, written by Vibhatha Lakmal Abeykoon, and entitled Towards Data Analytics-aware High Performance Data Engineering and Benchmarking, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Minje Kim

Prateek Sharma

Ariful Azad

Geoffrey Fox, Major Professor

Date of Proposal:

The dissertation proposal of Vibhatha Lakmal Abeykoon is approved.

Martin Swany
Luddy School of Informatics, Computing and Engineering

Raj Achariya
Dean of the University Graduate School

Indiana University Bloomington, 2021

ABSTRACT OF THE DISSERTATION PROPOSAL

TOWARDS DATA ANALYTICS-AWARE HIGH PERFORMANCE DATA ENGINEERING AND BENCHMARKING

by

Vibhatha Lakmal Abeykoon

Indiana University Bloomington, 2021

Indiana, United States

Data analytics has become the centre of novel research and extensively growing industrial applications. With the rapid growth of data, such data analytics workloads have focused more on the high-performance computing (HPC) paradigm. In general, in a data pipeline the data engineering component holds the key to providing pre-processed data by operating on raw datasets. With the rapid growth of high-performance data analytical systems, data engineering frameworks have also shifted towards high-performance. Implementing data analytics-aware HPC data engineering operators are vital in providing scalable operators for HPC environments.

This thesis focuses on data engineering beneficial for HPC environments. The critical factor is to provide a compatible software stack utilizing HPC clusters efficiently. Mainly the HPC environments rely on efficient communication via MPI. In addition to this, designing optimized compute kernels allows using HPC resources efficiently. This thesis considers three main areas: data engineering operators, interoperability, and usability. Data engineering operators discuss a set of widely used operators in data engineering. Interoperability focuses on the ability to be used by existing data analytics and data engineering systems, and usability details how HPC-aware data engineering operators are made available for efficient data exploration using the widely used dataframe abstraction.

In data engineering, the most popular data abstraction is the dataframe. This thesis analyses in-depth a set of data engineering operators implemented to run on HPC resources, and these operators are exposed to the user in terms of the state-of-the-art dataframe abstraction, which involves less overhead in migrating an existing data engineering program to the introduced novel dataframe on HPC. The seamless integration between HPC data engineering operators and dataframe abstraction in Python is enabled via efficient language bindings designed using Cython. Applying Cython efficiently provides the ability to seamlessly integrate data structures across programming languages (C++ and Python). Compared to current sophisticated big-data systems, having this mode of operation offers the ability to execute efficiently on HPC environments.

In data analytics, the most widely used data structures are Numpy and Tensors. Seamless integration among data engineering data structures and data analytics data structures provides efficiency in data exploration research. Such tactics as well as existing data structures like Pandas dataframe also enable facilitation with current data engineering programmes. This thesis investigates how the developed HPC data engineering operators are performing compared to existing data engineering operators. Also, this thesis comprises scaling a scientific data analytics-aware data engineering workload deployed on PyTorch and Pandas in an HPC cluster using the introduced novel data engineering dataframe. A set of benchmarks is carried out to analyse the data engineering workload's performance on HPC clusters involving GPUs and CPUs for deep learning and CPUs for data engineering. Additionally, these benchmarks are packaged with a framework designed for scientific applications.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

CHAPTER 1

MOTIVATION

The modern day data analytics has become a vital component in logistics, e-commerce, health, security, transportation and many other scientific explorations. In the early days, the main component was very focused on developing algorithms to model such problems in an accurate way. But with the growth of the data these problems scaled into a much larger problem which was not only restricted to modelling, but to process the data efficiently to model vivid scientific curiosities and unknowns to a simply understandable expression. Modelling such problems relied on very accurate statistical models which were focused on various numerical modelling methods. But these statistical model evolved into a very structured form of data analytics domain called machine learning in the early part of the last two decades. Machine learning became a very powerful tool to solve a wide variety of problems very efficiently. Since the dawn of big data age, the data started growing rapidly and scientists needed to tools to process such big data sets efficiently. This was the dawn of big data systems which started to couple with machine learning workloads. With time, the machine learning models evolved into deep learning models which are very much sophisticated algorithms developed based on neural networks.

Many tools were developed to solve the problems associated with data processing for efficient data analytics. Data exploration tools like Pandas[M+11] has become a key tool in data processing for data analytics for smaller scale problems. For distributed data exploration data engineering frameworks like [das] was created on top of Pandas. Frameworks like Apache Spark[ZXW+16], Apache Flink[apaa], Apache Storm [IS15] and Apache Hadoop [apab] were created to provide data processing ability for streaming and batch computations. Individually these existing tools are built to perform on specific tasks like do data exploration at small scale or do data

processing in large scale. But the underlying core problem set is much deeper and it requires more involvement from distributed system researchers to built seamlessly integrating tools for data analytics aware data engineering.

Data analytics has also grown to larger scale problems based on two factors. Increasing data for analytics and model size. In early days, the model size did fit into a single machine so that using data parallelism was enough. With evolving problems and granularity of problems, scalability for data analytics has also become a vital key. Frameworks like PyTorch[PGM+19], Tensorflow [ABC+16] and MxNet [CLL+15] have become popular tools for data analytics. While frameworks like Horovod[SDB18] has become a popular tool to scale do data analysis at scale with many data analytics frameworks in a unified manner. Integrating these tools with data processing is a vital key to build scientific data pipelines.

Figure 1.1 shows the system overview for data analytics aware data engineering. The software stack associated with data analytics aware data engineering comprise of two sets of softwares focused on two goals. The data analytics software stack contains a set of algorithms specific for data analysis. Also frameworks build for this purpose support distributed computing on a high performance computing (HPC) compatible way. PyTorch is one of the leading bulk synchronous parallel (BSP) deep learning framework supporting distributed data parallel training. Tensorflow and MxNet also provides interfaces for extending sequential data analytics workload to run on multiple machines. To enhance the performance and provide unified software stack for distributed deep learning frameworks like Horovod[SDB18] has been created. This software stack is entirely assumes the provided data are tensors and are in numeric format for math-base calculations. In data exploration based research, the data engineering component plays a major role in pre-processing the data to provide numerical features for the data analytics workloads. Currently, the

existing software stack comprises a few options to do data engineering sequentially or parallel. Frameworks like Pandas provide a definition to represent tabular data and pre-process them with basic dataframe operations widely used by the data engineers. And frameworks like Modin and Dask were created by following Pandas to scale Pandas on CPU stack. These frameworks are entirely written in Python and focus on a client-server based distributed model to support data engineering. These frameworks are not entirely focused on classical HPC software stack to provide efficient kernels for distributed computation. But these frameworks are easy to use and provide users the ability to scale existing Pandas workloads on CPUs. Besides CPUs, frameworks like Cudf was created to do data engineering on GPUs. But Cudf is based on HPC kernels specifically written for GPUs and scale on top of dask for distributed computing. Cudf also doesn't posses a BSP mode of execution for data engineering and rely on the classic client-server architecture to scale the dataframe-based workloads. We belive we can design a high performance dataframe which suits the BSP execution which seamlessly integrates with deep learning workloads for distributed computing on HPC hardware. In addition we believe that a BSP model is more effective in scaling large workloads across multiple nodes. Also, having a BSP model which bases on HPC software stack also provides the ability to seamlessly integrate with the existing data analytics workloads specifically designed to run on HPC hardware.

Our research focuses on understanding the importance of high performance data analytics and analyze in depth the integration of high performance data analytics aware data engineering operators to enhance data exploration based data analytics. We observe that the data analytics, data engineering and HPC paradigms are not very well integrated to provide better support for data analytics aware data engineering. In this research, we focus on solving this key problem with a sub set of in

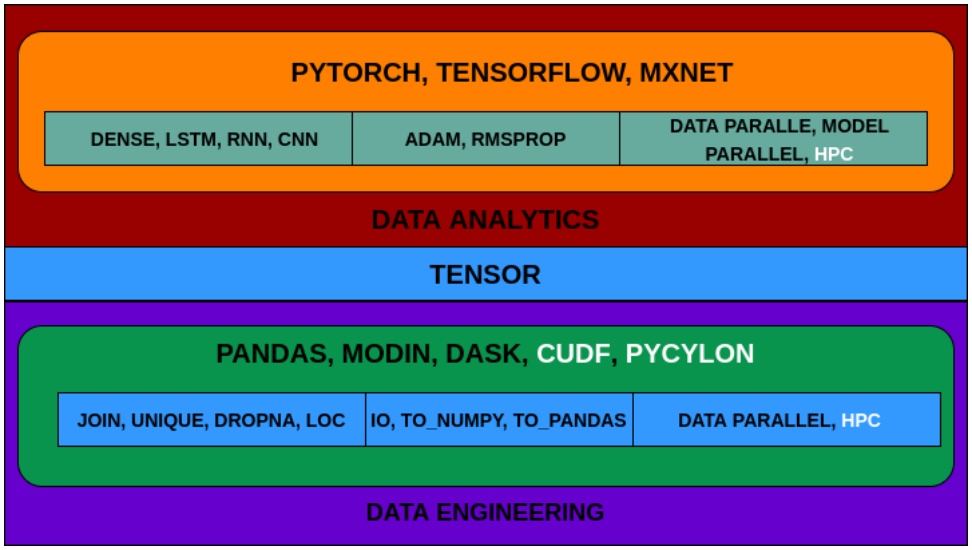

Figure 1.1: Systems overview for data analytics aware data engineering

depth analysis on the importance of high performance data analytics, efficient and effective data engineering and usability for seamless integration with the existing data analytics sub-systems.

Our research goals are focused on evaluating and solving the following key problems.

1.1 Research Goals

- Importance of high performance computing for distributed machine learning with big data
- Importance and necessity of high performance computing for data analytics aware data engineering.
- Identifying limitations of existing data engineering frameworks
- Evaluate the necessity of a distributed memory oriented dataframe for HPC on CPUs.

- Evaluate of high performance data engineering kernels to improve existing dataframe operators.
- Usability of data engineering tools with high performance computing.
- Seamless integration with existing data analytics and data engineering tools
- Efficient implementation of end-to-end scientific data engineering and data analytics workloads

1.2 Research Contributions

- Evaluating the performance of a Support Vector machines with high performance computing approach vs big-data approach
- Introducing a novel distributed memory dataframe for high performance data engineering.
- Evaluating the limitations in the current data engineering solutions with the novel distributed dataframe.
- Designing and building PyCylon, a high performance Python framework for data analytics aware data engineering.
- Integrating with state of the art distributed deep learning frameworks
- Integration with state of the art distributed training libraries for data analytics
- Implementing an end-to-end scientific application for high performance data analytics on the introduced novel dataframe.

CHAPTER 2

INTRODUCTION

Data engineering has become a major component of today's analytical workloads in every major business and scientific application. These analytical workloads depend on the structured data with expected data formats and data types. Such structured data are ingested by vivid analytical platforms to provide intelligence and harness important information. Majority of these applications rely on tabular data and later converted into more complex data structures like graphs depending on application requirements. In an end-to-end analytical workflow, data engineering becomes the first point of entry. Later, the processed data is fed to data analytical systems for training and inference. Since data engineering is a key component, it is important to improve the existing data engineering stack for higher performance and usability.

In the classical data engineering world, big data computing plays an important role. Apache Spark[ZXW+16], Apache Flink[apaa], Apache Hadoop[apab], Apache Beam[Roo20] and Apache Storm[IS15] can be considered as major big data systems. These systems are designed to pre-process the data for most of the industrial applications. The main programming languages used to build these frameworks are Java, Scala and Python. They perform very well in cloud environments. Majority of these systems are designed for high throughput and scalability. One drawback in these systems are the lack of ability to scale well in high performance computing environments which are mainly built on top of high performance compute kernels written in C, C++ and Fortran and communication kernels like MPI[SGO+98], PGAS[ZKD+14] and HPX[KHAL+14]. The importance of running in HPC is driven by the fact that the majority of data analytics workloads are running in HPC-driven environments.

So there exists a tendency for data engineering workloads to be compatible with such requirements.

In the modern data engineering world, a set of data engineering frameworks have gained great popularity due to the core programming language used. Pandas[M+11] can be considered as one of the early systems designed even before some of these big data systems were created. Pandas provides ideal conditions to do data engineering in an effective manner in Python. But this system is not scalable beyond a single core. One major reason for Pandas gaining popularity is the usage of Python in data analytical systems like Scikit-Learn[PVG+11], PyTorch[PGM+19], Tensorflow[ABC+16] and MxNet[CLL+15]. These systems are focused on machine learning and deep learning workloads. Since Pandas was developed entirely on Python, the seamless integration between data engineering and data analytical workloads were made easy. Pandas also support Numpy[num] which is the state of the art numerical data representation format in the scientific computing community. The tensors in machine learning and deep learning frameworks are created based on similar compute capabilities like Numpy and seamlessly integrate with Numpy.

Adopting the Python data engineering best practices, PySpark, PyFlink, PyHadoop, PyStorm and Beam-Python were created to breach the gap between data analytical workloads and to improve usability. Here the data is moved between the JVM-based data engineering backend and Python API exposed to the user. This implementation has many bottlenecks when considering usability and performance. Data movement causes data serialization and deserialization and it takes away a majority of time in large scale applications. Also, the complex task-based systems take away the ability to efficiently prototype a problem unlike it is done in Pandas.

The Python community and research community came up with frameworks like Dask[das] and Modin[mod] to overcome these bottlenecks by introducing scalability

on top of Pandas. But the majority of the compute kernels are written in Python. These frameworks are not scaling well when compared to frameworks like PySpark.

Considering the computer architecture, CPUs are still widely used in heavy data engineering workloads compared to GPUs. One major drawback in GPUs is the lack of ability to do large scale in-memory data engineering problems. Even though frameworks like CuDF[cud] are promising and developed on high performance compute kernels which efficiently run compared to existing Pythonic data engineering frameworks. But the limited memory poses an issue when working with large scale compute jobs which are mainly done in distributed memory.

In addition to data engineering, the programming environment plays a major role in improving the efficiency of the researchers. In classical scientific research, the most important tool is a notebook which contains diagrams, notes and ideas required for conducting an experiment. The Python community also presents a notebook[Per18] environment to visualize the intermediate stages of data engineering and data analytics. These implementations work well with single process computation but doesn't provide a better usability for distributed computing. There are existing commercial products to support this on cloud environments, but there is lacking in usability from the existing open source frameworks like IPyParallel[VOS18] when dealing with the data representation in high performance computing environments. Figure 2.1 shows the higher-level view of data analytics aware data engineering.

We believe that data engineering on CPU-stack can be further enhanced for high performance by retaining the usability provided by existing Pythonic data engineering frameworks. In this research, we introduce PyCylon, a dataframe abstraction written for distributed memory computation in high performance computing environments. PyCylon[APW+20] is the Python data engineering framework written on top of Cylon[WPA+20, PAW+20] data engineering system designed by us. With this

system, our focus is to breach the gap between high performance and high usability in data engineering.

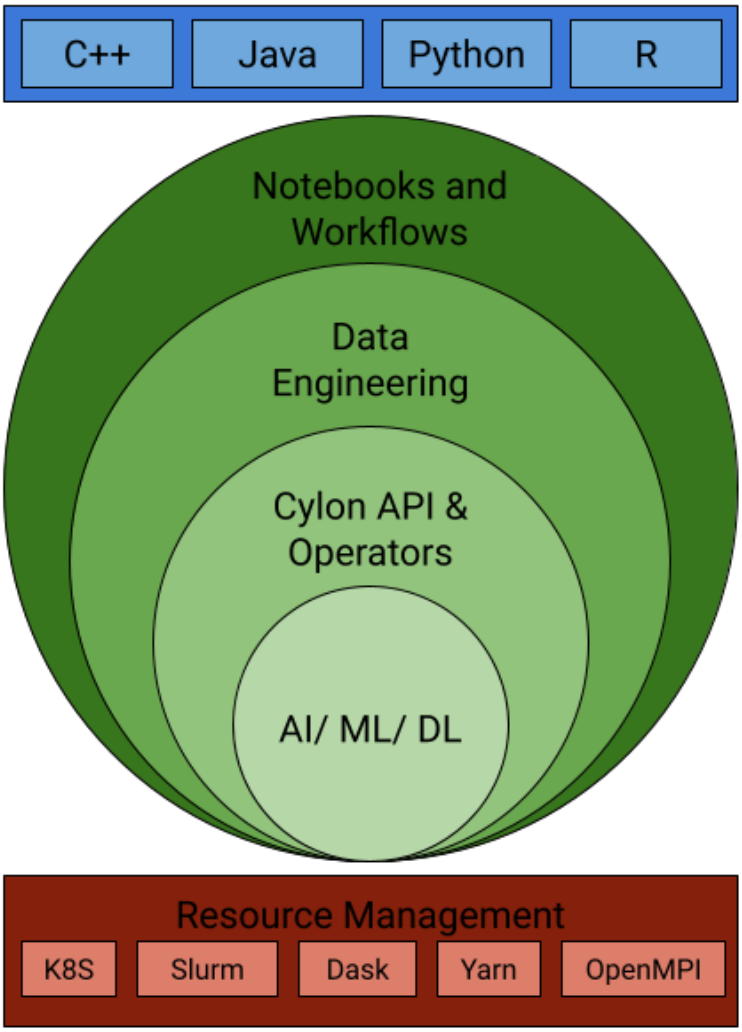

Figure 2.1: Higher Level View of Data Analytics aware Data Engineering

CHAPTER 3

LITERATURE REVIEW

Classical big data systems were a breakthrough in data engineering in the past decade. Major contributions came from open-source software development, enterprise and academic research. Apache Spark [ZXW+16], Apache Hadoop [apab], Apache Beam [Roo20], Apache Flink [apaa] and Apache Storm [IS15] can be recognized as such big data systems capable of data engineering. These systems support both batch and stream data computation on the distributed computing paradigm. Apart from the big data systems, the High-performance computing (HPC) community from academia provided frameworks like MPI[SGO+98], PGAS[ZKD+14] and HPX[KHAL+14] running on supercomputing environments. These frameworks are specialized with high performance compute kernels for math-base computations and distributed memory computations. The HPC systems are mostly favourable for compute-intensive workloads with basic compute and communication kernels. The major differences between big data and HPC systems are the way they are designed and the tasks they are specialized to do. For a better understanding of the general big data use case, it is vital to understand the core values of both systems and design a hybrid system which can do both. Big data systems are easy to use and provide a large variety of compute kernels abstracted by layers of application programming interfaces (APIs) and provide easy access for users to design systems. On the other hand, big data systems are only providing the major compute kernels and communication kernels to build such APIs. But HPC systems are much faster in most cases compared to big data systems. To breach this gap, Twister2[Fox17, twi17, WKG+19, KWG+18] was created from our research to support common requirements in both big data and HPC applications. This system bridged the gap between scientific and industrial research problems conducted on

larger data. Twister2 can run in distributed in-memory, spill to disk and provides all the state of the art communication and compute kernels written in dataflow passion.

Since we developed that system, we have been closely analysing its capabilities and limitations when it comes to computation-intensive applications from modern data-related applications. Data science wrapped in machine learning and deep learning are such sets of applications which require a special set of requirements. Such analytical problems consist of two major aspects. To solve such analytical problems, an efficient system is required to do the computation and an effective system is required to model the problem. The accelerated data processing inherently becomes an HPC problem and existing knowledge can be extended towards designing an efficient system. Considering the effectiveness of the system, programming language, data structures, computation model and communication model can be recognized as the key attributes. Both aspects focus on accurate and efficient model prototyping to solve data analysis problems. With the increasing complexity of analytical problems and nature of data, the data scientists and engineers who work on such problems need efficient and effective systems to make available data analytical model prototypes for production in various scientific domains. The aforementioned requirements can be partially seen in three systems in existing scientific research. Efficiency is provided by HPC systems. Effectiveness is provided by big data systems. But this effect is enhanced to a greater extent with the Python programming layer added on top of existing big data systems. These three characteristics provide the capability to efficiently and effectively prototype a scientific analysis and design the end system for production. We realized this simple pattern which has been adopted by major big data systems like Apache Spark with PySpark[DL17] and Apache Flink with PyFlink[AZR17]. Also, the HPC community extended MPI with mpi4py[Tes16]. These frameworks provided a solution to data engineering problems

to a certain extent.

We observe a set of major drawbacks in the existing systems. The major drawback from big data systems with Python is the massive serialization-deserialization cost when data is moved back and forth from Python to Java. Since core computes kernels in big data systems are written in Java, even though the user program is written in Python, the real workload runs in a JVM-based distributed system. So data has to be continuously serialized and deserialized. In addition to that, the lazy execution model in most of these frameworks takes away the capability of writing eager applications which are easy to debug and prototype with existing scientific workloads written on classical compute kernels on eager execution. In addition to this, the learning curve for maintaining such systems and using such systems is higher compared to modern-day Pythonic data science tools. Pandas[M+11], Modin[mod], Cudf[cud] and Dask[das] can be denoted as some of the prominent tools used in data science. These Python systems are highly effective in designing but suffer from performance issues mainly because of having computed kernels written entirely on Python. Also, these systems do not scale well on distributed computations. But systems like Numpy[num] written with high-performance C++ compute kernels provide better performance over classical Python systems. This reveals a very significant point in designing better systems for efficiency and effectiveness.

One aspect of the motivation behind an efficient and effective data engineering system is the evolving intensive data analytical workloads. Today's data analytical workloads are mainly focused on machine learning and deep learning approaches. These analytical workloads are layers and layers of classical data analytical kernels focused on extracting as much as useful information from the raw data processed by data engineering systems. These analytical systems are built based on two principles. The principles are efficiency and effectiveness. This looks

very similar to the modern-day motivation in data engineering systems. In the early days of machine learning, frameworks like Scikit-learn[PVG+11] and Scikit-Image[VdWSNI+14] were designed entirely using Python. The effectiveness of these systems was very pleasing to the scientists for rapid model prototyping. Evolving towards high performance factor, deep learning frameworks like PyTorch[PGM+19], Tensorflow[ABC+16], MxNet[CLL+15] and Chainer[TOHC15] provided high performance compute kernels written in C/C++ and exposed the kernels via efficient Python bindings. A major part of the computation workload is running on C++, but deep learning system definitions, layer definitions, computation models and distributed computation models are all exposed via Python for effective usage. Similar to big data systems, there are data structures used in deep learning and it is limited to a math-based data structure called tensors. Tensors are the form in which data is being injected into these data analytical systems.

Data analytic systems ingress data from a disk-based or in-memory approach. In the model prototyping stage, this could be mainly done in-memory rather than by disk. Since the scientist is working on evaluating the feature extraction based analytical model convergence, it is vital to keep an efficient data pipeline when processing large data sets. Even for the disk-based approach, it is vital to store them in tensor-compatible data structure rather than other data structures. Focusing on data structures, the big data systems are in favour of structured data in tabular format. In modern-day data engineering, these are also known as DataFrames. Such dataframes can store heterogeneous data in tabular format. In the last stage of data engineering, the data in these dataframes would be mostly numerical for the math-based analysis of data analytical systems. Here the data conversion from data engineering data structures to data analytical data structures is a key. Having an efficient and effective methodology for this conversion is vital in building a seamless

integration between data analysis and data engineering workloads.

Another aspect of data engineering and data analysis is a better medium in sharing the workloads and allowing complex computation models to be visualized. Especially in distributed programming models, it is hard to visualize the intermediate data structures in distributed memory. A medium which is acceptable and widely used by scientists must also be a key to implement such enhancements. Notebooks have been widely used by scientists in the long history of scientific discoveries to take notes and write down experiments and observations in a presentable manner. Extending from this best practice, interactive notebooks like Jupyter notebooks[KRKP+16, GG16] have been widely used by many scientists. To provide enhancements for many programming languages like Python, Java, C++, Scala, Ruby and Julia has been added to such notebooks by scientific research communities. Extending on this many of the industrial research communities have extended this to support various requirements. Netflix has created an opensource version of Scala-driven interactive notebook called Polynote [LDMG20] to match their industrial needs. Apache Zeppelin[CLJ+18] is another such project created opensource to extend the capabilities towards specific goals. For cloud environments and remote compute capability Google Colab [Bis19], Databricks notebooks and Microsoft Azure notebooks [Eta19] have also been created. One main component missing from these tools is to provide distributed computation support on HPC-driven models. IPyParallel [ipy] a parallel compute kernel for IPython[PG07] has been produced to breach this gap. It supports MPI models and task-based models. Dask also provides their notebook [Hay20] extending from the IPyParallel kernels. Still, the main issue is these runtimes are not properly designed to visualize and link with massively parallel computation models. The existing work can be further improved to provide a better user experience and high-performance computing capability to remotely

link to HPC clusters to run scientific workloads, monitor progress and extend to distributed data visualization in an optimized manner.

CHAPTER 4

DISTRIBUTED MACHINE LEARNING

This chapter discusses about the integrating HPC integration for distributed machine learning algorithms. Here we discuss how distributed machine learning algorithms can be efficiently implemented for better scalability and usability with big data analytics. Here, we mainly focus on our early research conducted on building a faster big-data system specifically designed for data engineering for data science. Furthermore, this chapter mainly focuses on a set of machine learning algorithms implemented on our big-data system, Twister2.

Distributed data analytics has been a widely used approach in domain sciences and industrial applications for the better half of the last century. In the very early stages of distributed computing, most of these systems were designed for simulating various domain sciences models. These systems were designed to run on 100s of machines with high-performance capabilities. Later on, this domain was very well established as high-performance computing (HPC). But later on, the trend of analyzing data moved towards the big-data systems with the increasing industrial applications. A set of big-data specialized systems were introduced to meet these specific requirements. Frameworks like Apache Hadoop, Apache Spark, Apache Flink, Apache Storm and Apache Heron are the most prominent frameworks that provided the ability to do computations on batch and streaming data. These systems were specialized to process big data sets with high-level API abstractions following the dataflow model. But we observed that the classical HPC model could be adopted to process big data more efficiently in both batch and streaming settings. Twister2 was designed to provide an efficient communication layer using Twister2:Net to do distributed computing operations efficiently on the HPC hardware. Internally Twister2:Net [KWG+18] uses MPI point-to-point communication to build a com-

munication abstraction having the state of the art collective communication used in HPC. This provides the ability to incorporate the classical HPC communication model into the big data by bridging the dataflow model with the HPC communication. Following this novel model, we developed a set of applications and advanced programming models to fit well with the state of the art dataflow operators in the existing big-data systems. In the following sections, we discuss the distributed SVM, distributed streaming SVM, distributed streaming KMeans and benchmarks carried out on the Twister2 system compared to existing big-data systems.

4.1 Distributed Support Vector Machines for HPC and Big Data Overlap

Support Vector Machines (SVM) is one of the most prominent machine learning algorithms used for classification prior to the deep learning models dominated artificial intelligence applications. With the larger datasets, SVM requires more computing resources to train efficiently. There are multiple implementations which provides distributed computation for SVM. Among these implementations, Apache Spark and MPI-based implementations are very dominant. Since our focus is to improve the performance and retain the big-data attributes in programming, we implemented a distributed version of SVM on Twister2.

4.1.1 Anatomy of the SVM Algorithm

There are a few optimizations algorithms widely used in SVM. Sequential minimal optimization, chunking algorithm and gradient descent are some of these variations. Recently, the usage of gradient descent has been widely considered with the growth of deep learning algorithm. In this implementation we selected a gradient-descent

(GD) based optimization algorithm for the implementation. Algorithm 1 shows the sequential version of the GD-based SVM.

$$S = \{x_i, y_i\}$$
$$\text{where } i = [1, 2, 3, ..., n],\ x_i \in R^d,\ y_i \in [+1, -1] \tag{4.1}$$

$$\alpha \in (0, 1) \tag{4.2}$$

$$g(w; (x, y)) = \max(0, 1 - y\langle w|x\rangle) \tag{4.3}$$

$$J^t = \min_{w \in R^d} \frac{1}{2}\|w\|^2 + C \sum_{x,y \in S} g(w; (x, y)) \tag{4.4}$$

Equations 4.1,4.2,4.3 and 4.4 denote the configurations of the sample space, helper functions for gradient calculation and the loss function.

Algorithm 1 Gradient Descent SVM

```
INPUT : [x, y] ∈ S, w ∈ R^d, t ∈ R^+, b ∈ Z^+
OUTPUT : w ∈ R^d
procedure GRADIENT DESCENT(S, w, t, b)
    for i = 0 to n with step size b do
        if (g(w; (x_i, y_i)) == 0) then
            ∇J^t = w
        else
            ∇J^t = w − C x_i y_i
        w = w − α∇J^t
    return w
```

4.1.2 Parallel Gradient Descent SVM

We use a parallel gradient descent SVM algorithm designed based on the sequential version of the algorithm. After the completion of each epoch, a model synchroniza-

tion is performed by doing an MPI_Allreduce call. Here the model weights across each process is aggregated and averaged over the number of processes involved. Algorithm 2 is the parallel algorithm implemented based on the sequential algorithm in 1. Here K refers to the number of processes, S_i refers to the i^{th} batch and T refers to the total number of epochs.

Algorithm 2 Parallel Gradient Descent SVM

1: $INPUT : [X, Y] \in S, w \in R^d, b \in R^d$
2: $OUTPUT : w \in R^d$
3: **procedure** PARALLEL GRADIENT DESCENT(S, w, b)
4: **Parallel in K Machines** $[S_1, ...S_k] \in S$
5: **for** $t = 0$ to T **do**
6: **procedure** GRADIENT DESCENT(S, w, t, b)
w = MPI_AllReduce(w) / K
return w

4.1.3 Datasets

We use three datasets to determine the performance of the algorithm under a variety in data sparsity, number of features, training data size and testing data size. The table 4.1 refers to the composition of the data selected for the performance benchmarks.

Table 4.1: Datasets

DataSet	**Training Data (80%)**	**Testing Data (80%)**	**Sparsity**	**Features**
Ijcnn1	39992	9998	40.91	22
Webspam	280000	70000	99.9	254
Epsilon	320000	80000	44.9	2000

4.1.4 BLAS Optimizations

In the distributed SVM implementation, we further looked into improving the sequential performance. Here we integrated linear algebra optimizations by using BLAS routines where necessary. In equation 4.5, the *ddot* signature refers to a BLAS operation which performs dot product of two vectors [Donb]. In equations 4.6, 4.7 and 4.8 refers to *daxpy* BLAS operation which performs constant times a vector plus a vector [Dona]. Additionally, the inc_x and inc_y refers to the storage space between the elements in the x and y arguments of the *daxpy* notation where x and y refers to two vectors of similar length.

$$g(w;(x,y)) \Longrightarrow max(0, 1 - y\langle w|x\rangle) \Longrightarrow max(0, 1 - ddot(d, x, inc_x, w, inc_y)); \quad (4.5)$$

$$\langle X_j, y_i\rangle \Longrightarrow daxpy(d, y_i, X_j, inc_x, x_i y_i, inc_y); \quad (4.6)$$

$$w = w - \alpha C X_i y_i \Longrightarrow daxpy(d, \alpha C, x_i y_i, inc_x, w, inc_y) \quad (4.7)$$

$$w = w - \alpha w \Longrightarrow daxpy(d, \alpha, w, inc_x, w, inc_y); \quad (4.8)$$

4.1.5 Performance Benchmarks

We conducted a set of benchmarks by considering the state of the art computing engines specialized for distributed computing. We design two sets of experiments discussing the performance of distributed SVM. The first set of experiments were implemented to compare the performance of implementations on Java and C++

along with BLAS integration. The second set of experiments compare the performance of big-data systems, MPI systems against big-data and MPI hybrid system, Twister2. The experiments were conducted in 16 nodes of Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz and the maximum of processes per node was set to 16.

Figure 4.1 refers to the experiments conducted on the first set of experiments. In these experiments, we evaluated the performance of Java-based and C++ based distributed SVM with BLAS optimizations. From these tests we gathered that the C++ oriented programming provides better performance compared to the JVM-based implementation. The main reasons for the performance boosts are the optimized memory management done in the application development compared to autonomous memory management in JVM-based implementation. In addition to that, the BLAS operations provides slightly better performance when implemented in C++ compared to Java.

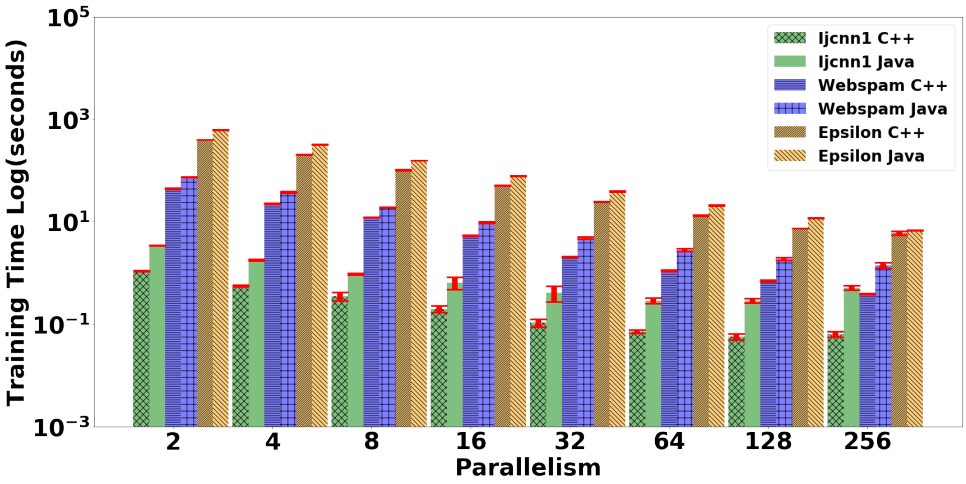

Figure 4.1: SVM Distributed Data Parallel Training with BLAS Optimizations with MPI

Figure 4.2 refers to the experiments conducted on the second set of experiments. These experiments were conducted to evaluate the performance of distributed SVM

algorithm implemented Apache Spark (as a big-data system), MPI (as a HPC system) and Twister2 (a hybrid big-data and HPC system). The objective of these experiments are to show case the importance of integrating machine learning algorithms with a HPC and big-data hybrid system compared to classical big data systems. These results show that, the performance of Twister2 is very much similar compared to the implementation on MPI. Also, Twister2 outperforms Spark-based implementation at scale.

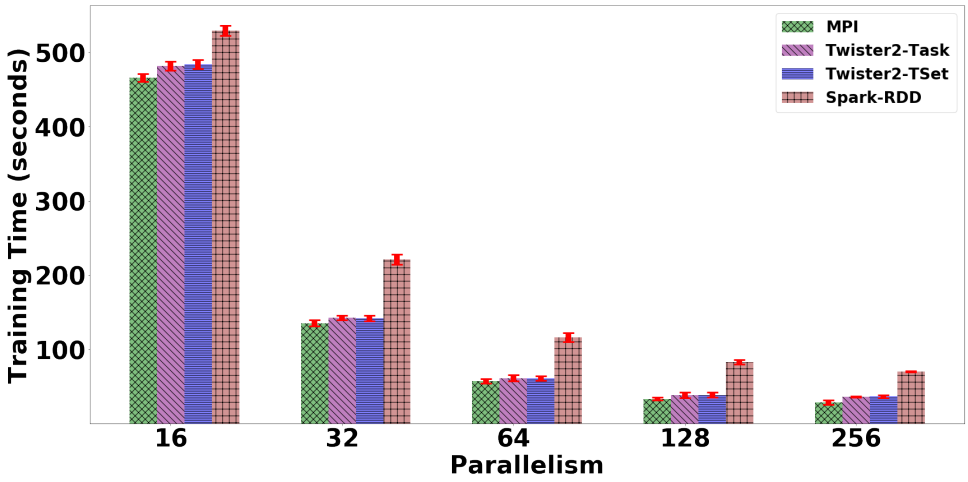

Figure 4.2: SVM Distributed Data Parallel Training with BLAS Optimizations with Big Data HPC Overlap

4.2 Iterative Streaming for Data Analytics

Impact of big-data processing is not only limited to batch data, but also for streaming data. With the expansion of data growth and various IoT applications, it is important to evaluate the application of iterative streaming algorithms for data analytics. Iterative computations are widely done on batch applications for data analytics. When we consider streaming applications, one way is to just compute a given data point once and create a state and use it for the next data point. But when

it comes to the accuracy and various computational requirements, sometimes the streaming data can be converted into mini-batches and computed iteratively. This is the simple idea in iterative streaming processing. A stream can be discretized by partitioning a stream of data into a container called a window. In streaming, a window contains a specified number of elements that is being gathered by means of a windowing schema. Windowing schema can be considered in two ways as far as discretization is considered,

- Tumbling Window (overlapping elements are not included)
- Sliding Window (overlapping elements are included)

In addition to the windowing schema, the window size can be considered either as number of elements in the window or time taken to acquire elements to the window. To provide HPC-aware iterative stream processing, we implemented an iterative streaming component on top of the core streaming engine of Twister2. We developed this component, specifically to focus on data analytics on iterative streaming. The implemented iterative-streaming component is known as Windowing API in the Twister2 system. Figure 4.3 shows how an iterative streaming workflow can be used to design to train machine learning algorithms online.

Initially the data is loaded from a data source which can be a messaging-queue or a stream of data coming from a data storage. The source task does the pre-processing required to formulate the expected features required for the machine learning algorithm. This involves raw data processing to formulate numerical vectors. In the window-compute task, the training mini-batches are generated based on the windowing configurations and iterative computation is done on the formulated mini-batch to create the training model. Here the windowing configuration includes a set of hyperparameters. They are, window length, sliding length, window type

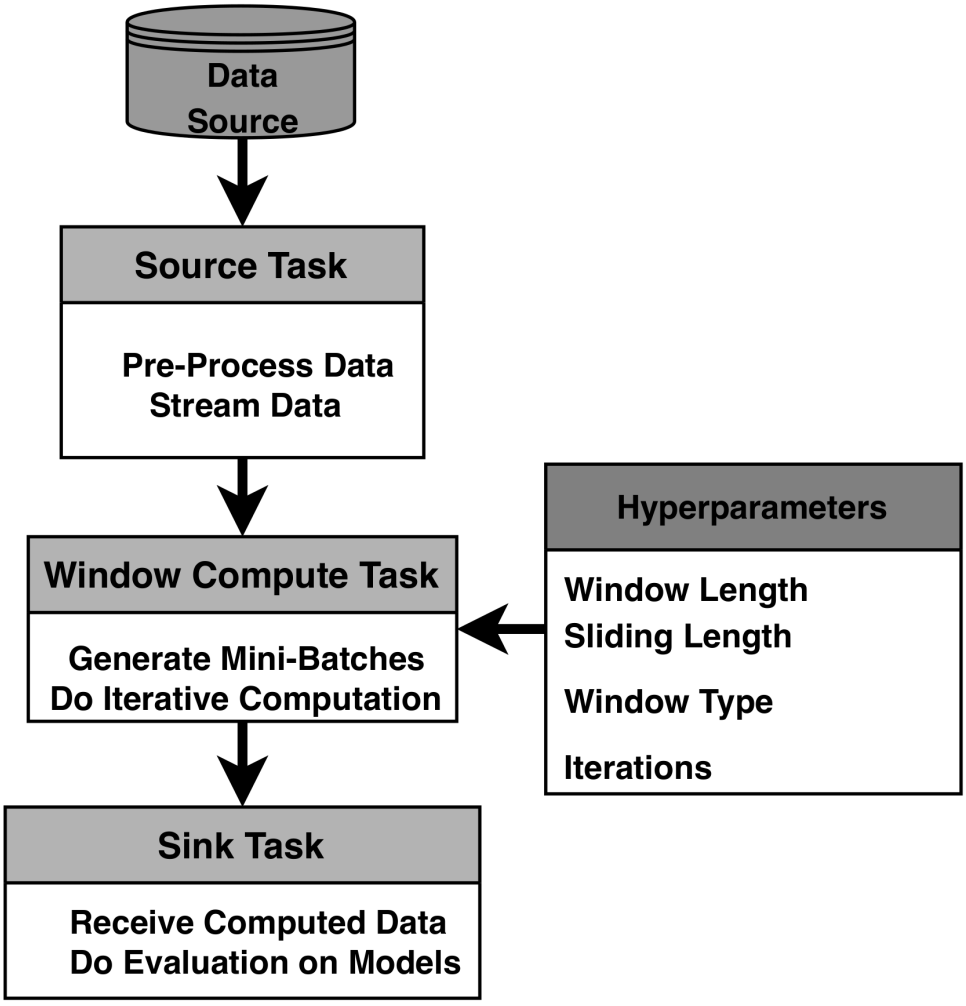

Figure 4.3: Twister2 Iterative Streaming Workflow for a ML Application

and number of iterations per the iterative computation done on a single mini-batch. In the sink-task the computed training model is evaluated on the testing data.

4.2.1 Streaming SVM

Support Vector Machine is one of the most prominent classification algorithm used in the machine learning domain. In an online version of this algorithm, we first discretize a stream of data points into a mini-batch or a window and do an iterative computation on each window. Here a variable number of iterations can be used in tuning the application towards expected accuracy in the training period. The core of the algorithm adopted is a stochastic gradient descent-based model. For each window, the weight vector is updated and synchronized to a global value by doing a model aggregation over the distributed setting. Once a model is globally synchronized over all the processes, it is then tested for accuracy. This implementation follows the principle of a batch model developed to evaluate batch-size based performance on SGD-SVM. We adopted the same approach to calculate the weight vector or gradient in the discretized stream (windowed elements) and globally synchronized the calculated weight vector once the computation per window was completed.

Equations 4.1,4.2,4.3 and 4.4 denote the configurations of the sample space, helper functions for gradient calculation and the loss function.

In algorithm 3, the stochastic gradient descent-based step to update the weights is described as a pseudo-code. This algorithm shows the computation done per data point.

Algorithm 4 shows the complete iterative algorithm with windowing configurations. The l symbol in the algorithm refers to the window length and the s symbol refers to

Algorithm 3 Iterative SGD SVM

INPUT: $[x, y] \in S, w \in R^d, t \in R^+$
OUTPUT: $w \in R^d$
procedure ISGDSVM(S, w, t)
    for $i = 0$ to n **do**
        if $(g(w; (x_i, y_i)) == 0)$ **then**
            $\nabla J^t = w$
        else
            $\nabla J^t = w - Cx_i y_i$
        $w = w - \alpha \nabla J^t$
    return w

Algorithm 4 Iterative Streaming SVM

INPUT: $X_\infty, Y_\infty \in S_\infty, w \in R^d, l \in R^+, s \in R^+, m < K, m \in R^+$
OUTPUT: $w \in R^d$
procedure ISSVM($\bar{S}_i, w, T, l, s$)
    In Parallel K Machines $[\bar{S}_1, ..., \bar{S}_b] \subset S_\infty$
    procedure WINDOW($\bar{S}_m, w, l, s$)
        for $t = 0$ to T **do**
            procedure ISGDSVM($\bar{S}_m, w, t$)
        All_Reduce(w)
    return w

the sliding length. The algorithm encapsulates both tumbling and sliding window-based computations.

4.2.2 Streaming KMeans

KMeans is another popular clustering algorithm in the machine learning domain. We apply an online version of this algorithm in our research. In the streaming setting, we use the stream discretization by means of a window operation. In Algorithm 5 we have implemented a basic version of the online-KMeans algorithm. V refers to the cluster centroids, k refers to the number of centroids, and n refers to the number of total data points observed down the stream. The number of data points observed must be at least equal to the number of cluster centroids. In this algorithm, a single

data point is observed only once and the closest centroid is located by calculating the euclidean distance. The new centroid is calculated as shown in the algorithm. But in the initialization step, the centroids can be either handpicked from the data set or randomly selected. Here we select it as shown in the algorithm. Our objective is to see how each framework works on global model synchronization when working with machine learning models.

In implementing this algorithm we followed the state-of-the-art time notion-based window-less streaming KMeans implemented in Apache Spark. Once the computation related to a window finishes, a global model synchronization is performed. Unlike in a classification algorithm, there is no cross-validation involved during the model generation step.

Algorithm 5 Online KMeans

INPUT:$X = \{x_1, ..., x_m\}, x_i \in \mathbf{R^m}$
2: $V = \{v_1, ..., v_k\} v_i \in \mathbf{R^m}, k \leq n$
OUTPUT: V
procedure STREAMING-KMEANS(X, V)
    procedure WINDOW($\bar{X}, \bar{V}$)
        for x_j in $\bar{X}$ **do**
            if $j \leq k$ **then**
                $v_i = x_j$
                $k_i = 1$
                $i = i + 1$
            else
                $v_i = argmin_i ||x_j - v_i||$
                $v_i = v_i + \frac{1}{n_i + 1}[x_j - v_i]$
                $n_i = n_i + 1$
        All_Reduce(V)
    return V

4.2.3 Model Synchronization

In the distributed setting, generating a synchronized model is vital. In implementing the online versions of the machine learning algorithms, we adopted the strategies specific for each framework. In Apache Flink, the reduce function is used for synchronizing the models. This is the only possible way to get an approximation to the all-reduce model. Apache Flink doesn't support an all-reduce like communication for synchronizing models globally. In Apache Spark, reduce function and RDD broadcast is used to synchronize the model. In Apache Storm, all-grouping is used to generate a synchronized model. Twister2-HPC model uses MPI-AllReduce collective communication to synchronize the models. Twister2-Dataflow model uses a variation of all-reduce communication with a tree-like communication model. The model synchronization is thus carried out in Twister2.

4.2.4 Performance Evaluation

For the experiments, we use a distributed cluster with 8 physical nodes. We schedule 16 tasks per each node to run the experiments. Each node consists of Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz with 250GB of RAM capacity. For running an experiment for a finite period, a stream of 49,000 records for training and a stream of 90,000 records for testing is used. For the experiments, we only use a finite stream to evaluate training accuracy and performance. The implementations used for the performance evaluation are Apache Storm 1.2.8, Apache Flink 1.9.0 and Twister 0.3.0.

Streaming SVM

For streaming SVM model, we use a dataset with two classes with 22 elements per data point. For the experiments, we used an iterative computation on windowed elements. This operation is supported by Apache Flink, Apache Storm and Twister2. We tried this model using Apache Spark streaming engine. With the provided APIs and system constraints, we were able to design an approximate model to that of design with the aforementioned frameworks. The main constraint is that it only provides windowing considering the notion of time. This makes it hard to do a stress test on the stream engine. Because, by the notion of time, the minimum number of elements that can be set per batch is in millisecond level. Furthermore, it doesn't support iterative streaming models. This feature is not directly supported with DStream in Apache Spark streaming engine. With the approximate model, the accuracy obtained was comparatively very low concerning the other frameworks. A workaround is to use structured streaming in Apache Spark. This implementation works on the SQL engine of Spark, and it only considers the notion of time. We didn't implement that model in this research as it is a very different implementation concerning the other implementations. In the conclusion section, this will be explained in detail.

Figure 4.4 shows the experiment results for tumbling window is shown. From these results, it is clear that the Twister2 models outperform both Apache Storm and Apache Flink implementations. Figure 4.5 shows the sliding window related experiments. Similar to tumbling windowing, with sliding windows, Twister2 implementations outperform Apache Flink and Apache Storm implementations. Twister2 possesses a faster stream processing capability through a strong MPI-based backend. This provides a scalable solution for an iterative stream processing on a window. With Apache Flink, the main bottleneck is the reduce task doing the model

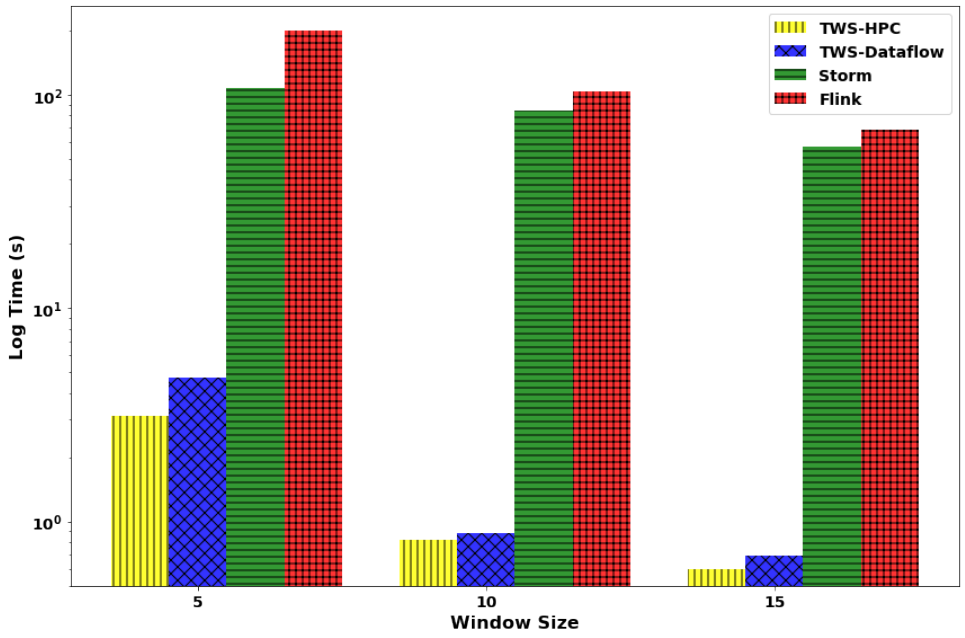

Figure 4.4: Streaming SVM with Linear Kernel-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

synchronization. In Twister2 and Apache Storm, the all-reduce and all-grouping mechanisms involve in providing all-toall model synchronization capability. But in Apache Flink, this process becomes all-to-one and makes a bottleneck in processing the data. In this case, both Twister2 and Apache Storm outperforms Apache Flink.

From all implementations in Apache Flink, Apache Storm and Twister2, 90.49% of test accuracy was obtained after a finite length of the stream was processed. With Apache Spark implementation, we were able to get an average accuracy of 40%-50% with the same number of iterations. We didn't include the graphs here, because the number of iterations required to get the same accuracy is much higher. The main issue for this is Spark streaming API is not designed with iteration compatibility. Also, it doesn't provide a window function to capture the elements belonging to a window. This functionality is available in Apache Storm, Apache Flink

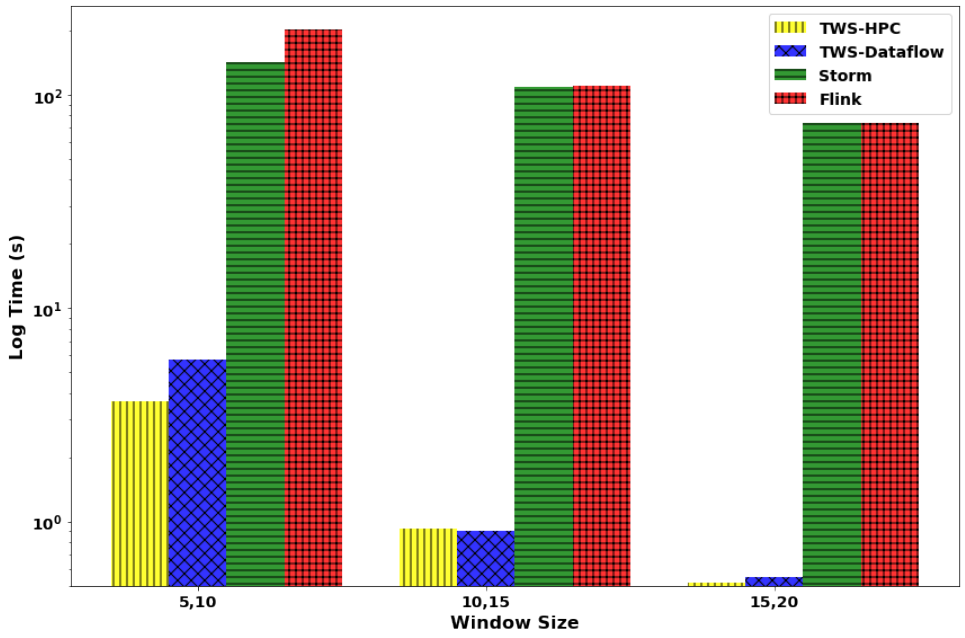

Figure 4.5: Streaming SVM with Linear Kernel-based experiments for sliding window is recorded for HPC model and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x-axis in the right figure is labeled with the pair of (window length, sliding length).

and Twister2. Apache Spark only provides basic element wise operators like map, flatmap, etc. If this was attempted with a forEachRDD function, the user has no capability to synchronize the model as it is a sink function. In addition Spark only provides a windowing functionality with the notion of a time and it has no support for windowing based on the count of elements.

Streaming KMeans

For the streaming KMeans model, the dataset we used contains 23 elements per a data point. Here a non-iterative computation is done. Apache Flink, Apache Storm and Twister2 support the windowing functions to implement an algorithm like this. With Apache Spark streaming, a non-iterative application can be developed but the count-based notion is not available in the API. In this research, we have only conducted windowed streaming with the notion of the number of elements per window. In achieving the current goal we have used the streaming systems which provide this functionality.

Figure 4.6 shows the tumbling window-based experiments carried out on streaming KMeans model. And in figure 4.7 shows the sliding window-based experiments carried out on streaming KMeans model. Similar to streaming SVM results, Twister2 models outperform both Apache Spark and Apache Flink. Twister2 model synchronization with an all-reduce mechanism provides faster execution than that of regular all-to-all communication in Apache Storm. In Apache Flink, there is no all-to-all communication, the model synchronization happens in an all-to-one setting. This is the same bottleneck as observed in streaming SVM application. But Apache Flink outperforms Apache Storm. This model is a non-iterative model and the pressure exerted on communication is lesser. This leads to quite faster data progress from the windowing task to the reduce task.

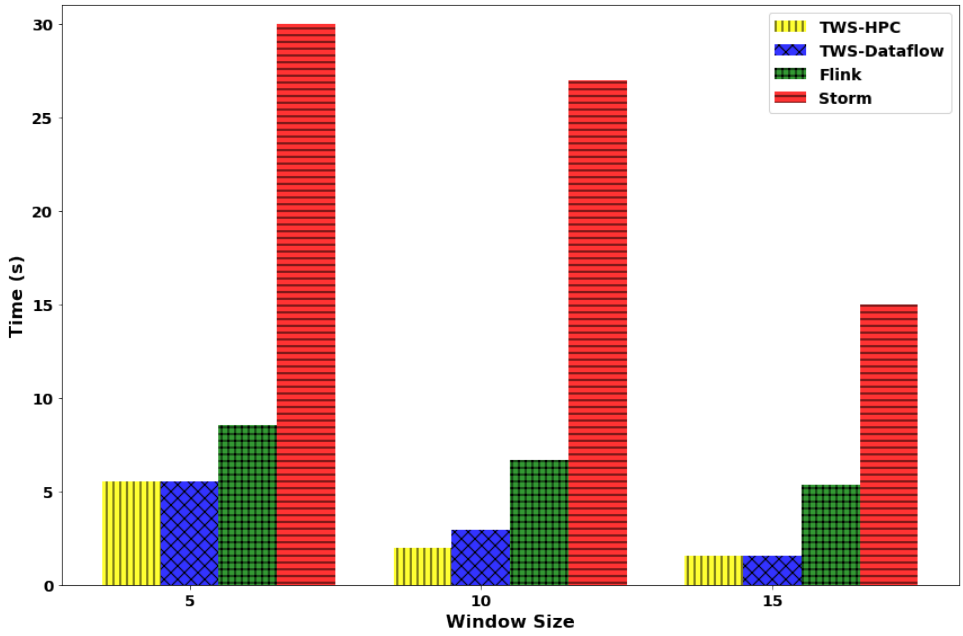

Figure 4.6: Streaming KMeans Results for 1000 cluster-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

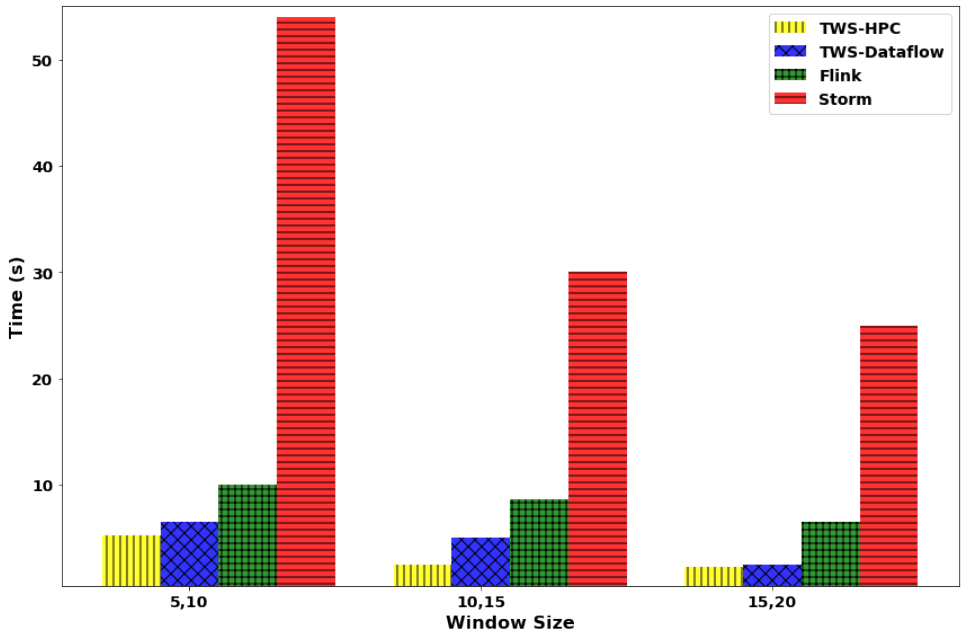

Figure 4.7: Streaming KMeans for 1000 cluster-based experiments for sliding window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x axis in the right figure is labeled with the pair of (window length,sliding length).

CHAPTER 5

HIGH PERFORMANCE DATA ANALYTICS AWARE DATA ENGINEERING

The importance of data analytics and the necessity of integrating high performance computing resources is a must for many scientific problems. From our in-depth research and discussion in chapter 4, it is evident that the data analytics workloads can be efficiently executed in large scale to train vivid scientific models effectively and efficiently. Another important aspect that we didn't analyze in depth is the data engineering operations that is being widely standardized and used in parallel to the evolved data analytics workloads. Even though the big data systems played a major role in the data analytics in the better half of the last decade, more systems took over data analytics by specializing into sub domains by not just providing distributed communication, but providing application development capability by writing a very smaller number of lines of code. In addition to that, these data analytics frameworks were built in such a way that the data scientists can easily write programs and analyze the data very efficiently. Frameworks like PyTorch [PGM+19] and Tensorflow [ABC+16] are such dominant frameworks specialized for data analytics. And also the user base, scientific applications and research efforts exponential grew into using these data analytics frameworks rather than using classical big data systems to implement problems. With the emergence of these frameworks, it is clear that the best is to provide better support for the data engineering portion of data analytics is very important than building data analytics components with classical big data systems. Taking all these facts into considering, we dive deep into investigate how we can provide better and faster tools to do data science by retaining the best practices of data engineering.

Even though data science is the key to understand patterns and behaviors in data oriented problems, the key component to make data available for such analaysis is data engineering. With the dawn of data science applications, most of these workloads moved to Python to make these tools available with much easier programming abstractions for efficient and effective application development. Besides, most of the data science platforms are developed in Python programming language to allow data scientists to develop applications efficiently. When it comes to data engineering, there are Python APIs provided on top of the JVM-based big-data systems to provide this usability. But using the existing systems is a very time consuming exercise for data scientists to pre-process the data. In addition to that, these APIs are not seamlessly integrated to support HPC-oriented data science systems to provide better performance and usability. From our current research efforts and existing literature, we believe that the data engineering stack can be further reinforced for high performance computing by building a set of high performance data engineering kernels and making them available via a simple Python interface to develop data science applications efficiently.

In the modern data engineering domain, most of the Python-based data engineering systems are developed by considering a data abstraction called dataframe. Dataframe is nothing but a tabular data representation for heterogeneous data. The raw data coming from various data sources contain data with multiple data types and are mostly in tabular shape. Having a tabular data representation is key to support a wide range of data engineering applications. Pandas[M+11] dataframe can be considered as the state of the art dataframe representation in the Python-oriented data engineering. Pandas only support data processing in a single process at the moment. There have been many efforts from the Python community to provide distributed computing for dataframe. This abstraction has been adopted by

many distributed data engineering frameworks like Modin[mod] (structured Pandas for parallel computing), Dask[das] (distributed Pandas), Cudf[cud] (dataframe for GPUs). Our data engineering interface mimics the state of the art dataframe and extends towards a HPC-compatible computation to provide distributed operators and pleasingly parallel operators.

We also observe that the existing data engineering workloads on CPUs can be further enhanced. The reasoning behind our observation is based on the way systems are implemented and the way they are integrated with the existing data analytics workloads. Since the theme of our contribution is data analytics aware data engineering, we do a dive from the top to bottom, by starting to analyze the modern day data analytics workloads and how this can be reinforced by building a data engineering system to support data analytics.

5.1 Methodology

In supporting data analytic applications rapidly evolving on Deep Learning, the vital task would be to identify where data engineering is efficiently applied. The direct relationship between data engineering and data analytics is the data and the pipeline to move data from the data engineering engine (DE) to data analytic engine (DA). For a better data engineering design, it is vital to understand the requirements of the DA frameworks and backtrack to design efficient solutions. The approach for understanding requirements is drawn from existing applications. Also, we expand the requirements for designing the DE solutions by understanding the future demands based on the DA application evolution.

The computations in DA engines are associated with numerical data structures like tensors. A seamless connection between DE engine and DA engine can be enabled by transforming DE data structures into tensors. For this task, efficient data

transformation and loading are key components. Figure 5.1 shows the data movement from a data source towards a data analytic workload. The DA applications in the future will be dominated by datasets since multi-modal training and multi-task training have become the newest data analytic models. To provide an efficient data pipeline from DE to DA, it is vital to understand the key components of DE and DA data loading components.

Data engineering can be discussed under three criteria namely data extraction, transformation and loading. Data extraction phase is related to reading data efficiently from data sources like distributed file systems, distributed messaging queues and local file systems. In processing big datasets, efficient data reading and data movement in distributed computing environments is a key attribute. Data in this phase is heterogeneous where both numerical and non-numerical data is in a structured or unstructured format. Structured data are with a schema and this is the most common data type (CSV or spreadsheets). Unstructured data formats are mainly involved in domain science research where a format like HDF5 is widely used. The key challenge is to read data efficiently and load into an efficient in-memory format where data can be efficiently transformed.

In the scope of this research, the main focus is to accelerate the CPU workloads on data pre-processing and data movement. The pre-processing can be defined into two main categories. The first category is the raw data processing to extract features. The second category is numerical data augmentations done to re-shape and transform the data into the data analytic models. Here some of the data transformation kernels are known to run much faster in GPUs. But the limitation in the in-memory computation becomes a bottleneck in pre-processing larger datasets. The main objective of this research is to provide an efficient and effective data engineering system on CPU-based DataFrame abstractions with seamless integration to Python.

On the developed system, state of the art scientific applications and workloads are benchmarked. In the initial phase of the benchmarks, we micro-benchmark system-level performance compared to the existing state of the art systems. For evaluating an end-to-end workload, we benchmark state of the art scientific workloads written on the proposed system.

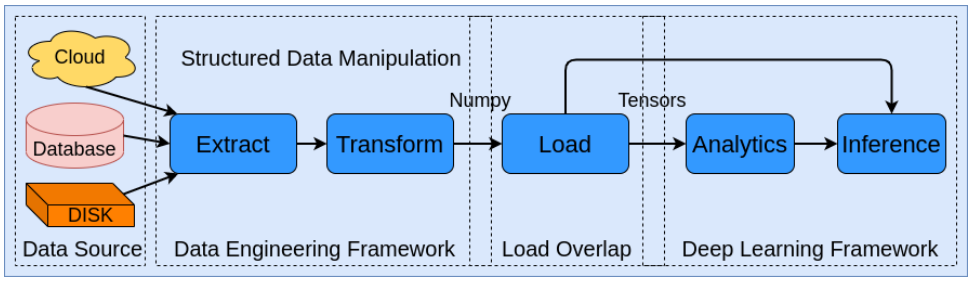

Figure 5.1: Data analytics aware data engineering workload

5.2 System Architecture

Figure 5.2 refers to the high-level architecture of the proposed system. The lower layer comprises the high-performance communication layer written with MPI and high performance compute kernels written in C++. To facilitate the support for existing frameworks and to improve usability, a layer of language bindings is implemented to expose to multiple languages. But the main focus of our effort is to enrich the Python data engineering stack to seamlessly integrate with data analytics frameworks which are mainly written with a Python user interface. On top of the language bindings, the usability APIs and sub-algorithms are developed. DataFrame API is the key component providing high-performance data engineering. The DataFlow API is the gateway towards external systems like machine learning and deep learning systems allowing data movement from a data engineering workload to a data

analytics workload. The highest level of abstraction is focused towards an annotated Python API which allows users to write distributed or sequential code without paying attention to internal details of writing a parallel code. The distributed computation is abstracted away from the user in terms of writing data engineering kernels. Since, the programming model is on the classical bulk-synchronous-parallel (BSP) model, in some advance applications user needs to handle parallelism aware local computations by dealing with the rank or process id.

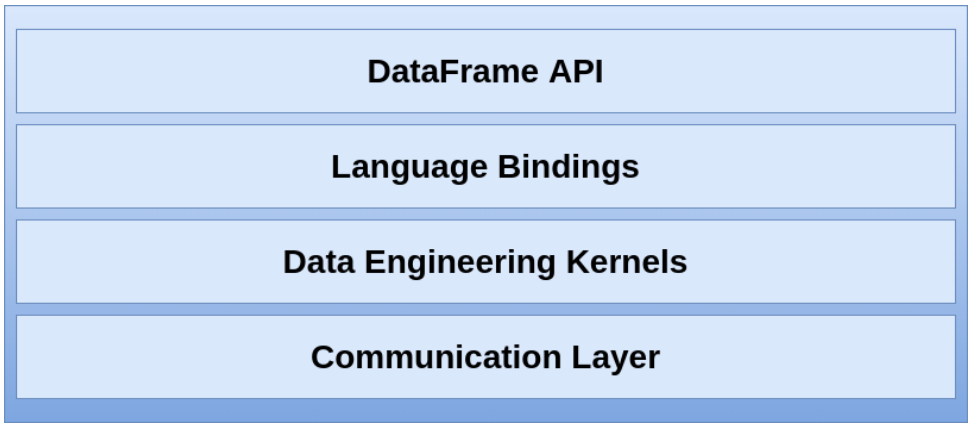

Figure 5.2: System Architecture

These data engineering operators are built into Cylon framework. Following are the key components contributed from our research to the Cylon framework.

- Designing and implementing high performance data engineering kernels
- Designing and implementing high performance language bindings for Python
- Designing and implementing a dataflow abstraction for data analytics aware data engineering
- Designing and implementing a fast and scalable DataFrame abstraction on distributed memory.

- Specification for data engineering operators on distributed memory computation with BSP awareness.
- Seamless integration with Python-based data structures for data analytics and data engineering
- Seamless integration with data analytics frameworks for distributed data parallel computations

5.3 Communication Kernels

The data engineering kernels require the ability to compute in parallel. The communication layer provides an All-To-All abstraction to move the data around machines based on the data distribution. All-To-All implementation is written with MPI point-to-point isend and irecv calls. Compared to classically data-parallel jobs which require mere data parallelism, data engineering workloads are more focused on processing a data sample with a given attribute in data. Relational algebra operators like join, union and intersect are operating on a given sub-attribute on the data set. Join requires a specific column/s to do the join and the data type of that column and the value of the data is required to do the required computation to join two tables. In the distributed setting, when comparing such values, hashing becomes a very prominent technique which allows moving the values with the same hash to a single machine to do the relational algebra operation locally and provide the expected result similar to the sequential algorithm. This is the main difference between a regular data-parallel workload and a data engineering workload.

5.4 Data Engineering Kernels

Data engineering kernels are the core compute kernels required to process raw data. For better performance most of our compute kernels are written in C++ or we refer to vectorized C++ and Python implementations to enhance the performance. These are the main cateogries of data engineering kernels supported in the system.

Kernel	Operation
Relational Algebra Kernel	Join, Union, Intersect, Difference and Project kernels
Indexing Kernels	Hash, Linear and Range indexing kernels
Search Kernels	Hash, Linear and Range indexing based searching
Filter Kernels	Filter values by conditions
Duplicate Handling Kernels	Locate duplicate values and filtering
Null Handling Kernels	Locate null values and replace or remove
Linear algebra operators	Basic math operations

Table 5.1: Core data engineering kernel classification

These data engineering kernels can be divided into two groups based on the scalability.

- Local Operators
- Pleasingly Parallel Operators
- Distributed Operators (distributed memory operators)

Note that there are no explicit implementations of pleasingly parallel operators, the local operators can be executed in a pleasingly parallel way depending on the parallelism. The distributed operators are only designed to run on $parallelism > 1$ and it falls back to local computation when used in $parallelism = 1$ setting.

5.4.1 Relational Algebra Kernel

Considering the dataframe, the main data relational algebra kernel used is the join operation. In the dataframe domain, the join operation requires a set of parameters

Operator	Local	Distributed	Pleasingly Parallel
Relational Algebra	Yes	Yes	Yes
Indexing	Yes	Not Implemented	Yes
Search	Yes	N/A	Yes
Filter	Yes	N/A	Yes
Duplicate Handling	Yes	Yes	Yes
Null Handling	Yes	N/A	Yes
Linear Algebra	Yes	Not Implemented	Yes

from the user. The join column/s, join type, join prefixes (optional) and join algorithm (optional). The join prefixes are to provide readability when visualizing the join outputs. The join type falls under four categories.

- Inner Join : Includes records that have matching values in both tables.
- Left (outer) Join : Includes all records from the left table and just the matching records from the right table.
- Right (outer) Join : Includes all records from the right table and just the matching records from the left table.
- Full Outer Join : Includes all records, but combines the left and right records when there is a match.

The join algorithm is an additional feature supported from our implementation to provide user the ability to use the most suitable algorithm depending on required performance and scalability. We support two join algorithms, namely hash join and sort join. In the sort join, we sort both relations by join column and do a merging operation by scanning from top to bottom in both relations.

In the local hash join, hashing is done on the join column of one relation (preferably the smallest relation) by keeping them in a hashmap and scans through the other relation join-column and computes the hash to build the resultant table by comparing hashes. In the distributed setting, before performing the join, we do a

hash-based data shuffling. The hashes for join columns are computed and data is being re-shuffled in such a way that hashes with equal values come to a designated process. After this communication process is completed, a local join is computed in each process.

5.4.2 Indexing Kernel

Indexing kernels support fast data querying. We have implemented 3 indexing types to support vivid use cases. The current implementation only supports single column indexing. The supported indexing kernels are;

- Vector Indexing : A column of a table is used as an index and vector search operations are applied
- Hash Indexing : A column of a table is used to create a multi-map of key value and row indices
- Range Indexing : A virtual column is created with start and end index

In the vector indexing implementation, a column in selected and dropped off the table depending on a user argument. The idea is that particular column can be a data column and index or it can just be a column just to search values corresponding to a particular query criterion. Currently we only support searches based on equality. In the hash indexing implementation, the selected column is being hashed such a way that, hashes with same value maps a set of rows. Compared to vector indexing implementation, the hash indexing implementation takes more time due to the hash collisions and multi-map population time. Range indexing is basically a virtual indexing interface to provide compatibility for a non-indexed table. In a non-indexed table, we create a virtual column with 0 to num_rows with $int64$ data type. But

this virtual column doesn't exist in actual memory until user wants to get index values. It only records the starting index and end index and generates the index as per user's requirement in data visualization.

We have designed a generic indexing interface with endpoints to implement advanced search operations on retrieving data from a table. The generic indexing interface includes the following end points to retrieve data. For the distributed mode, we provide a unique index for each process, at the moment the searching happens by considering a local index in each process. For distributed operations, at the moment user require to re-implement the indexes after the distributed computation. When using vector indices the time taken to generate the indices is negligible. We will discuss more on the performance in the benchmark section 6.1.

- Retrieve by range of keys (a start key and a end key)
- Retrieve by a list of keys (single key retrieval implicitly includes)

5.4.3 Search Kernel

For fast data retrieval index based searches provide the edge over a linear search across all the records. The search kernels are implemented to support the higher level search capability. When retrieving data, users can provide a start index and end index or a vector of indices and specify which columns need to be included in that query. The search kernels exposes these querying capability. Basically there are 6 types of sub-queries involved with indexing-based search involving the following two categories.

- Search by value range (start value and end value): this retrieves a set of records start from the start value and ends with end value. Here the start and end values must be unique values in the given index.

- Search by a vector of values: A search is done upon each value in the vector with the index values.

When retrieving search table, the user can specify a range of column/s or a list of columns as well. These combinations all together gives 6 types of sub-querying. For both distributed and sequential context the same search kernel is used. In the BSP setting, when a search occurs, each operator will be executing the search operator for the search parameter given in each process. For this operator a distributed search function is not applicable.

5.4.4 Filtering Kernel

Filtering provides the ability to retrieve a table from an existing table by using a mask. A mask is a set of boolean values. In tabular format this is can be a table with multiple columns or a single column table. To generate this filter table, there are multiple ways. A boolean-valued table can be generated by comparing table or subset of table values to a given value/s. Filtering kernel basically provides the ability to compare the values considering the basic comparator operators allowed by any programming language. The supported comparator operators are;

- Greater than
- Greater than or equal
- Lesser than
- Lesser than or equal
- Equal
- Not Equal

In implementing these comparator operators, we have used two types of implementations to support vector operations. Since we use Arrow-tabular format to represent the data internally, we use Arrow compute operations to provide the vectorized filters rather than implementing them from the scratch. Also we have integrated numpy kernels internally to support optimized vector operations for comparators. We have exposed the choice of selecting compute engine as a parameter when defining the context of the application runtime. These are discussed in detail in the dataframe section 5.6.

In terms of parallelism, the filter function is considered as a pleasingly parallel operator. Each process will provide the filtering operator along with the filter value. The filter value can be different for each process or it can be same as far as a BSP programme is considered.

5.4.5 Duplicate Handling Kernel

Duplicate records can be widely located in a raw dataset. The objective of the duplicate handling is depending on a keep policy, where keeping the first value found as duplicate into records or the last value found into records. This implementation consists of a hashset where we iterate through each row and create a hash. When new rows are inserted to the hashset, a row comparator is used to evaluate the hash to identify whether the value is already present or not. This provides the ability to differentiate between a unique record and a duplicate record. In the default setting, all the columns are considered when considering a duplicate record, but we have also provided the option to include a subset of columns to decide the uniqueness of a record. In the distributed setting, prior to executing this algorithm, a data shuffling operation is done based on the selected columns considering the hash value of a row.

5.4.6 Null Handling Kernel

In raw data processing removing null values and replacing null values with meaning full values is a useful function. And also depending on data representation, null value can be represented as a string or can be particular phrase. In the data loading stage, we provide options to denote such references 5.9. Null handling can be categorized into three main operations;

- Check null values (boolean response)
- Drop null values
- Fill null values

When checking null values, since we are integrated with Apache Arrow format, currently we use internal kernels of Apache Arrow. The same process is followed for filling null values with a user specified option. When dropping null values, we implemented a based on dropping null values based on the existence of null values row-wise and column-wise based on existence of any null value or all null values. For column-wise operations we check for nulls and do a algebraic sum of boolean values to determine whether all or any values are null and corresponding row indices are dropped. But in the column scenario, we designed a couple of heuristics. The reason for designing the following heuristic is that, since we depend on columnar data, we cannot physically do row-base computations directly. For row-wise null check implementation, a column-wise null check is done and the obtained boolean array is then casted to $int32$ array and addition of all columns is taken. Then we follow the following heuristics in deciding all or any null values are present.

- Heuristic 1 : If the resultant sum-array contains an element with value equals to 0. It implies that all elements in that row are not None.

- Heuristic 2 : If the resultant sum-array contains an element with value equals to the number of columns. It implies that all elements in that row are None.

Selection criterion is decided on how the dropping is done, when 'any' is selected, the addition value in corresponding row is greater than 0 that row will be dropped. For 'all' criteria that value has to be equal to the number of columns.

5.4.7 Linear Algebra Kernel

Linear algebra kernels are another important set of kernels required when numerical computations are involved in numerically typed data. Currently we support basic math operators like addition, subtraction, multiplication and division on column based and table based (if all columns are numerically typed). To support efficient vector operations for linear algebra, we have implemented the support by using Numpy and Arrow compute kernels. Currently, we don't support an internal version of linear algebra kernels but rely on highly optimized kernels written in Numpy and Arrow. With arrow we use the C++ level compute kernels exposed via Arrow compute APIs. Matrix level operations are not yet supported in the current implementation.

5.5 PyCylon

Language bindings are a key component of our system. Since we focus on providing highest usability and seamless integration to the data science eco-system, ability to write programmes in Python is essential. We introduce the framework PyCylon the Python API for data engineering workloads as one of the major contributions from our research.

Majority of our data engineering kernels are written in C++ and kernel level APIs require efficient language bindings when it comes to linking up with extensively used languages like Java or Python. The main focus on our research is to make the tools available in Python. To enable C++ kernels to Python, the most efficient and developer friendly mode is to use Cython as the interface between C++ and Python. Cython has been widely used in scientific computing libraries like Numpy, Scipy, Pandas and many other research projects to write efficient Python bindings.

Figure 5.3 depicts the high level API abstraction in our data engineering framework. The lower most layers consist of the core kernels for data engineering and communication. On top of these core kernels, we have the C++ Cylon API. This API contains all the endpoints for writing communication modules, data engineering operators, data loading operators and other util operators. We expose thee C++ Cylon API with the Cylon Cython API to make an efficient Python interface. This layer is also seamlessly integrated with PyArrow Cython API and Numpy Cython API. Also, the current Cython layer can also seamlessly integrate with other libraries providing Cython interfaces. On top of the core Cython API, we develop the PyCylon API. The PyCylon API is purely Python and calls to Cython interface when computations needed to be done on data engineering operators.

5.5.1 Cython for Python Bindings

Cython is a special language created to design Python programmes for high performance computing. The advantage of using Cython provides the ability closely work with C/C++ data structures using the internal Cython APIs. When building HPC systems for Python, Cython provides a wide variety of APIs to integrate C/C++ kernels to Python such that from Python C++ functions can be called very

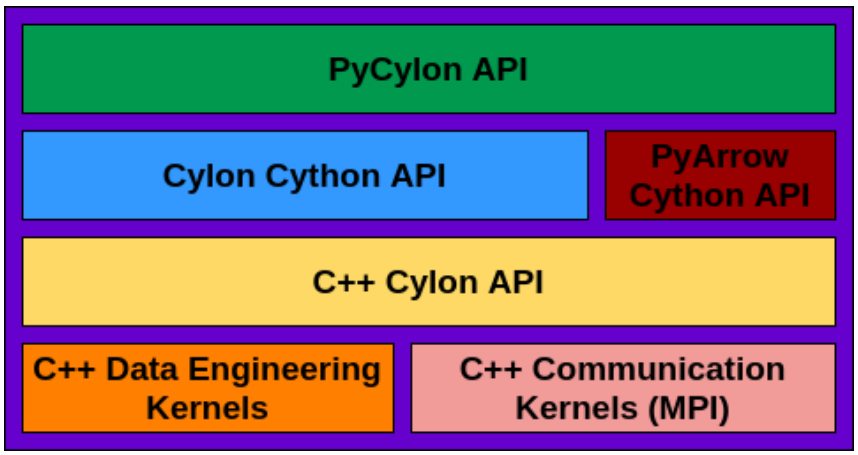

Figure 5.3: High Level API Abstraction

efficiently.

In terms of data copying across languages, if the data is created on Python data structures (entirely Python) and if the data are primitive types it will be copied when calling functions or creating C++ objects via corresponding Python objects. But in our design, since we are using Apache Arrow format to load the data, there is no data copy even when we do computations on a data created on PyArrow or LibArrow using Cylon. This provides an advantage since we don't need to serialize or deserialize data.

Referring to the function calls made from the Python interface, the actual computation happens only in C++ based memory allocation formulated when loading data via Arrow. Here Python actually doesn't do any memory allocation, but calls Cython bindings to do data engineering operators internally on that designated memory location (or Arrow Table) and the output is presented to the user. Here the data copying refers to the pure data loading to the memory for computations. Figure 5.4 shows how a particular Python interface has been integrated via language bindings to the core compute kernels.

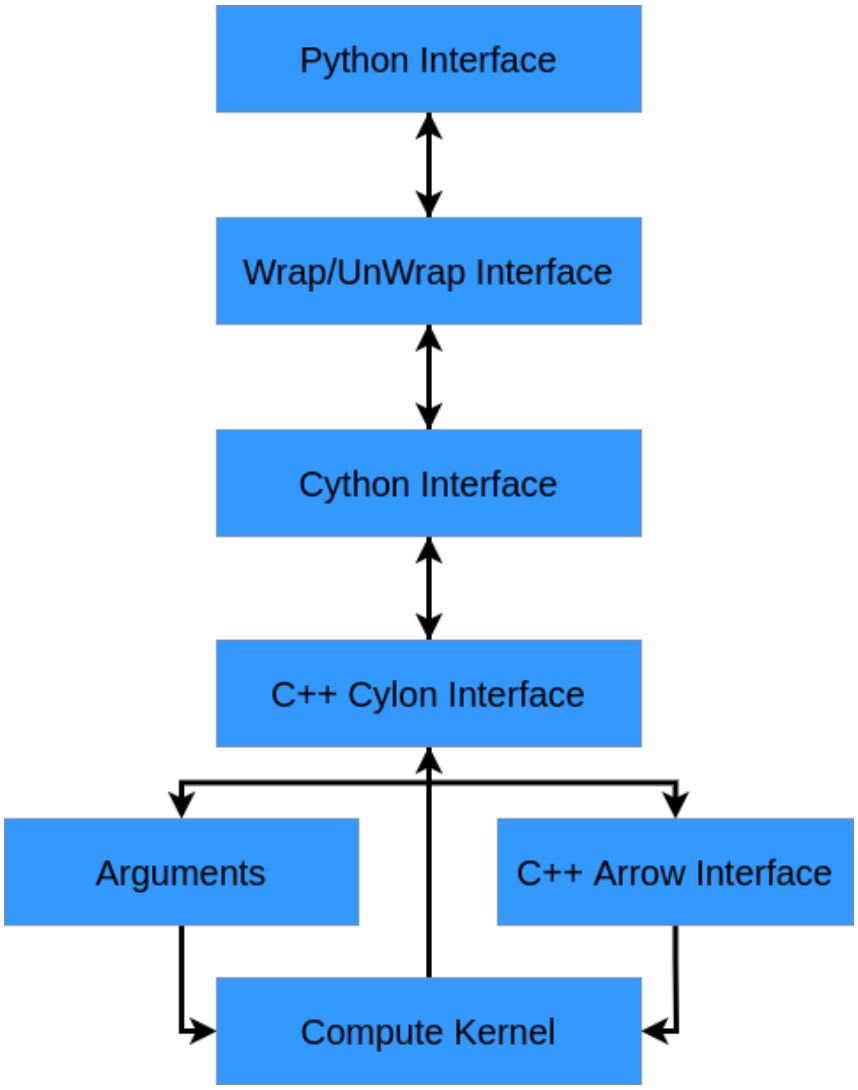

Figure 5.4: Cython Interfacing with Computing

Here any compute operator or compute interface that needs to be executed using C++ core kernels require the flow shown in the figure 5.4. The Such interfaces can be unwrapped to access the Cython interface object which is generally a shared pointer in C++. The Cython interface provides the expected shared pointer reference to the C++ Cylon interface. Within our C++ Cylon interface lies the C++ Arrow interfaces (Arrow Table or Arrow array) which will be used in the compute kernel. Once the expected computation is done on the shared pointer, the resultant shared pointer is again wrapped as a Python object using the wrapping interface. The unwrap interface is nothing but casts the Python objects into Cython object and extract the underlying shared pointer in the C++ implementation. The wrap interface use the underlying shared pointer in C++ to create the Cython object and forms the Python object. Here there is no data copy since we do this by pointing to the shared pointer created when initial data was created in memory. The Apache Arrow interfacing provides the ability to even extend this to multiple languages without doing any serialization or deserialization when moved across vivid language layers.

5.5.2 Cython API

With the discussion in section 5.5.1, the objective of this layer is to provide the efficient execution of C++ kernels and provide outputs to Python interface without copying data but use the underlying memory allocated from C++ kernels. For extended use and advanced usages the Cython API can be used to build third-party libraries and add-ons. Currently our Cython API includes major functions available in the C++ kernels. The exposed Cython APIs are as follows;

- Table API : Includes data engineering high level operators

- Context API: Interface to determine distributed runtime information
- Configuration API: Sub modules related to network information and other configurations
- IO API : Input and output modules related to data (read, write, convert)
- Compute API : A high level wrapper for sub-set of data engineering kernels built on top of Apache Arrow Compute API.

We have used the Cython API to extend the functionality to a higher level Python API 5.5.3. Similary, when building third-party libraries or writing additional functionalities, kernels from existing libraries can be extended via the Cython layer.

5.5.3 Python API

Python API is the highest level of API abstraction in the framework. The Python API relies on the immediate underlying layer, Cython API. The Python API consists of the wrapped interfaces containing Cython interfaces. This layer is designed to provide more usability in programming environments and abstract away Cython syntaxes and utils from the users. Python API mainly contains the DataFrame API 5.6 and other util APIs supporting data engineering.

5.6 Dataframe API

Dataframe is the highest level of API abstraction in our system. This dataframe API is designed such that it mimics the functionality of the state of the art Pandas Dataframe representation. The major differences compared to Pandas dataframe is the functions with distributed computing interfaces and a few additional interfaces supporting to determine the distributed context of an application. The dataframe

API is built on top of the Table API exposed in the Cython API. The Table API contains all the core data engineering operators abstracted to be used as a simple Python class. But the dataframe API abstracts away the distributed compute function calls and other internal details and provide a streamline function definitions. The supported dataframe operations are grouped into sequential (also function as pleasingly parallel operators) and distributed operators as shown in the table 5.2 and 5.3 respectively. Here we mainly support the widely used data engineering operators and our motivation is to improve and add more operators based on the scientific applications developed as a research outcome of this research.

In the dataframe representation, most of the operators are pleasingly parallel operators because the nature of the computation mimics a sequential computation that can be executed across all the processes. The distributed operators that can be supported for dataframes are dataframe initialization, joins, groupby, join and duplicate handling operators. In our research we mainly focus our attention towards initializations, joins and duplicate handling.

5.7 Interoperability Among Python Data Structures

When it comes to application development, one of the most important feature is the ability to seamlessly integrate with other existing data structures which are widely used in the interdisciplinary domains. The data structures mainly represent the data in a useful abstraction with vivid compute functions useful in manipulating the data. As far as data engineering is considered, the widely used data formats are CSV, Parquet, HDFS and other binary formats like HDF5. When it comes to tabular data representation, the widely used data format is CSV or Parquet. Parquet is an efficient columnar representation for data storage on disk. In the data

Pandas Operator	Description
index	Indexing for faster search
columns	List Columns
shape	Show dataframe shape
empty	Create an empty dataframe
isin	Check whether a value/s exists in the dataframe
where	Check the index of a value/s located in the dataframe
add	Addition of a scalar to a dataframe object (numerical)
sub	Subtraction of a scalar to a dataframe object (numerical)
mul	Multiplication of a scalar to a dataframe object (numerical)
div	Division of a scalar to a dataframe object (numerical)
lt	Comparator for lesser than
gt	Comparator for greater than
le	Comparator for lesser than or equal
ge	Comparator for greater than or equal
ne	Comparator for not equal
eq	Comparator for equal
add_prefix	Add a prefix for dataframe columns
add_suffix	Add a suffix for dataframe columns
drop	Drop a column from a dataframe
rename	Rename a dataframe
take	Obtain a sub-sample of a dataframe by indices
dropna	Drop not applicable values
fillna	Fill not applicable values with a user given value
isna	Check for not applicable values
isnull	Check for null values
notna	Inverse check for not applicable values
notnull	Inverse check for not null values
set_index	Index a table by a given column
reset_index	Reset index of a table
loc	Locate sub-sample of dataframe by value
iloc	Locate sub-sample of dataframe by position

Table 5.2: DataFrame Pleasingly Parallel Operators

Operator	Description
DataFrame	Create a dataframe in distributed memory
join	Join two dataframes by a column or index
merge	Join two dataframes by the index column
drop_duplicate	Drop duplicate values by config

Table 5.3: DataFrame Distributed Operators

loading (discussed in section 5.9), we provide support to load data into a PyCylon dataframe. To understand how we provide the data inter-operability, it is vital to showcase how we have interfaced the tabular data representation.

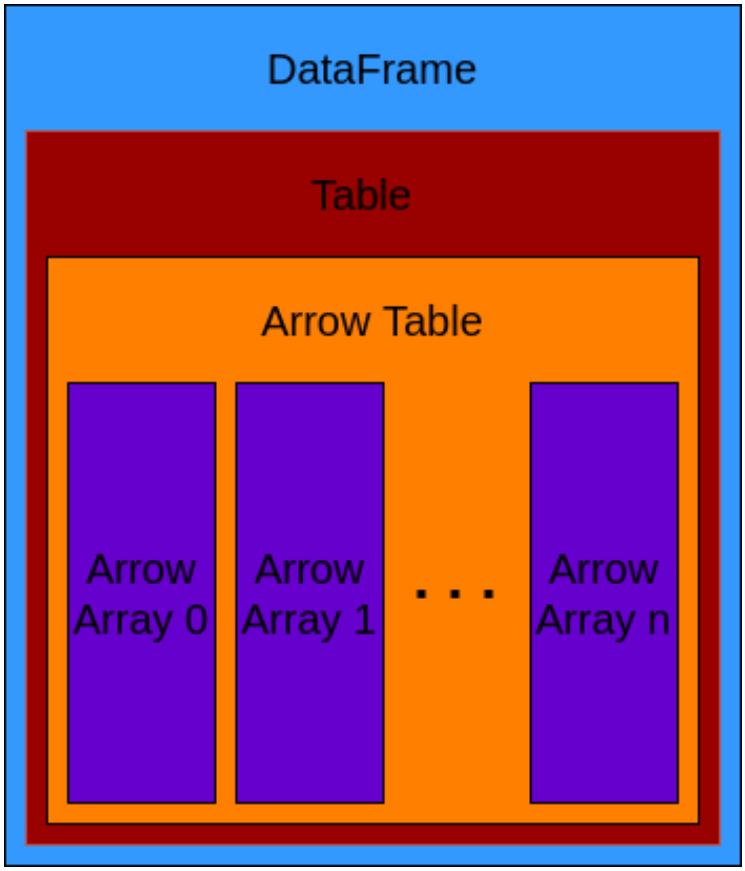

Figure 5.5: Data Structure Hierarchy

From figure 5.5, the underlying data structures in the PyCylon framework are shown. Here the dataframe is the aforementioned higher level API for data engineering operations. Table refers to the C++/Cython table abstraction integrated with data engineering kernels. Internally, we represent the data using Arrow tables. Arrow format with the columnar representation provides a naturally efficient mechanism to read the data from the disk by considering the memory layout.

For data engineering operators, we access the underlying Arrow array data structures and pass to compute functions. One important feature with Arrow table is that the Arrow table is an immutable table, when we do a particular data engineering operation we have to create a new table with updated data eventhough it is viewed as an inplace operation in the higher level. Here we replace the underlying shared pointer of the table from the original table with the table created with the computation results.

Since our dataframe is internally integrated with Apache Arrow Table data structure, our Dataframe posses the ability to seamlessly integrated with other data engineering data structures like Pandas dataframe, Numpy arrays and PyArrow Tables. Figure 5.6 shows the data inter-operability in across data engineering operators.

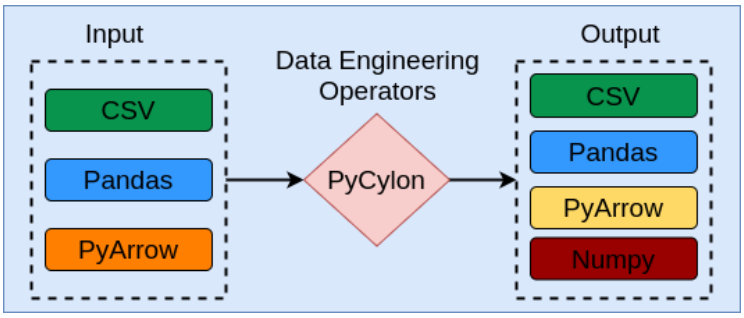

Figure 5.6: PyCylon Data Inter-operability

5.8 In-Memory Conversions

Facilitating high-performance data movement and zero-copy among systems is a key component of a data analytics aware data engineering workloads. Figure 5.7 shows the memory copy overheads and the ability to move data back-and-forth in vivid data structures used in data engineering and data analytics. Since PyCylon dataframe is internally using Arrow data structure, it facilitates the seamless integration to

Pandas and Numpy. Here, the Arrow Table cannot be directly converted to a multi-dimensional Numpy array. But iterating through each column, a multi-dimensional array can be created. Arrow internally supports zero-copy when data representation is numeric, with no null values and chunk-array (a list of arrays is represented as a chunk-array in Arrow format) with $chunk\_size = 1$.

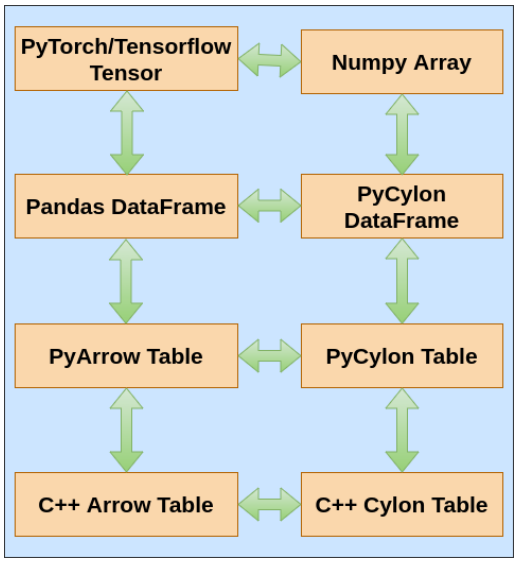

Figure 5.7: In-memory data conversion

5.9 Data Loaders

Data loading is the entry point for any data engineering application. In our data engineering framework, we currently support the loading CSV and Parquet formatted files via Arrow readers, Pandas readers and by in-memory data structures like Pandas, Numpy and PyArrow. Data loading is not only important for data engineering but also for data analytics workloads. Since our dataframe abstraction has a seamless integration to Numpy arrays, providing input to deep learning frameworks is a possibility with the support of efficient data conversion from Numpy array to

Tensor in PyTorch and Tensorflow. This allows users to rely on existing data loaders from the widely used deep learning frameworks.

5.10 Productivity and Usability

Productivity and usability plays a major role in prototyping applications and designing efficient production frameworks. In recent years, Python has played a major role to enable such objectives. Specifically for data engineering workloads, Pandas operator specification has become the state of the art operator specification. In our research contribution, our objective is to minimiz the programming overheads by providing a similar API to Pandas. This allows users to migrate existing data engineering solutions to PyCylon with minimum code changes. When migrating an existing workload, users will be only modifying an import statement and add a distributed context to enable efficient data engineering.

The usage of data engineering operators with the sequential relational algebra operators can be seen in the algorithm 6.

Algorithm 6 Using Sequential Relational Algebra Operators

```
1: from pycylon import DataFrame
2: import random
3: df1 = DataFrame([random.sample(range(10, 100), 50),
4: random.sample(range(10, 100), 50)])
5: df2 = DataFrame([random.sample(range(10, 100), 50),
6: random.sample(range(10, 100), 50)])
7: df2.set_index([0], inplace=True)
8: df3 = df1.join(other=df2, on=[0])
9: print(df3)
```

The difference between a sequential and distributed operation is the function call and the way the context is initialized. This is clear from the algorithm in 7. Here we provide a similar API to Pandas and optimize the API to provide minimum code

change to do code migrations efficiently. The availability of local and distributed operators allow a user to develop an application with dynamic capabilities to solve complex data engineering problems more easily. Another advantage of having this mode of API support allows user to use pleasingly parallel or local operators along with the distributed operators with minimum code changes.

Algorithm 7 Using Distributed Relational Algebra Operators

```
1: from pycylon import DataFrame, CylonEnv
2: from pycylon.net import MPIConfig
3: import random
4: env = CylonEnv(config=MPIConfig())
5: df1 = DataFrame([random.sample(range(10*env.rank, 15*(env.rank+1)), 5),
6: random.sample(range(10*env.rank, 15*(env.rank+1)), 5)])
7: df2 = DataFrame([random.sample(range(10*env.rank, 15*(env.rank+1)), 5),
8: random.sample(range(10*env.rank, 15*(env.rank+1)), 5)])
9: df2.set_index([0], inplace=True)
10: print("Distributed Join")
11: df3 = df1.join(other=df2, on=[0], env=env)
12: print(df3)
13: env.finalize()
```

Algorithm 8 shows way data filtering operators are designed in PyCylon DataFrame API similar to Pandas. In addition to the usage of context, the rest of the data engineering code is designed to match with existing Pandas definition for data filtering. Similar to a regular Pandas program, our APIs provide the ability to retrieve a subset of data by providing a slice of row indices like $1 : 3$, providing a column name/s and retrieving those values, filtering out a sub set of data by using a comparator operation and filtering a dataframe using another dataframe with boolean values on the entire table.

Algorithm 9 shows way data location operators are used with indexing. This also has a similar syntax compared to Pandas dataframe. Here the user can either set an index manually by providing index values or use an existing column to set

Algorithm 8 Using Data Filtering Operators

```
1: from pycylon import DataFrame
2: data = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
3: df = DataFrame(data)
4: df1 = df[1:3]
5: df2 = df['col-1']
6: df3 = df[['col-1', 'col-2']]
7: df4 = df > 3
8: df5 = df[df4]
9: df8 = df['col-1'] > 2
```

the index. Then *loc* operation can be executed by providing slice of starting and end indices with expected columns.

Algorithm 9 Indexing Operator

```
1: from pycylon import DataFrame
2: data = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
3: df: DataFrame = DataFrame(data)
4: df.set_index(['a', 'b', 'c', 'd'])
5: df1 = df.loc[2:3, 'col-2']
6: df2 = df.loc[2:3, 'col-3':'col-4']
```

Algorithm 10 shows the way math operations can be used on the dataframe with scalar values and dataframes with similar shapes. Currently PyCylon support the operations on a sequential mode and embarassingly parallel mode. We have not yet implemented distributed operations for a adding a table from one process to a table in another process.

Algorithm 10 Math Operators

```
1: from pycylon import DataFrame
2: data = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
3: df: DataFrame = DataFrame(data)
4: df1 = df + 1
5: df2 = df1 * 10
6: print(df2)
7: df3 = df1 + df2
8: print(df3)
```

Algorithm 11 shows duplicate handling can be done using the dataframe. This semantic similar to Pandas routine.

Algorithm 11 Drop Duplicates Operators

```
1: from pycylon import DataFrame
2: import random
3: df1 = DataFrame([random.sample(range(10, 100), 50),
4: random.sample(range(10, 100), 50)])
5: df3 = df1.drop_duplicates()
6: print("Local Unique")
7: print(df3)
```

Algorithm 12 shows duplicate handling can be done using the dataframe as a distributed operation. This semantic similar to Pandas routine except for the designation of the PyCylon context to identify it as a distributed operation.

Algorithm 12 Distributed Drop Duplicates Operators

```
1: from pycylon import DataFrame, CylonEnv
2: from pycylon.net import MPIConfig
3: import random
4: env = CylonEnv(config=MPIConfig())
5: df1 = DataFrame([random.sample(range(10*env.rank, 15*(env.rank+1)), 5),
6: random.sample(range(10*env.rank, 15*(env.rank+1)), 5)])
7: print("Distributed Unique", env.rank)
8: df3 = df1.drop_duplicates(env=env)
9: print(df3)
10: env.finalize()
```

CHAPTER 6

PERFORMANCE AND BENCHMARKS

To evaluate our system under stress tests, we did a couple of benchmarks by grouping our compute operations as parallel and sequential execution mode. For sequential operations we benchmarked our system with Pandas since it is considered as the state of the art dataframe implementation on CPUs. For distributed operations, we compared the performance with Dask distributed Dataframe which is an implementation to scale Pandas on CPUs. Also, we conducted another set of benchmarks to compare our sytems performs compare to NVIDIA GPU devices with Rapids Cudf. Apart from this we also benchmarked the performance of our languge bindings compared to the C++ core to identify the overheads from each language binding.

6.1 Indexing and Searching

For indexing experiments we conducted three sets of experiments to show case the indexing performance, search performance and overall performance for a search followed by indexing. Generally an indexing operation is done once or a few times, but search operations dominate if there are more queries associated with a problem. Here we conducted the indexing experiment by using a dataset ranging from 100 million records to 1 billion records. These records were generated such that 10% of the keys are unique in the indexing column. This data distribution provides us the ability to test the ability to query large number of data pointing to the same key. In the search operation we search for all unique keys in the data distribution. Figure 6.1 shows the performance results gathered from this experiment. Here we can see that, PyCylon vector indexing mode outperforms Pandas indexing mode while PyCylon hash indexing mode is much slower compared to Pandas. The main reason

for this slowness is that, internally when we build the hash index, we populate a multi-map (a C++ standard multimap) which takes more time to build the map under large number of hash collisions are present.

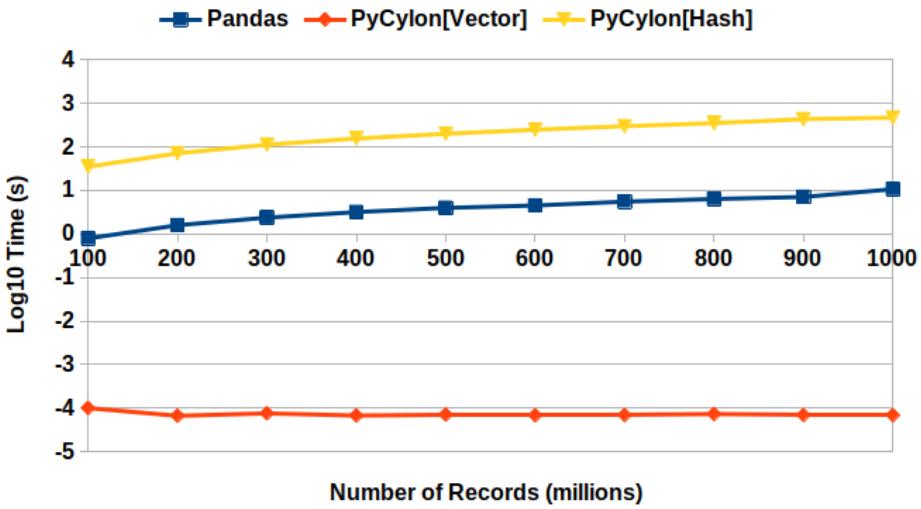

Figure 6.1: Indexing Operation Performance

When considering the search experiments, we used the aforementioned data distribution with the same index. Figure 6.2 shows the results obtained for loc operation or search operation. The results show that both PyCylon search implementations performs the search much faster when compared to the Pandas implementation.

Since we observed that, the indexing performance for hash-based indexing is much slower in PyCylon compared to default indexing mode in Pandas, we conducted another experiement by taking the indexing factor into consideration and calculated the total time taken to do a search followed by an indexing operation. In a real world scenario the number of searches per indexing operation is a factor which is greater than or equal to 1. Figure 6.3 shows the experiments conducted with the aforementioned scenario. Eventhough the hash indexing performance is

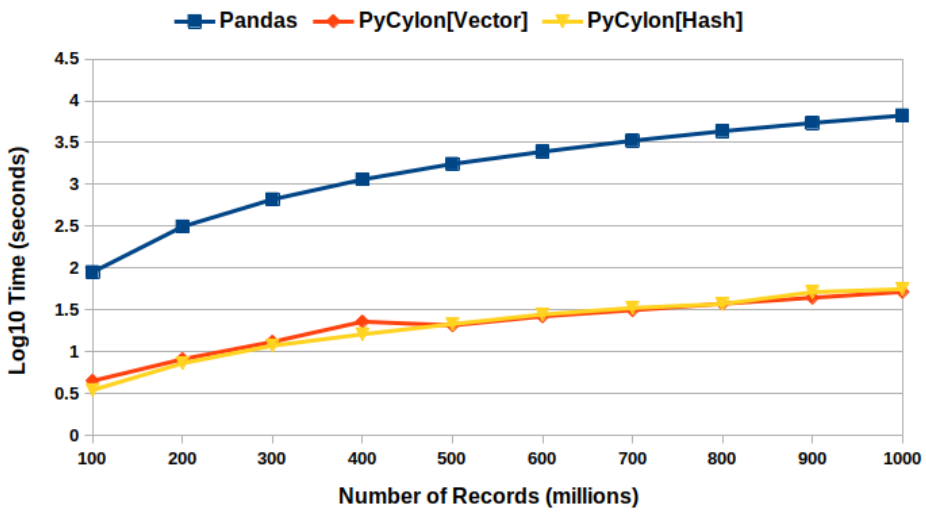

Figure 6.2: Search By Value Operation Performance

slow in PyCylon, it is clear that a search operation followed by indexing is still faster compared to the time taken to Pandas to do the same operation.

6.2 Duplicate Handling

Duplicate handling operator refers to the drop duplicate function. Here we benchmarked records generated such that 10% of the data is unique. Figure 6.4 shows the results for the experiments conducted compared to Pandas. This shows that with the increasing data size, PyCylon is performing faster compared to Pandas.

6.3 Comparator Operations

In comparator operations we refer to the operators which determine whether a particular value is greater, lesser, greater than or equal, lesser than or equal, not equal and equal operators. With respect to a dataframe these operators mean the same

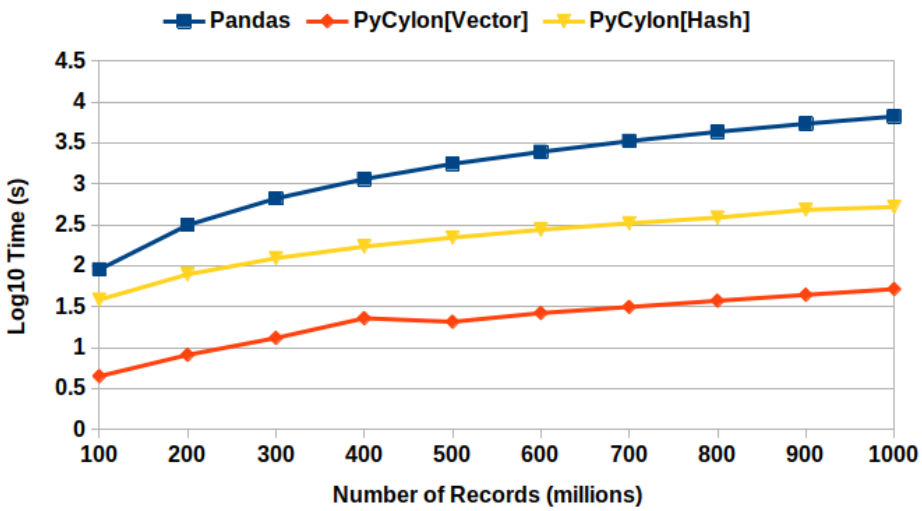

Figure 6.3: Indexing and Search By Value Operation Performance

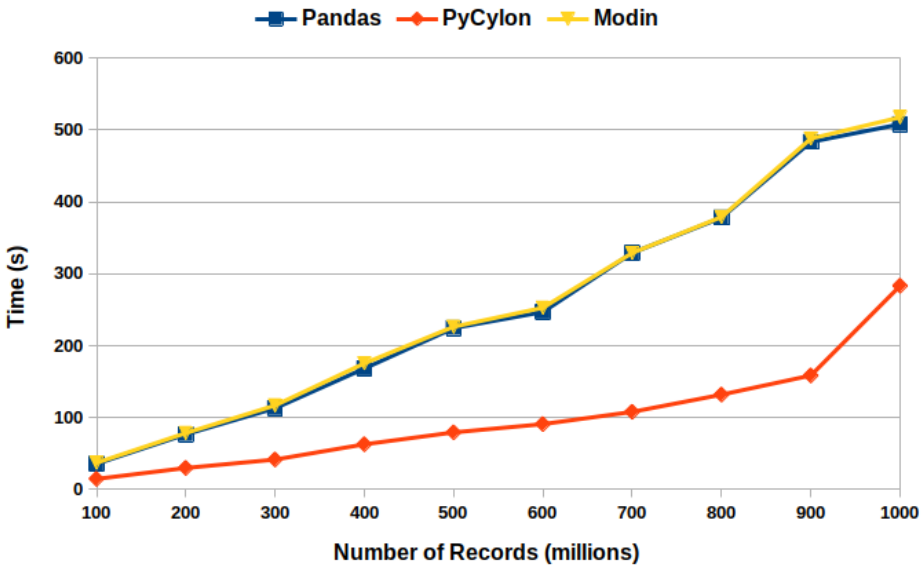

Figure 6.4: Duplicate Handling Operation Performance

idea. Figure 6.5 shows the results from the conducted experiments. Here we observe that the PyCylon comparator operator is slower in performance compared to that of Pandas. We did a micro-benchmark and observed that the internal performance for computation of the filter is as fast as Pandas. But the boolean valued output created for each column must be transformed to a PyCylon table. In Arrow, the overhead observed in creating a boolean type table is much higher than creating a table with numerical values. We believe that this will be further enhanced in a future Arrow release. In addition to this, we also support these computation do be done either using Arrow or Numpy compute kernels.

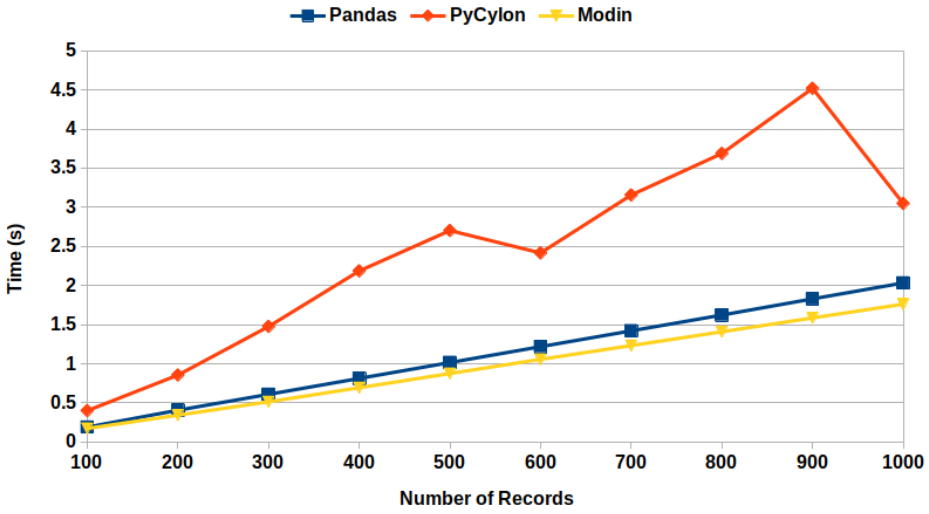

Figure 6.5: Comparator Operation Performance

6.4 Math Operations

Math operations we benchmarked are basic operations like scalar addition, subtraction, multiplication and division. Here the benchmark refers to a scalar addition to the dataframe created with the variable number of records. Here we observe that the

performance of Pandas and PyCylon is very close. Here we internally use Numpy and Arrow interface as the compute engines to do these linear algebra operations. Both compute engines performs similarly. In the benchmark the compute engine can be specified by passing a configuration parameter to the context.

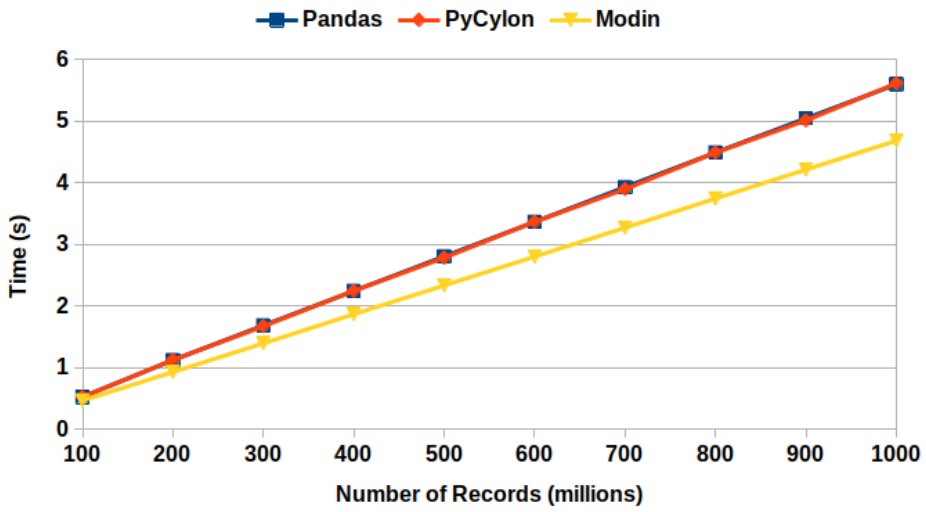

Figure 6.6: Math Operation Performance

6.5 Null Handling

Null handling operator benchmark is designed with 90% of null values in the records and dropna operator is executed to drop null values on the dataframe for all columns. Here the performance benchmark is shown in figure 6.7. These results show that our null handling implementation executes faster compared to the Pandas operator. This operation is executed on column-wise.

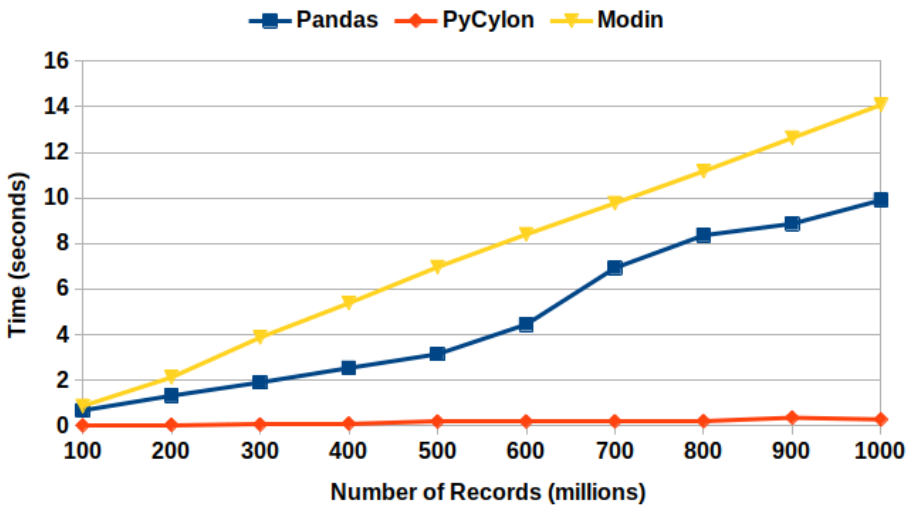

Figure 6.7: Null Handling (DropNa) Performance

6.6 Distributed Join Performance

In this experiment, we used 200M records per relation (for both left and right tables in a join) and scaled up to 128 processes. Here we generated random data by considering the uniqueness of data to be 10% such that the join performs under a higher stress considering hash functions and hash based shuffles. In the parallel exeperiments, each process will be loading equal amount of data such that the total amount of data is limited to 200M records. The results from figure 6.8 show that our distributed join implementation is faster compared to dask implementation. Also, the scalability in dask is not very strong compared scaling provided by PyCylon.

6.7 Distributed Drop Duplicates

Distributed drop duplication operation is a widely used operator to organize the data such that overall data in a distributed computation must be unique based. This is a vital operator when distributed deep learning is done in data parallel

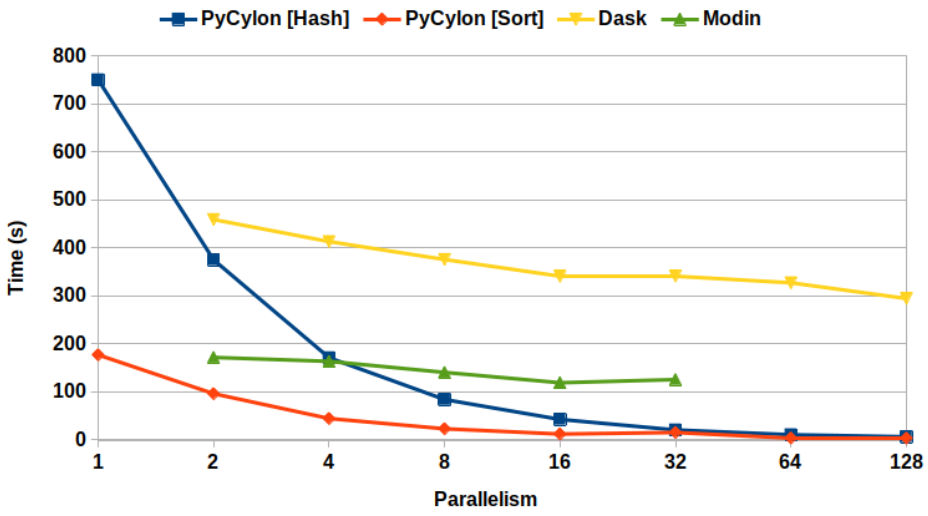

Figure 6.8: Distributed Join Performance

manner. For this experiment we used 500M records of data generated with 10% uniqueness where 90% of the data must be cleaned up for duplicate handling. Here the performance comparison compared to Dask distributed shows that PyCylon is scaling better than Dask. And also we observe that the Dask workload doesn't scale after parallelism 32.

6.8 Join with CPU and GPU

Since PyCylon only supports for CPU-based computations, we conducted an experiment to compare the join performance with CuDF on GPUs. Here we selected an experiment criteria where full resources of a CPU node is compared to the full resources of a single GPU device. For this comparison we selected, Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz as CPU and NVIDIA TESLA T4 for GPU. The CPU base experiments are executed on 48 cores while GPU based experiments are consuming all cuda cores (this depends on Cudf internal operations) of the GPU

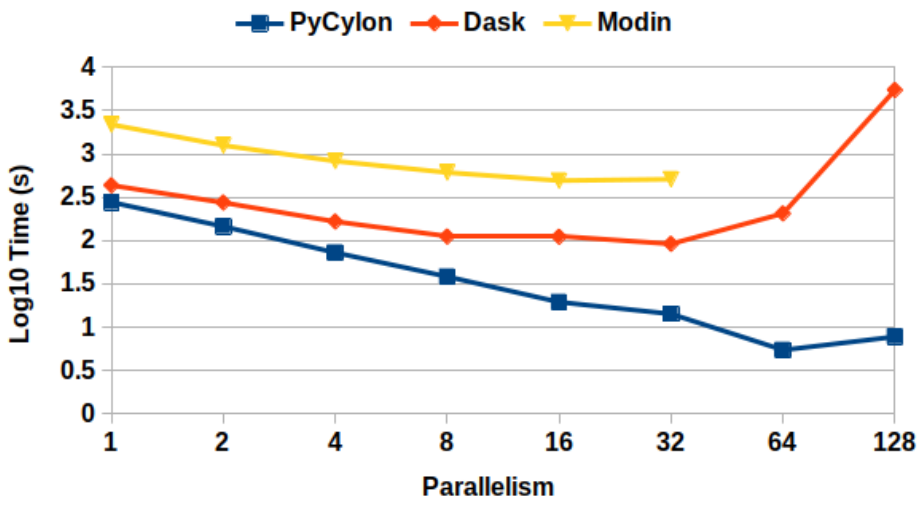

Figure 6.9: Distributed Drop Duplicates Performance

device. Here we selected the sort algorithm of PyCylon for join benchmark. Figure 6.10 shows that the performance of a single GPU device vs the used multi-core CPU has similar performance. Eventhough CPU with single core performs slower compared to fully Cuda core utilized GPU, the multi-core response provides the same result. This shows that, still CPUs can be used to get descent performance compared to a GPU. The important fact is that, the time taken for join operation is much higher compared to other data engineering operators. For instance the math operators in GPU is much faster that of in CPUs. But still in a data engineering query, if a single join query is there the time taken for that query is much higher compared to other vector operations. This also shows how resources can be utilized in a maximum manner. Another important fact is the memory limitation. The memory limit in the CPU node we used is 250 GB while the limit of the GPU device is 16 GB. In the particular CPU we can execute much larger data engineering workload compared to the Single GPU device for the same performance. In terms of pricing, in GCP GPU node hourly costs 2.33 USD dedicated and GPU node costs

0.35 USD. Compared to the workload size limitation, the capacity the CPU can handle is approximately 15 times. Here the cost vs the resource utilization ratio is much favorable to the CPUs. This doesn't state that the CPU is faster than GPUs, but the overall cost and computation capacity of using a CPU is favorable compared to a GPU.

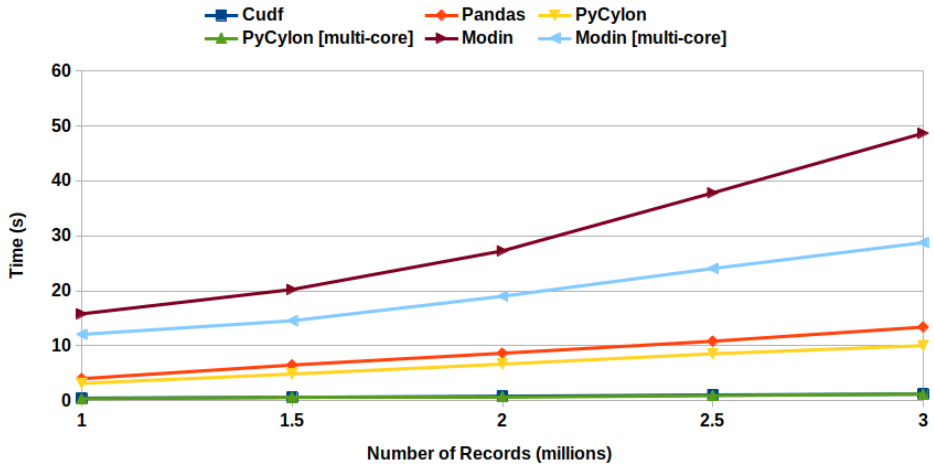

Figure 6.10: Join CPU vs GPU Performance

6.9 Overhead from Python

One of the most important attributes for creating high-performance data engineering libraries is that, understanding the overheads caused by language bindings. Especially, in the case of Python, there are some overheads when systems are designed in-efficiently. Here, we conducted a set of tests to evaluate the performance of the language bindings by considering a set of operators. Figure 6.11 shows that the overhead caused by language bindings is very small compared to the core C++ API. Especially, the overhead from the Python layer is negligible when compared

to Java. These results were obtained by experimenting with multiple processes for an inner-join with 200M records per relation.

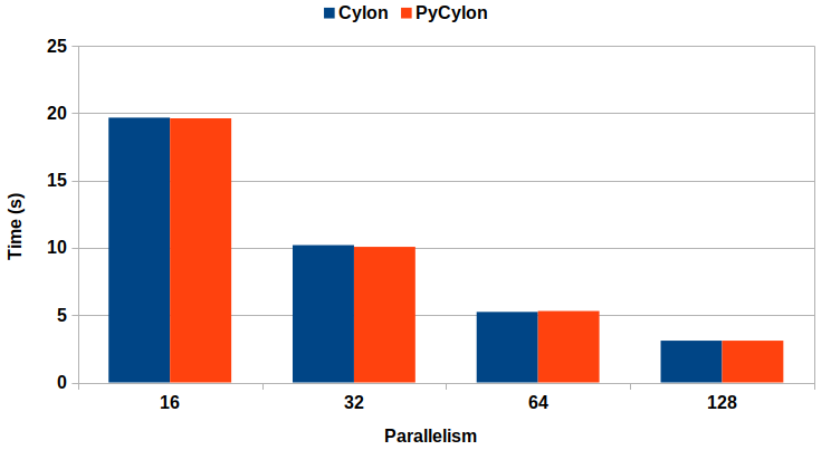

Figure 6.11: Performance Overhead by Language Bindings

CHAPTER 7

INTEGRATION WITH DEEP LEARNING FRAMEWORKS

Since the objective of our research is to make available the HPC-based data engineering framework for data analytics aware computations, we also designed our system such that we can effortlessly integrate with existing distributed data analytics frameworks by using the existing core of PyCylon. In a data analytics aware data engineering workload, there are three main factors that governs the usability and performance.

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

Single source is a very powerful concept when it comes to data analytics with data exploration. For such workloads feature engineering and data engineering components are extensively modified to see how the data analytics workload performs for different settings. In such cases, the data scientist must have the room to write the usual Python script and run the data analytics workload efficiently not only in single node, but also in across multiple nodes. Simple execution mode refers to running the workload with a simple mode to spawn the processes to run in parallel.

When considering vivid frameworks, they provide various ways to execute the framework on multiple nodes. For instance, frameworks like Dask require to start the workers and schedulers on each node and provide host information for distributed communication. On top of that, MPI provides a single execution command *mpirun* to spawn all the processes. Such factors are important in providing a unified interface to do deep learning easily. Also the execution mode on various accelerators for deep learning is a very important component. Majority of the frameworks supports both

CPU and GPU execution. So it is vital to provide the support to seamlessly integrate with these execution models to support data analytics workloads.

Figure 7.1 shows the high level component overlay of a data analytics aware data engineering workload. Here we have partitioned the workflow into 4 stages.

- Stage 1: In the first stage depending on the parallelism, the processes must be spawned. A unified process spawning mechanism which identifies worker information such as host ip addresses for each machine, network information, etc are identified at this stage.
- Stage 2: Worker information is extracted and data engineering operators will run in distributed mode on top of the data engineering platform which depends on the worker initialization component. Here the operations can be distributed or pleasingly parallel.
- Stage 3: For data analytics workload, the worker information, network information, chosesn accelerator and data must be provided from the corresponding data engineering process. This mapping is 1:1 for data engineering worker to data analytics worker. But also, this can be a many to many relationship.
- Stage 4: The worker information, network information and data will be used to execute the data analytics workload in distributed mode or pleasingly parallel manner.

Considering this generic overview on deploying deep learning workloads with data engineering workloads, we have integrated PyCylon with PyTorch distributed data-parallel, Horovod-PyTorch distributed data parallel and Horovod-Tensorflow distribtued data-parallel models. Horovod has become an state of the art distributed deep learning framework which supports a unified API for supporting distributed deep learning for multiple deep learning frameworks. PyTorch, Tensorflow and

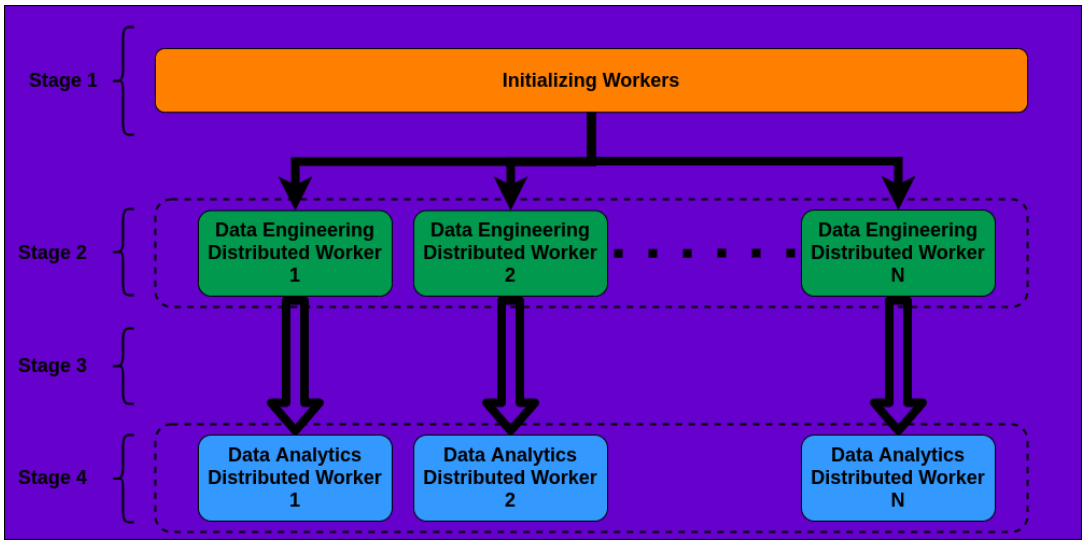

Figure 7.1: Integrating Data Engineering Workload with Data Analytics Workload

MXNet and supported by Horovod. In our research, we pay close attention towards PyTorch and Tensorflow. Horovod internally uses mpirun to spawn the processes and this model very well fit with PyCylon internals as we rely on mpirun to spawn the processes. This makes PyCylon uniquely qualified to become a supporting data engineering framework for Horovod.

7.1 PyTorch

PyTorch provides a distributed data parallel (DDP) module from the distributed runtime which allows the user to initialize an existing model using DDP to make available distributed computing easily. But the key factor is choosing the distributed runtime either MPI, NCCL or GLOO.

7.1.1 Stage 1

The first step is to initialize the runtime. Here either PyTorch distributed initialization or PyCylon distributed initialization can be called. But for specially on CPUs the PyTorch initialization must be called since PyTorch internally doesn't handle the MPI initialization check. But if we use NCCL as the backend this constraint is not there. This is one of the bugs we discovered from one of our previous research. For the PyTorch DDP the master address and port must be provided since the NCCL backend needs to identify which worker is going to be designated as the master worker to co-ordinate the communication. In addition to that the initialization method must be set. After the distributed initializaiton in PyTorch, PyCylon context must be initiailzed to set to distributed mode. After this stage we complete the requirements for stage 1 and partial requirements for Stage 3 (network information is also passed along with data in stage 3 which is being initialized in this step). Figure 7.2 shows a sample code snippet related to the initialization step.

```
def setup(rank, world_size, backend, master_address, port):
    os.environ['MASTER_ADDR'] = master_address
    os.environ['MASTER_PORT'] = port
    os.environ["LOCAL_RANK"] = str(rank)
    os.environ["RANK"] = str(rank)
    os.environ["WORLD_SIZE"] = str(world_size)
    # initialize the process group
    dist.init_process_group(backend=backend, init_method="env://")
    mpi_config = MPIConfig()
    env = CylonEnv(config=mpi_config, distributed=True)
    print(f"Init Process Groups : => [{hostname}]Demo DDP Rank {rank}")
    return env
```

Figure 7.2: Stage 1: Initialization for PyTorch With PyCylon

7.1.2 Stage 2

The data engineering workload is done as usual in PyCylon assuming the distributed mode initialization. Here what we do is join two table and use the join response for a deep learning workload. Here we call the distributed join by providing the initialized context information to the join function. At the end of this stage we create the resultant dataframe and later on in the stage 3, this dataframe can be used to generate the Numpy array required for deep learning. This stage is common for any framework including PyTorch, Tensorflow or any other deep learning framework. Figure 7.3 shows a sample data engineering workload for a data analytics problem.

```
user_devices_data = DataFrame(pd.read_csv(user_devices_file)) #read_csv(user_devices_file, sep=',')
user_usage_data = DataFrame(pd.read_csv(user_usage_file)) #read_csv(user_usage_file, sep=',')

print(f"Rank [{rank}] User Devices Data Rows:{len(user_devices_data)}, Columns: {len(user_devices_data.columns)}")
print(f"Rank [{rank}] User Usage Data Rows:{len(user_usage_data)}, Columns: {len(user_usage_data.columns)}")

print("--------------------------------")
print("Before Join")
print("--------------------------------")
print(user_devices_data[0:5])
print("-------------------------------------")
print(user_usage_data[0:5])

join_df = user_devices_data.merge(right=user_usage_data, left_on=[0], right_on=[3], algorithm='hash')
print("----------------------")
print("Rank [{}] New Table After Join (5 Records)".format(rank))
print(join_df[0:5])
print("----------------------")
feature_df = join_df[
    ['_xplatform_version', '_youtgoing_mins_per_month', '_youtgoing_sms_per_month',
     '_ymonthly_mb']]
feature_df.rename(
    ['platform_version', 'outgoing_mins_per_month', 'outgoing_sms_per_month', 'monthly_mb'])
if rank == 0:
    print("Data Engineering Complete!!!")
print("=" * 80)
print("Rank [{}] Feature DataFrame ".format(rank))
print(feature_df[0:5])
print("=" * 80)
data_ar: np.ndarray = feature_df.to_numpy()
```

Figure 7.3: Stage 2: PyCylon Data Engineering Workload

7.1.3 Stage 3

In the stage 3, the data from the stage 2 is used to create tensors required for the deep learning stage. Also we do the data partitioning for training and testing. This stage is different from framework to framework since the tensor creation and data partitioning steps can have various internal utils. Here we don't used data loaders or data samplers. But please note that these tools can be used to generate data loaders or data samplers. Figure 7.4 shows a sample code snippet for data movement from data engineering workload to data analytics workload.

```
data_features: np.ndarray = data_ar[:, 0:3]
data_learner: np.ndarray = data_ar[:, 3:4]

x_train, y_train = data_features[0:100], data_learner[0:100]
x_test, y_test = data_features[100:], data_learner[100:]

x_train = np.asarray(x_train, dtype=np.float32)
y_train = np.asarray(y_train, dtype=np.float32)
x_test = np.asarray(x_test, dtype=np.float32)
y_test = np.asarray(y_test, dtype=np.float32)

sc = StandardScaler()
sct = StandardScaler()
x_train = sc.fit_transform(x_train)
y_train = sct.fit_transform(y_train)
x_test = sc.fit_transform(x_test)
y_test = sct.fit_transform(y_test)

x_train = torch.from_numpy(x_train).to(device)
y_train = torch.from_numpy(y_train).to(device)
x_test = torch.from_numpy(x_test).to(device)
y_test = torch.from_numpy(y_test).to(device)
```

Figure 7.4: Stage 3: Moving data from Data Engineering workload to Data Analytics Workload

7.1.4 Stage 4

In stage 4, we initialize the deep learning model. Here we also initialize the DDP model by using the sequential model and we pass device information such that tensors and models are being copied to the corresponding devices (if accelerators are involved) for training and testing. This initialization part is vivid from framework to framework depending on the requirements and APIs. Figure 7.10 shows the initialization of a DDP model with PyTorch.

```
model = Network().to(device)
ddp_model = DDP(model, device_ids=[device]) if cuda_available else DDP(model)
loss_fn = nn.MSELoss()
optimizer = optim.SGD(ddp_model.parameters(), lr=0.01)

optimizer.zero_grad()
if rank == 0:
    print("Training A Dummy Model")
for t in range(epochs):
    for x_batch, y_batch in zip(x_train, y_train):
        print(f"Epoch {t}", end='\r')
        prediction = ddp_model(x_batch)
        loss = loss_fn(prediction, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Figure 7.5: Stage 4: Distributed Data Analytics Workload

7.2 Horovod with PyTorch

Horovod PyTorch provides the ability to scale on both GPUs and CPUs with a unified API. Here the importance is PyTorch doesn't need to be compiled from source to get MPI capability. Horovod has already offloaded the distributed trainer, opti-

mizer and allreduce communication packages so that the internal DDP mechanism in PyTorch is offloaded.

7.2.1 Stage 1

In the stage 1, the Horovod init method must be called to initialize environment. Followed by that the Cylon context can be initialized with distributed runtime true. Followed by this step if the GPUs are used the correct device must be set to PyTorch Cuda configs. To obtain the device ids we can either use the rank from Horovod initialization or PyCylon initialization. But Horovod at the moment support the local rank as well. It is more suitable in terms of effortlessly integrating with the distributed runtime for Horovod-PyTorch. Figure 7.6 shows a sample code snippet showing how this is done.

```
def setup():
    hvd.init()
    mpi_config = MPIConfig()
    env = CylonEnv(config=mpi_config, distributed=True)
    rank = env.rank
    print(f"Init Process Groups : => [{hostname}]Demo DDP Rank {rank}")
    cuda_available = torch.cuda.is_available()
    device = 'cuda:' + str(rank) if cuda_available else 'cpu'
    if cuda_available:
        # Horovod: pin GPU to local rank.
        torch.cuda.set_device(hvd.local_rank())
        torch.cuda.manual_seed(42)
    return env, device, cuda_available
```

Figure 7.6: Stage 1: Initialization for Horovod-PyTorch With PyCylon

7.2.2 Stage 2

Similar to section 7.1.2, the data engineering workload remains irrespective of the deep learning runtime.

7.2.3 Stage 3

Similar to section 7.1.3, the data engineering output can be converted to a Numpy array using the endpoints from PyCylon dataframe. Also the tensors can be created by providing the device ids obtained from horovod runtime and data can be prepared for deep learning workload.

7.2.4 Stage 4

In Stage 4, followed by the tensor creation step, the Horovod related initialization must be done to prepare the optimizers, network and other utils for distributed training. Here, for PyTorch-Horovod integration, the PyTorch's default neural network model, loss function, optimizer can be used as an input to the distributed computation enabled Horovod components. Here first the model parameters and optimizer must be broadcasted using the Horovod broadcast method from 0^{th} rank. There are two method calls designated for initial network values and the optimizer values. Also Horovod provides a compression algorithm to select whether a compression is required for distributed communication. After these steps, the distributed optimizer must be set by passing the initialized values. Figure 7.7 shows sample code snippet to initialize a the Horovod components for distributed data parallel deep learning with PyTorch.

```
# create model and move it to GPU with id rank
lr = 0.01 # learning rate
model = Network()
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=lr)
optimizer.zero_grad()

# By default, Adasum doesn't need scaling up learning rate.
lr_scaler = 1

if cuda_available:
    # Move model to GPU.
    model.cuda()
    # If using GPU Adasum allreduce, scale learning rate by local_size.
    if hvd.nccl_built():
        lr_scaler = hvd.local_size()

# Horovod: scale learning rate by lr_scaler.
optimizer = optim.SGD(model.parameters(), lr=lr * lr_scaler, momentum=0.01)

# Horovod: broadcast parameters & optimizer state.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.broadcast_optimizer_state(optimizer, root_rank=0)

# Horovod: (optional) compression algorithm.
compression = hvd.Compression.fp16

# Horovod: wrap optimizer with DistributedOptimizer.
optimizer = hvd.DistributedOptimizer(optimizer,
                                     named_parameters=model.named_parameters(),
                                     compression=compression,
                                     op=hvd.Adasum,
                                     gradient_predivide_factor=1.0)
```

Figure 7.7: Stage 4: Distributed Data Analytics Workload

7.3 Horovod with Tensorflow

Similar to PyTorch integration, Horovod also supports Tensorflow. Tensorflow has its own distributed training platform. It contains distributed mirrored strategy as the equivalent routine for distributed data parallel training.

7.3.1 Stage 1

Similar to PyTorch-Horovod integration, here we initialize Horovod and PyCylon. And also similar to PyTorch, here we also need to decide how the device is selected depends on the accelerator. The Tensorflow config API provides the listing of GPUs and this information is added to the Tensorflow configurations to make available all the GPU devices. Figure 7.8 shows a code snippet for the aforementioned initialization.

```
def setup():
    hvd.init()
    assert hvd.mpi_threads_supported()
    mpi_config = MPIConfig()
    env = CylonEnv(config=mpi_config, distributed=True)
    rank = env.rank
    world_size = env.world_size
    print(f"Init Process Groups : => [{hostname}]Demo DDP Rank: {rank} , World Size: {world_size}")
    gpus = tf.config.experimental.list_physical_devices('GPU')
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)
    if gpus:
        tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
    return env
```

Figure 7.8: Stage 1: Initialization for PyTorch With PyCylon

7.3.2 Stage 2

Similar to section 7.1.2, the data engineering workload remains irrespective of the deep learning runtime.

7.3.3 Stage 3

The data analytics data structure creation is different from framework to framework. Tensorflow has its own set of APIs to make these steps easier and structured. The Tensorflow dataset API can be used to create tensors from the numpy arrays and this API can be used to shuffle and create mini-batches as expected by the deep learning workload. Figure 7.9 shows a code snippet for the aforementioned steps.

```
data_features: np.ndarray = data_ar[:, 0:3]
data_learner: np.ndarray = data_ar[:, 3:4]

x_train, y_train = data_features[0:100], data_learner[0:100]
x_test, y_test = data_features[100:], data_learner[100:]

x_train = np.asarray(x_train, dtype=np.float32)
y_train = np.asarray(y_train, dtype=np.float32)
x_test = np.asarray(x_test, dtype=np.float32)
y_test = np.asarray(y_test, dtype=np.float32)

sc = StandardScaler()
sct = StandardScaler()
x_train = sc.fit_transform(x_train)
y_train = sct.fit_transform(y_train)
x_test = sc.fit_transform(x_test)
y_test = sct.fit_transform(y_test)

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))

print("=" * 80)
print("Tensorflow DataSets")
print("=" * 80)

BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100

train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

Figure 7.9: Stage 3: Moving data from Data Engineering workload to Data Analytics Workload

7.3.4 Stage 4

Horovod-Tensorflow also require a set of initialization steps to train a Tensorflow deep learning model. Similar to PyTorch, here the Tensorflow loss function, op-

timization function and neural network model are compatible with Tensorflow-Horovod internals. Here the gradient tape from Tensoflow autograd can be used and for this Horovod provides a DistributedGradientTape operator which takes the gradient tape instance as a parameter. Also priror to training, this Distributed-GradientTape must be initialized with the model parameters, loss function, and the optimizer values must be set to initial values. Similar to Horovod-PyTorch, the model parameters and optimizer values must be broadcasted using designated Horovod broadcast functions. Figure 7.10 shows a code snippet showing the aforementioned steps.

```
# define network
model = tf.keras.Sequential([
    tf.keras.layers.Dense(3), tf.keras.layers.Dense(1)])
# define loss function
loss = tf.losses.MeanSquaredError()
# define optimizer
opt = tf.optimizers.Adam(0.001 * hvd.size())

@tf.function
def training_step(images, labels, first_batch):
    # define a step function for training
    with tf.GradientTape() as tape:
        probs = model(images, training=True)
        loss_value = loss(labels, probs)

    tape = hvd.DistributedGradientTape(tape)

    grads = tape.gradient(loss_value, model.trainable_variables)
    opt.apply_gradients(zip(grads, model.trainable_variables))

    if first_batch:
        hvd.broadcast_variables(model.variables, root_rank=0)
        hvd.broadcast_variables(opt.variables(), root_rank=0)

    return loss_value
```

Figure 7.10: Stage 4: Distributed Data Analytics Workload

CHAPTER 8

IMPLEMENTING A SCIENTIFIC WORKLOAD

A scientific application is implemented with the designed framework with end-to-end workload containing data engineering and data science. Here our objective is to showcase how a sequential workload can be designed in a distributed manner using PyCylon and run a deep learning workload seamleesly using a single script with a unified runtime. Here we selected an application from academic sciences which involves Pandas dataframe for data engineering and PyTorch for data analytics. The original application is a sequentially executed application, we have implemented a distributed version of this application with PyCylon and Distributed PyTorch with unified execution.

8.1 UNO

UNO application is part of CANDLE[WYMY+, XAB+21] research conducted by Argonne National Laboratory focused on automated detection of tumour cells using a deep learning approach. With the dawn of deep learning domain science problems have gained a lot of attention in converting classical data analytical models to deep learning. The uniqueness of this workload is the composition of a heavy data engineering workload followed by a data analytical workload written in PyTorch. This provides us with an ideal scientific experiment to showcase the performance of an enhanced data engineering system to facilitate efficient data pipeline for a state of the art scientific problem.

The data engineering workload of the UNO application contains a set of steps to load the raw data and create the processed data set by using a sub-set of meta data associated with application. We partition the main dataset of 2.5 million records in a data parallel manner and use the meta data to pre-process the main dataset.

The goal of the UNO application is to provide cross-comparison of cancer studies and integrate into a unified drug response model. In high-level intuition is to train a deep neural network on tumour dose responses. Cell RNA sequences, drug descriptors and drug fingerprints are used as such responses to train the model.

UNO application consists of two main components. The first component is a data engineering workload which cleans the raw data to formulate the trainable parameters. In the UNO application there are multiple networks involved in the training process which are working on smaller datasets and larger datasets. In our research we focus our attention on the distribtued network which is designed to calculate the drug response based on the cell-line information. This network is a regressor which is being supported by two other networks which provides gene configuration based and drug feature based network.

8.2 Deep Learning Component

UNO refers to a unified deep learning model to predict drug response as a function of tumor and drug features for personalized cancer treatment. Precision oncology is focused on providing treatments for specific characteristics of a patient's tumor. The drug sensitivity is quantified by drug dose response values which measure the ratio of treated to untreated cells after exposing to a drug treatment with a specific drug concentration. In this application a set of drug data obtained from NCI60 human tumor cell line data base is used with to predict the drug response by considering gene expression, protein and microRNA abundance. As per the considered scope the UNO application we focus on the study conducted on single-drug response prediction done using NCI60 and gCSI datasets. We use 1006 drugs from NCI60 database for this evaluation and use gCSI for the cross-validation.

To evaluate the drug response predictions (regression model), the metrics used are R^2 (explained variance) and mean absolute error (MAE). The input features used to evaluate drug response are the cell-line gene expression profiles, drug chemical descriptors and molecular fingerprints. Here the drug response is modelled as a function of cell-line features and drug properties. The input features are engineered such that RNAseq expression profiles, drug descriptions, drug fingerprints and drug concentration is used as input parameters for the deep learning model.

8.2.1 Drug Response Regression Network

Drug response regression network is an ensemble model which uses two other networks to support the classification. This network uses rna-sequence data, drug feature data and drug concentration feature set as input features. The rna-sequence data becomes an input to a pre-trained model called Gene network. And the drug feature data are used to train a network called a Drug network. Concatenating the trained response over the data, a unified model is trained to calculate the drug response. Figure 8.1 refers to the gene network which is being trained prior to be an input to the main drug response model. Gene network only contains three dense layers each followed by a relu. Figure 8.2 refers to the drug network which contains 3 dense layers. This network is also pre-trained prior to be used in the drug response regression network.

Drug network and Gene network provide a set of concatenated parameters with another feature called concentration by formulating a 1537 (512 + 1024 + 1) input size layer for the unified drug response model. Within the drug response regressor, there is another residual block being used repeatedly. This layer is called drug response block module, which contains 2 dense layers followed by a dropout layer

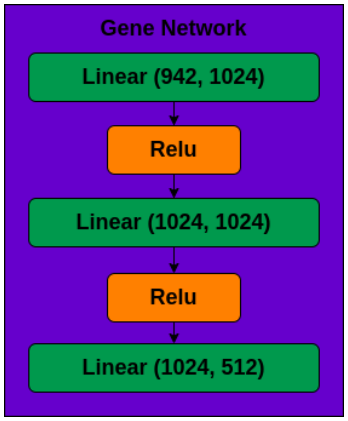

Figure 8.1: UNO DNN Architecture: Gene Network

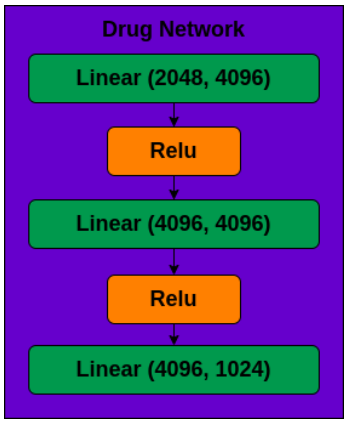

Figure 8.2: UNO DNN Architecture: Drug Network

and a relu activation layer. Figure 8.3 depicts the response block module.

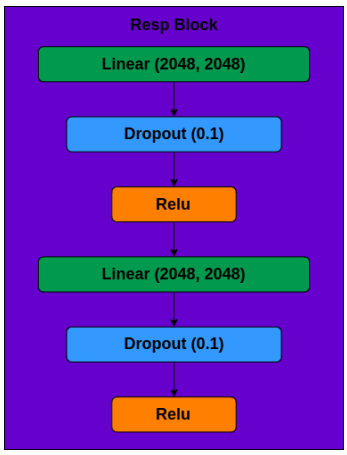

Figure 8.3: UNO DNN Architecture: Response Block Module

The ensemble model contains a dense input layer of shape 1537 to get the concatenated results of the gene-network and the drug network response along with the concentration value. Followed by the input layer, the residual blocks are stacked and a set of dense layers are stacked. And finally the regression layer contains a single output dense layer. Here the number of resp blocks can be customized to dynamically as well as the number of dense layers followed by it. All these parameters can be provided as a hyper-parameter in the application configuration file. Figure 8.4 shows the drug response regressor network.

Note that this network is trained in a distributed data parallel model since this network contains a very larger dataset and a complex network compared to the other networks trained simultaneously. The corresponding data engineering component is also distributed data parallel and discussed in detail in section 8.3.1.

UNO Deep learning component consists of 5 other auxiliary networks trained on a smaller data sets to predict a few other features. These networks are trained to get a broader view about the data related to drugs and cell information.In our

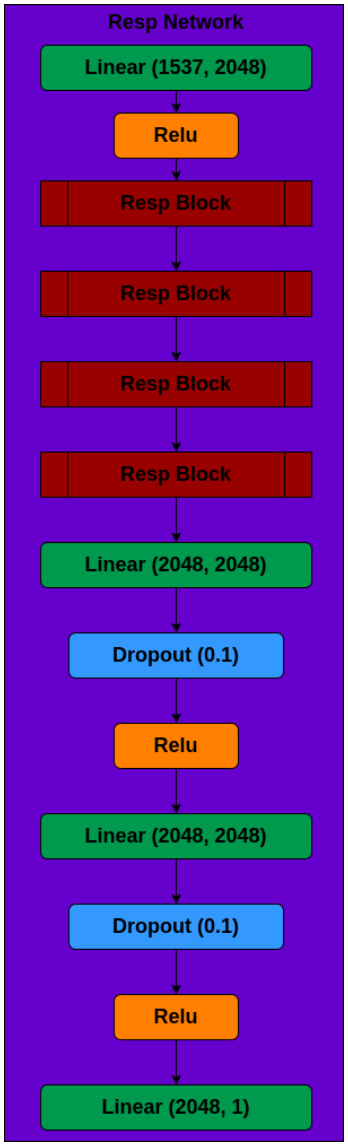

Figure 8.4: UNO DNN Architecture: Response Network

research we paid more attention towards the network with larger data to be trained efficiently. The other networks are as follows;

- Cell-line category classifier : Tissue category (normal vs tumor) classiciation
- Cell-line types classifier : Tissue type classification (melanoma, gynecologic, germ cell
- Cell-line Sites classifier : Tissue site classification (lung, skin, eye, etc)
- Drug target family classifier: Predict drug target family
- Drug QED weight classifier : Drug likeness score

The Cell-line related classifiers use a common network configuration defined as ClfNet. But for each classifer the number of hidden units, activations and output parameters are different.

8.2.2 Cell Line Category Classifier

The cell line category classifier uses the ClfNet to predict whether the cell category is a tumor, fibroblast or normal. Figure 8.5 shows the network architecture used for this network. The data used for this network is RNA sequence data and cell meta data. The cell-line category classifier is a customized output of a generic model designed for cell-based data analytics. In this classifier the last layer contains an output size of 3 to determine the 3 classes identified this classifier. Note that this network is trained sequentially and the corresponding data engineering component is also a sequential 8.3.2.

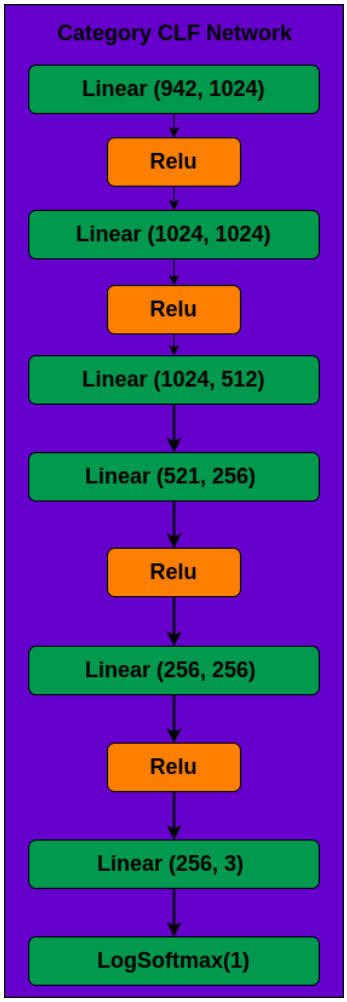

Figure 8.5: UNO DNN Architecture: Cell-line Category Classifier

8.2.3 Cell Line Types Classifier

The cell line types classifier also uses the ClfNet to predict the cell-line types. This is also a network customized by using the generic network created for cell-line based analytics. Figure 8.6 refers to the cell-line types classifier. The difference from the generic network is is the output size of the last layer which corresponds to the number of classes (18) predicted by this network. The cell-line types that are classified in this network are;

- gynecologic
- prostate
- lung
- kidney
- bladder/urothelial
- germ cell
- squamous
- melanoma
- sarcoma/mesothelioma
- head and neck
- endocrine and neuroendocrine
- digestive/gastrointestinal
- unknown
- breast
- hematologic/blood

- skin other
- neurologic
- liver/bile duct

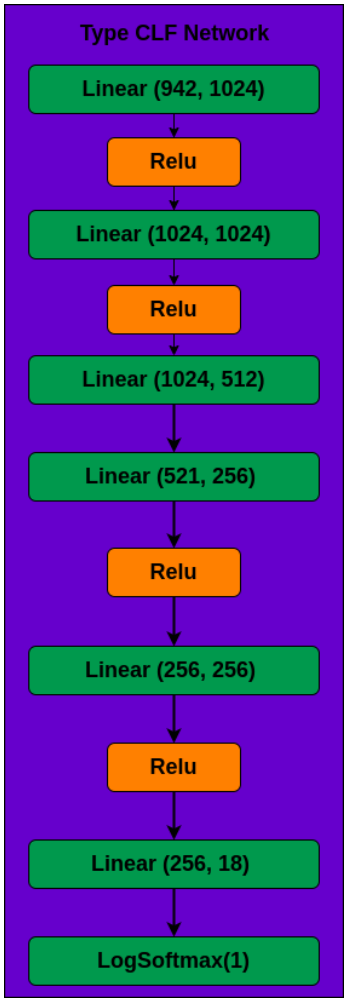

Figure 8.6: UNO DNN Architecture: Cell-line Type Classifier

Note that this network is trained sequentially and the corresponding data engineering component is also a sequential 8.3.2.

8.2.4 Cell Line Sites Classifier

The cell-line sites classifier is used to predict the cancer sites. For this classification, the same generalized network is used but the number of classes are 17. Figure 8.7 shows the network configuration used for this classifier. The cancer sites identified by this network are as follows.

- musculoskeletal
- gynecologic
- testes
- prostate
- lung
- skin
- kidney
- bladder/urothelial
- head and neck
- endocrine and neuroendocrine
- digestive/gastrointestinal
- breast
- hematologic/blood
- eye
- liver/bile duct
- neurologic
- unknown

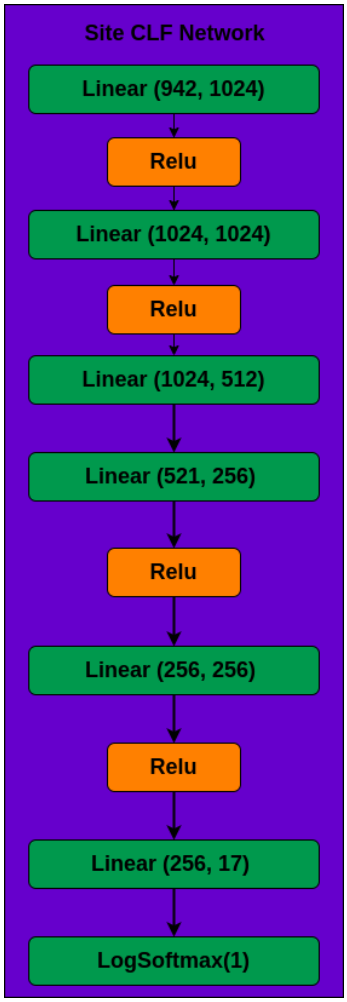

Figure 8.7: UNO DNN Architecture: Cell-line Site Classifier

Note that this network is trained sequentially and the corresponding data engineering component is also a sequential 8.3.2.

8.2.5 Drug Target Family Classifier

The drug target family classifier network is designed to identify the drug family. This network also uses the generic cell-line classifier network to model the required classifier. The drug families classified by this network are as follows;

- chaperone
- transferase
- enzyme modulator
- hydrolase
- receptor
- nucleic acid binding
- transporter
- signaling molecule
- transcription factor
- oxidoreductase

The data processing relevant for this network is discussed in the section 8.3.3. This network is also trained sequentially and the corresponding data engineering workload is also sequential.

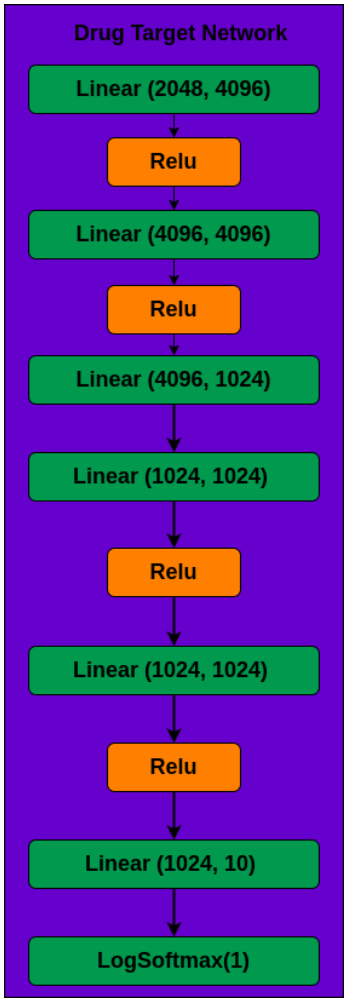

Figure 8.8: UNO DNN Architecture: Drug Target Family Classifier

8.2.6 Drug QED Regression Network

The drug QED network refers to the calculation of quantitative estimation of drug-likeliness. This is a very important study for selecting compounds in the early stages of drug discovery. This regression network is designed to obtain the likeliness score for the drugs used in this analysis. Figure 8.9 shows the network designed to calculate the druglikeliness. This network is also trained sequentially as per this application and the corresponding data engineering workload is discussed in section 8.3.3.

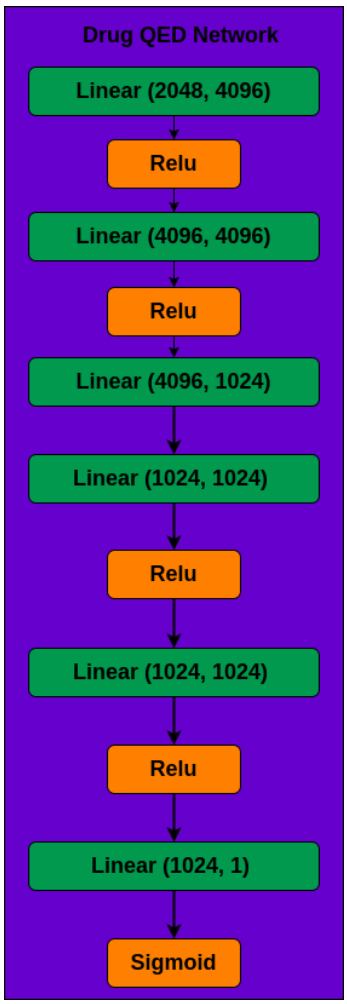

Figure 8.9: UNO DNN Architecture: Drug QED Regression Network

8.3 Data Engineering Component

UNO application uses 2.5 million samples of cancer data across six research centres. This model analyses the study bias across these samples to design a unified drug response model. Before building this model, the application comprises a heavy data engineering workload written in Pandas. The data engineering component is over 3000 lines of code in Pandas. This application uses the following data engineering operators.

- concat (inner-join)
- to_csv
- rename
- read_csv
- astype
- set_index
- map
- isnull
- drop
- filter
- add_prefix
- reset_index
- drop_duplicates
- drop_duplicates (unique)
- not_null

- isin
- dropna

The existing data engineering workload is written in Pandas and doesn't scale. We re-engineered this application to a scalable data engineering workload and design a seamless integration between data analysis and data engineering workload consuming state of the art high-performance computing resources. Also we integrated a Modin based implementation to show case the performance comparison with our implementation. The data engineering workload is executed in CPU-based distributed memory and the data analytical workload. We use Pytorch for data analytics workload and extend to use PyTorch distributed data-parallel training. Also, our objective is to integrate a HPC based full-stack of data analytics aware data engineering for scalability. This feature is only supported by PyCylon at the moment. And also, we stress out the importance of designing a BSP based model for deep learning workloads associated with data engineering components for better performance and scalability in HPC hardware.

8.3.1 Drug Response Data Processing

The data analytics component requires a set of features to be engineered from the raw data. Here there are three main datasets required to create the complete dataset used to create the drug response model. Figure 8.10 refers to the main dataset which contains the drug response. The raw dataset contains additional features so in the initial stage the data is loaded and the expected features are extracted by a column filtering operation, select. Then a map operation is performed to pre-process a drug id column to remove symbols from the columns to create a consistent drug id. Once the data is cleaned, it is scaled by using the Scikit-learn pre-processing

library for scaling numerical values. Then the data is fully converted into a numeric type to provide numeric tensors at the end for the deep learning workload. In the parallel mode, we partition this dataset with the set parallelism and passed to the corresponding operators.

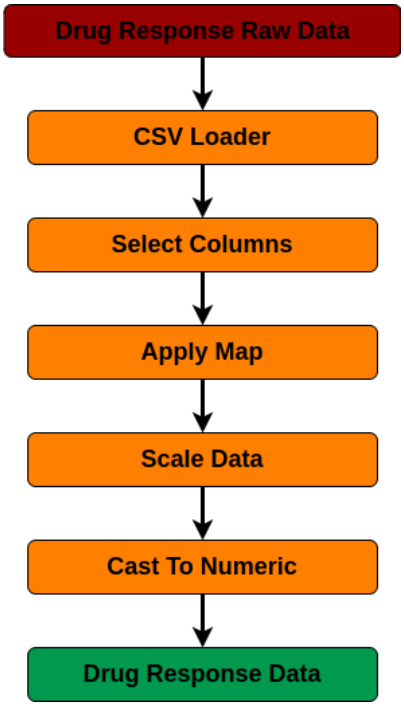

Figure 8.10: Drug Response Data Processing

To formulate the global dataset, we require two other datasets which is used as a meta-data to filter and process the the main drug response dataset. The first dataset is the drug feature raw dataset. This dataset contains drug features required to be in located in the drug response data. Here there are two sub-data sets contributing to formulate the drug feature dataset. We merge them by performing an inner join on the dataset based on the index formed on the drug ids. After that we cast the data into numeric types and output as a numeric array which later converted to a numeric tensor for deep learning. This data processing workflow is shown in figure

8.11.

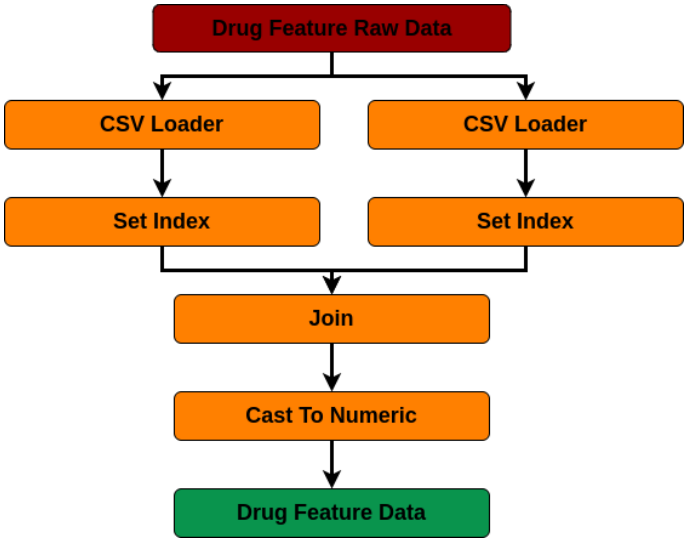

Figure 8.11: Drug Feature Data Processing

The other dataset required is the RNA sequence data set containing information about RNA sequences. Here the data set is first processed to remove specific symbols by a map operation and then duplicate records are dropped by using drop duplicate operator. Then an index is set for this dataset and later on scaling is done on the numeric data using the Scikit-Learn preprocessing library. Finally the data is casted to a numeric type and pre-processed RNA-sequence data is formulated as a Numpy array which later converted into a numeric Tensor for the deep learning workload. This data processing pipeline is shown in figure 8.12.

Once the Drug response initial dataset, drug feature data and RNA-sequence data is pre-processed the final dataset for drug response model is engineered as shown in the figure 8.13. The processed drug response data is further feature selected and unique operation is applied. Then the RNA sequence data is filtered by checking whether specific drug related RNA sequences are present and the same is done for the drug feature data set. These two operations are done by the isin operator. Then

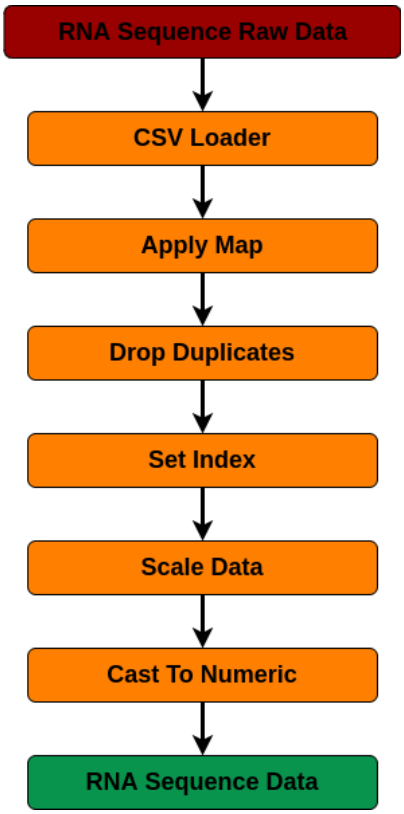

Figure 8.12: RNA Sequence Data Processing

the common drug set is selected by performing a and operation and later use these common drug related drug response data filter to get the final drug response data.

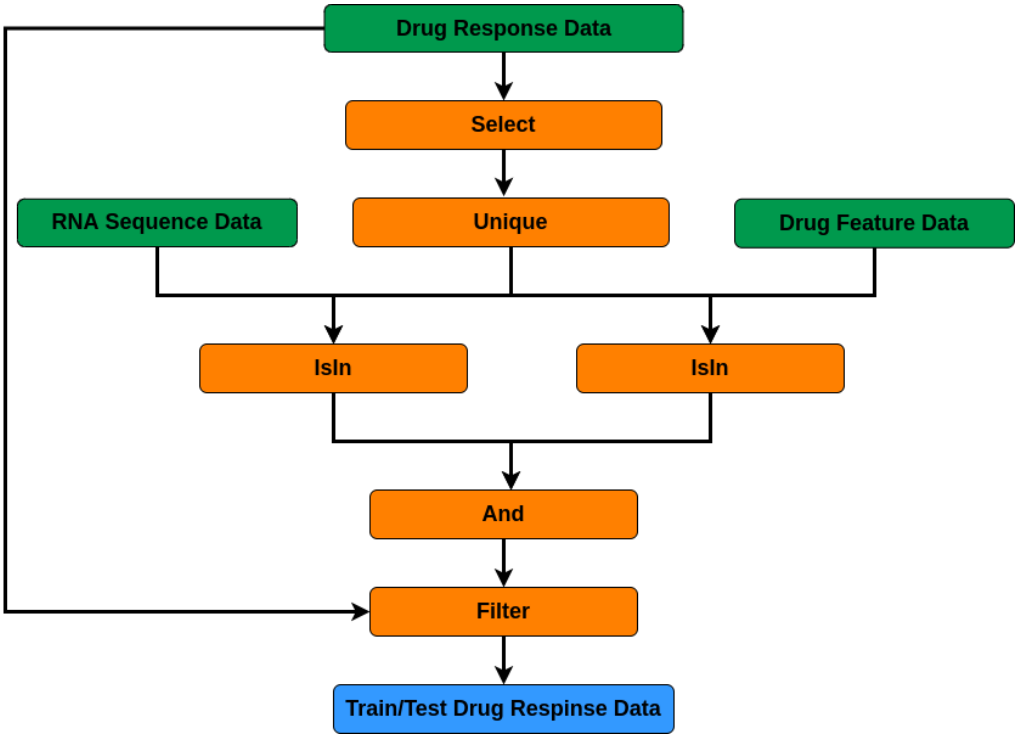

Figure 8.13: Drug Response Overall Data Processing

Among the operators used, since we partitioned the data each data engineering operator can work independently in a pleasingly parallel manner. But we can use the distributed unique operator to make sure no duplicate records are used for deep learning across all processes. Note that the data engineering component of this application is basically feature engineering meta data and use them to filter a very large dataset which is converted to formulate the expected input for the drug response model.

8.3.2 Cell-line Data Processing

The cell-line data processing component produces the numerical data required for the data analytics in the neural networks discussed in section 8.2.2, 8.2.4, 8.2.3 and 8.2.5. Here the first step is to load the cell-line information by pre-processing the raw data for cell-line information. Figure 8.14 shows the initial step to pre-process the cell-line meta data to extract the cell-meta data. Here the operations are executed in a sequential passion, followed by a set of filtering operations. And a encoding operation is used to convert the string naming conventions to a numeric category for classification. In the final step, the data is converted to a numpy array later to be converted into a tensor for the deep learning workload.

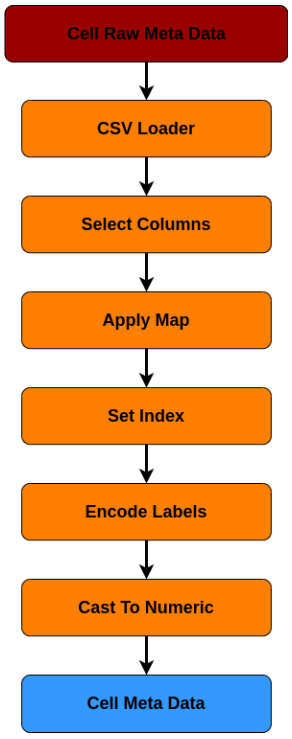

Figure 8.14: Cell-Meta Data Processing

Using the pre-processed cell-line meta data, the overall feature set required for the deep learning component is engineered as shown in the figure 8.15. Here we use

the pre-processed RNA-sequence data along with the cell-line meta data to formulate the final feature vector by performing a join operation on the drug sample column information in RNA-dataset.

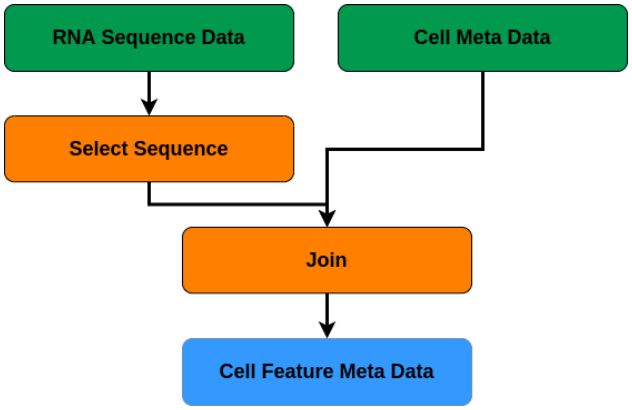

Figure 8.15: Cell Feature Meta Data Overall Processing

8.3.3 Drug Property Data Processing

The drug property data engineering component produces the data required to train drug target family classifier 8.2.5 and drug QED regression network 8.2.6. Initially the drug property data is pre-processed by loading the raw data CSV and filtering columns and setting up an index followed by a numeric cast option. This is a very straightforward sequential data processing component. Thus the drug property data is pre-processed as shown in the figure 8.16.

The pre-processed drug property data set is used to obtain the dataset required for the drug QED regression network. Figure 8.17 shows the corresponding workflow. Here the drug property data is used and drop all the null values, scaled using Scikit-learn pre-processing library and finally cast to numeric types to be used in the deep learning network to form tensors. Using the pre-processed drug feature data (a

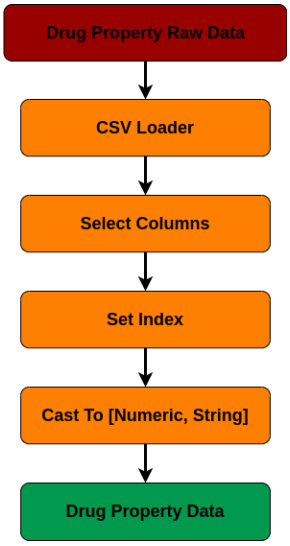

Figure 8.16: Drug Property Data Processing

sub input to the drug response network) and drug QED feature data the dataset required for drug QED regression network is formulated as shown in figure 8.18.

8.4 Performance Evaluation

The original application implemented is a single threaded application which is implemented on Pandas for data engineering and PyTorch for deep learning. Our main goal was to implement the sequential version of the application and improve the sequential performance. After the first stage we conducted distributed experiments to see how we can scale our workload on CPUs for data engineering. We also extended the deep learning component of this application by integrating with PyTorch distributed execution framework on both CPUs and GPUs using MPI and NCCL respectively. In this benchmark our objective was to seamlessly integrate a deep learning aware data engineering workload using a single Python data engineering and deep learning script with a single runtime. Also note that we use the

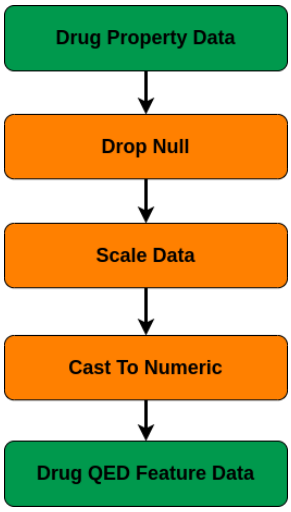

Figure 8.17: Drug QED Feature Data Processing

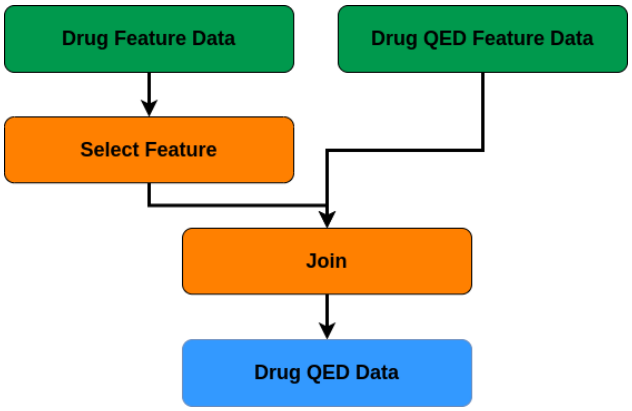

Figure 8.18: Drug QED Data Processing

drug response network related larger data distribution for the application benchmark while the smaller networks only takes a very smaller execution time compared to this larger model.

For the experiments we used two set of clusters for CPUs and GPUs. For CPUs, we used the future systems Victor cluster with 6 nodes and 16 processes per each on the maximum parallelism. This cluster contains Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz machine per each node. For GPU experiments we used Tesla K80s with 8 GPU devices on Google Cloud Platform. For single-node single-process executions, we used same Victor nodes. Here for the sequential performance comparisons we use Pandas, PyCylon (single core) and Modin (single core). And for the distributed performance comparisons we use PyCylon and Modin on single node multi-core scaling. We selected Modin instead of Dask, because it is more close to the data engineering stack proposed by PyCylon because of eager execution and the ability to convert an existing Pandas data engineering workload in a straightforward manner.

8.4.1 Data Engineering Sequential Performance

We first conducted a set of experiments to evaluate the single process execution of the propsed system PyCylon, Modin and Pandas. Modin provides the ability to convert a Pandas data engineering workload by a single line of code where PyCylon provides a dynamic API for the user to decide the nature of sequential and parallel operators in a dynamic manner. Here we evaluated the data engineering performance for the drug response data pre-processing workload used for the drug response regression network. Figure 8.19 shows the single core performance for the aforementioned data engineering workload. Here we observe that the performance of PyCylon and

Pandas are very similar while Modin is quite slower. This performance improvement includes data loading efficiency plus overall operator performance improvements. But in a general way, Pandas and PyCylon have almost similar performance in most operators except for data loading, duplicate handling, null handling and search operations involved in this application. Note that both PyCylon and Modin are evolving data engineering frameworks to support data engineering on a tabular data. In the distributed performance evaluation section 8.4.2.

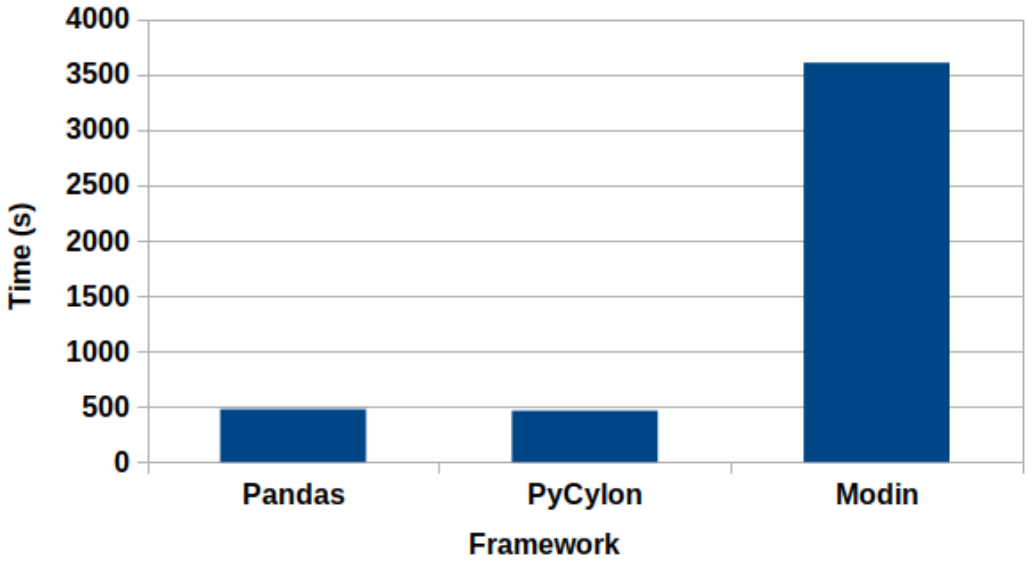

Figure 8.19: Sequential Data Engineering

We investigated the underlying sub components to understand why the sequential performance in PyCylon is better compared to Pandas. Table 8.4.1 shows the time taken for significant sub-components. The time breakdown shows that a set of components are taking a much longer time for the sequential execution. The drug response data loading, triming data, drug analysis and data split takes a very long time. We observed that Modin is slower in loading data compared to PyCylon and much slower in casting data which is a part of the drug response data loading com-

ponent. The drug analysis component contains iterating through the dataframe to create a subset of data by doing a statistics calculation using Scikit-learn. The looping through the dataframe is quite slow in Modin compared to both Pandas and PyCylon. The split operation uses the Scikit-learn pre-processing library to easily partition a dataframe as expected by passing through hyper-parameters. Here PyCylon can do zero-copy and convert into a Pandas dataframe to do this efficiently while Modin cannot be converted to a Pandas dataframe. Irrespective of the third party library performance with Modin, we observe that the core operators in dataframe are quite slower in Pandas when it comes to the Uno application.

Data Engineering Component	PyCylon	Modin
Drug response load	34.06	254.95
Drug feature extraction	1.42	0.44
RNA sequence load	2.47	3.33
Trim data	14.18	64.38
Drug analysis computation	386.25	744.38
Cell meta data load	0.044	0.33
RNA feature extraction	0.76	23.24
Data split	0.34	2515.05

Table 8.1: PyCylon vs Modin Sequential Time Breakdown for Data Engineering

8.4.2 Data Engineering Distributed Performance

First we conducted a single node performance evaluation metric based on various data engineering components in the application. Here we compared the multi-core performance of Modin vs PyCylon. We encountered scaling the Modin dataframe across nodes so, we conducted this initial set of performance to compare the performance with Modin. The current Modin documentation also mentions that the distributed component is experimental. And also we found out most of the Modin benchmarks in the published research are done on multi-core by comparing with

Pandas. Figure 8.20 shows the multi-core data parallel data engineering time breakdown for PyCylon vs Modin. Here we observe that the PyCylon is scaling relatively well compared to Modin.

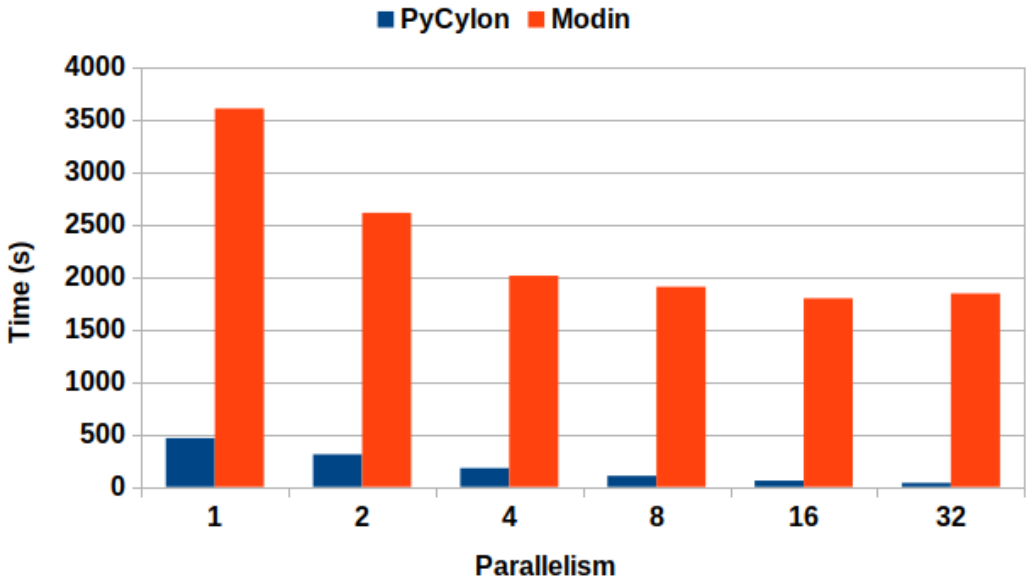

Figure 8.20: Multi-Core Data Parallel Data Engineering Performance

Considering the speed up gain compared to the base implementation of each framework, we plotted the speed up graphs as depicted in figure 8.21. The speed up from Modin is relatively low compared to PyCylon. Here the Modin dataframe is internally using Ray to scale up the dataframe. This is the default execution engine for Modin. By design, Modin doesn't have its own distributed execution engine, but rely on Ray to do the distributed computation. In PyCylon we have multiple modes of executing the application, distributed data parallel, pleasingly parallel and sequential. With the nature of this application we use a pleasingly parallel approach to execute the application. We also investigated whether Modin can provide dynamic parallelism as required by the application and we found out Modin doesn't have this capability. In case of Modin, the output is shown as a

sequential view or a one dataframe and operators are executing in a distributed manner. But in PyCylon's case, the dataframe is in the distributed memory. Each process contains its own dataset corresponding to the dataframe. We evaluated and verified the accuracy of the data engineering component by calculating the number of data points produced by the sequential version compared to the output from the distributed versions. In addition, we also performed micro-level validations for data engineering steps to verify that we are using correct number of drugs and cell-line information at intermediate stages of the distributed computation.

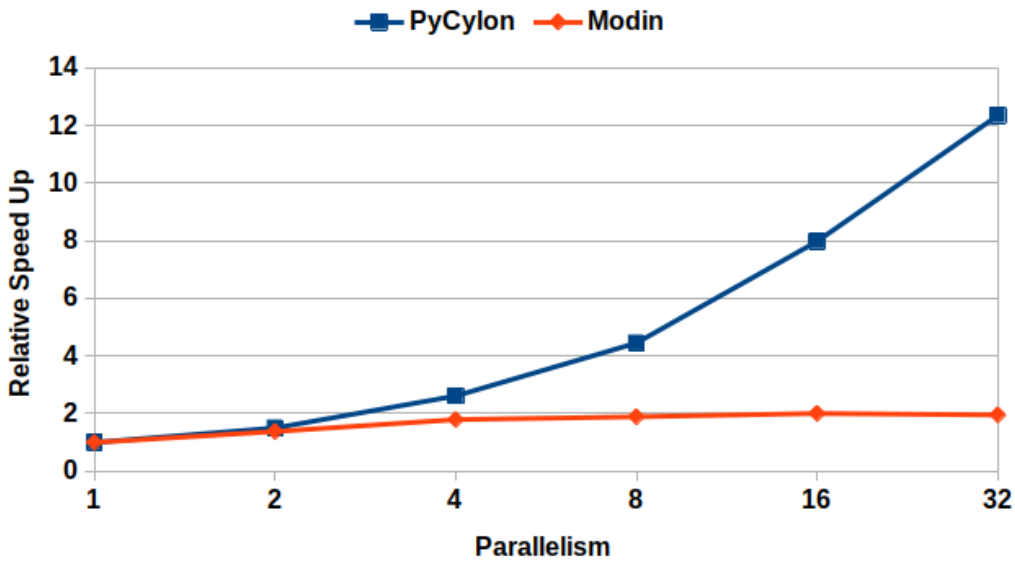

Figure 8.21: Multi-Core Data Parallel Data Engineering Speed Up

In order to understand how the parallelism works for the data engineering workload, we did a micro-benchmark for the multi-core experiments by partitioning the data engineering components into main component which produces sub-datasets. Figure 8.22 shows the time breakdown for these data engineering components. These results were taken by running the application in distributed mode on 32 CPU cores on a single machine. Here we can see that the majority of the time is taken for drug

analysis computation. This workload is a numerical calculation done on a drug analysis data. In this computation the majority of the time is taken because there is a iterative computation happening on Python with a Scikit-Learn data processing library. We also observed that, the majority of this time is taken on iterating the loop in Python. Here 2.5M samples are being iterated to do this calculation. We didn't improve this performance by doing further forking processes since it hinders distribution execution of threads along with MPI processes. But this execution can be improved by offloading this to a C++ kernel. But in general data engineering practices, we cannot introduce generic kernels to do such operations which are application dependent.

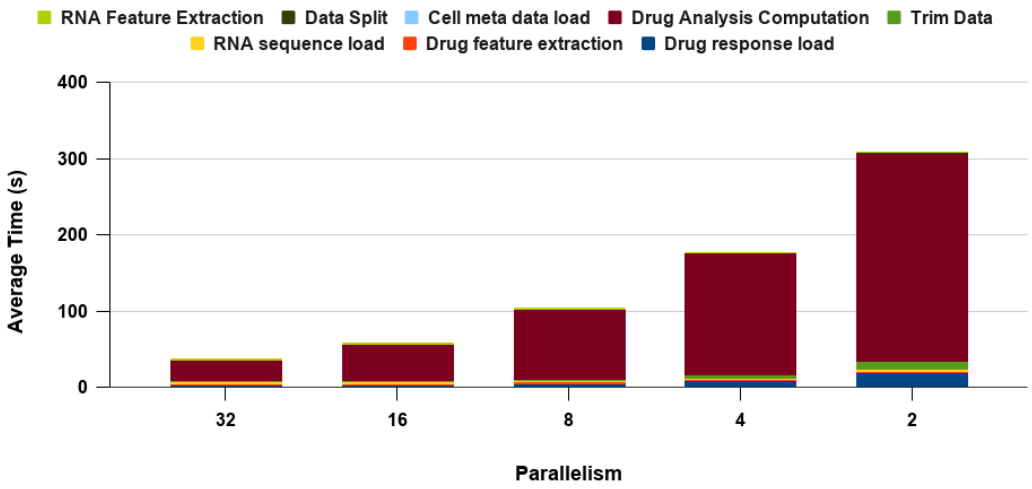

Figure 8.22: Distributed Data Engineering Time Breakdown

Also figure 8.23 shows the time breakdown for the same experiment conducted on time breakdown based on total time as a percentage. Here it is clearly seen how distributed components reduces as parallelism increases and how non-parallel components related to meta-data pre-processing is a constant component of the

overall workload. The figure clearly shows that the majority of the time is spent on the drug analysis computation.

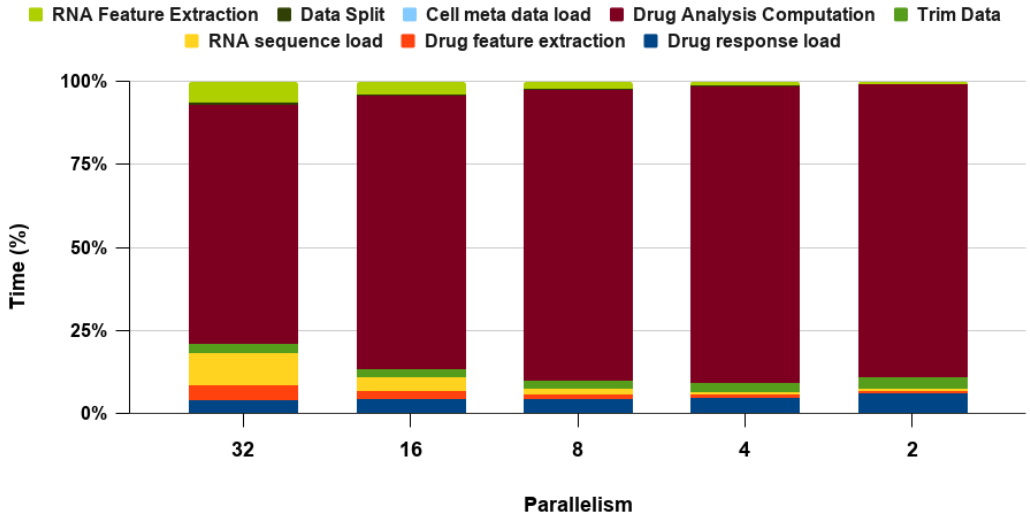

Figure 8.23: Distributed Data Engineering (CPU) Percentile Time Breakdown

Figure 8.24 shows the data engineering time for distributed experiments across multiple nodes. Here we used 6 physical nodes of the victor cluster and each node uses 16 processes for the computation. Here we observe that even though the workload is scaling, the scale up factor is not that significant. The major reason for this is the dominant drug analysis computation in the data engineering component. Even though the data is partitioned, the majority of the time is spent on this component compared to other components. We encountered problems in scaling Modin dataframe across multiple nodes. Also note that Modin framework is also an evolving framework similar to PyCylon in the parallel dataframe domain.

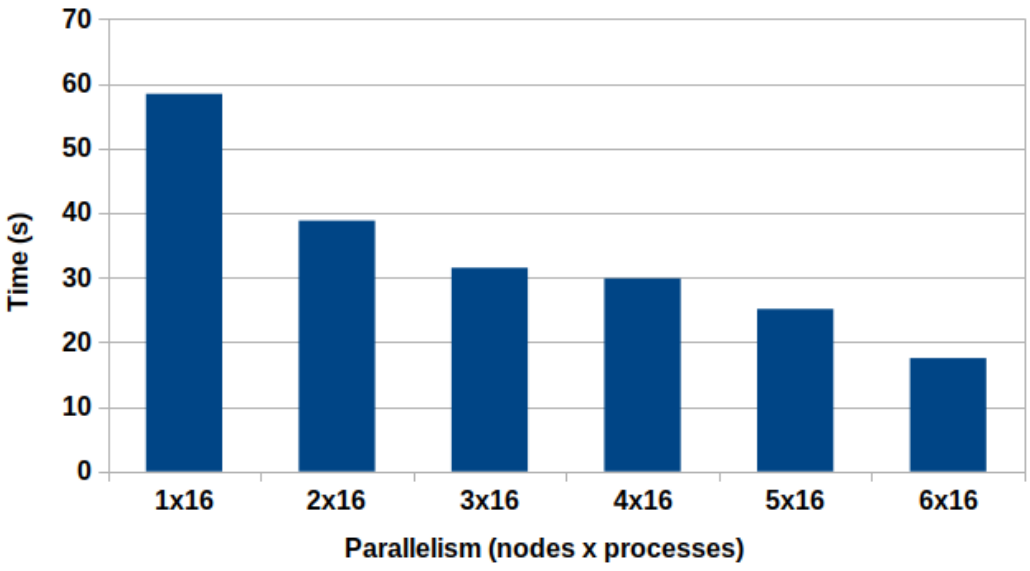

Figure 8.24: Distributed Data Parallel Data Engineering

8.4.3 Data Analytics Distributed Performance

For the data analytics scaling experiments we used PyTorch distributed communication framework with MPI for CPUs and NCCL for GPUs. The single process experiment results are same for both PyCylon and Pandas and both are using the same PyTorch code base for this. And also, all the data are in-memory prior to deep learning workload, so there is no overhead in loading data to create minibatches. The experiments conducted on CPUs are scaling well across multi-nodes, but we also observe a slight memory overhead causing the application to scale below the ideal scaling. We conducted more experiments to evaluate if there is an overhead from the data engineering framework, but we observed no significant overheads causing less scaling on CPUs. Figure 8.25 shows the single process and distributed experiments carried out on CPUs. Here we use the PyTorch build from source to enable MPI execution as it is a requirement forced by the framework. One significant factor is that PyTorch becomes an ideal distributed computation deep learning framework

for PyCylon since, PyCylon also supports and MPI backend for distributed computation.

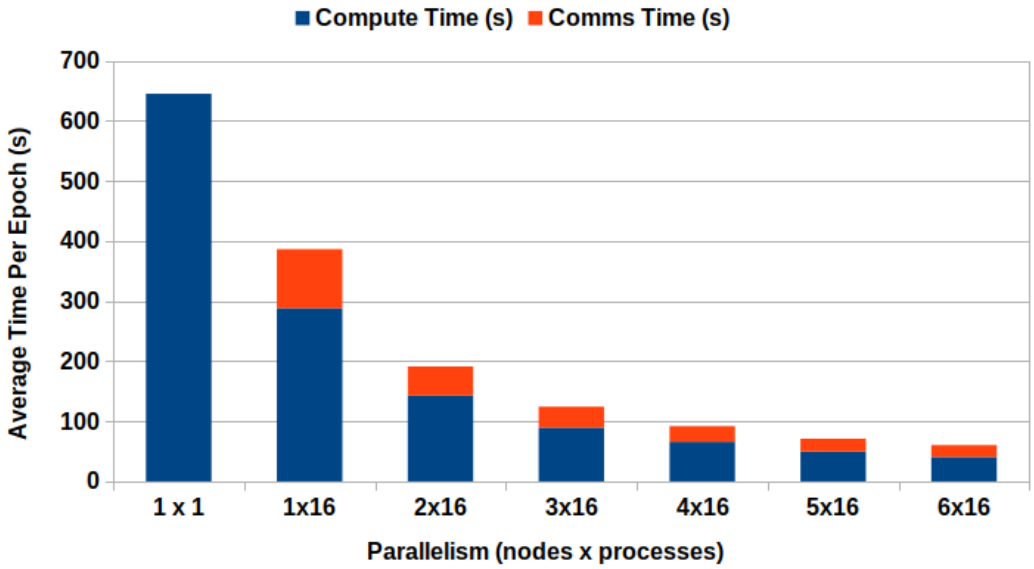

Figure 8.25: Distributed Data Parallel Data Analytics on CPU

For the GPU-based experiments we used a single node multi-GPU experiment setting to see how the data analytics workload can be scaled on NCCL execution framework with PyTorch. Figure 8.26 shows the results for single GPU and multi-GPU experiments. Here we observe that the execution time is dominated by the communication time. With the increase of parallelism, the number of communication across devices increases but the number of batches that has to be sent across devices lowers. So that it gives an advantage in scaling. When we consider the computation time, we observe that scaling happens closer to the ideal point of scaling in all parallel settings. Also, we observe that the computation is much faster in parallelism 2 compared to parallelism 1 where the memory overhead is 50% lesser compared to the sequential execution. When considering the CPU vs GPU perfor-

mance for the deep learning workload, we observe that the speed up from GPUs is 2x compared to the CPUs compared to this network.

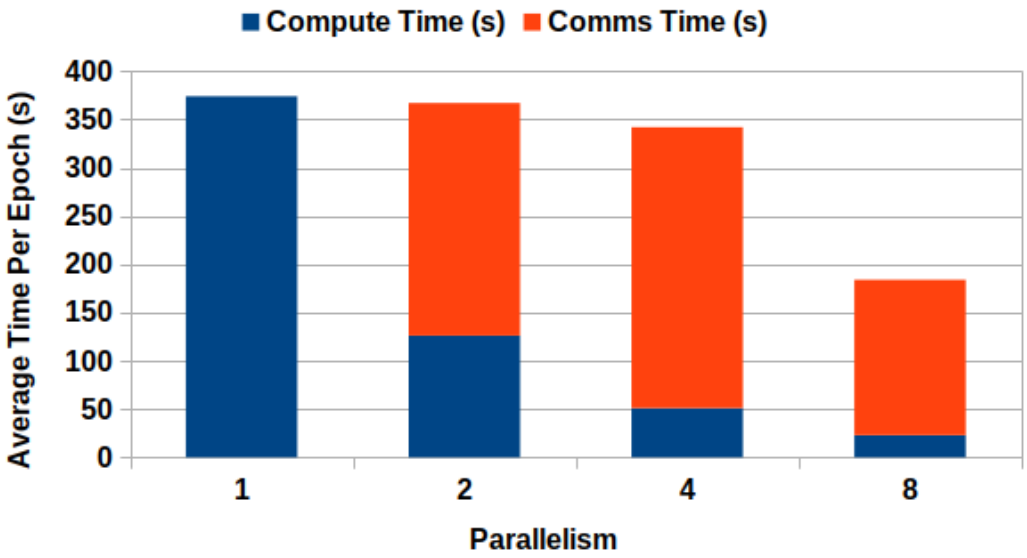

Figure 8.26: Distributed Data Parallel Data Analytics on GPU

CHAPTER 9

CONCLUSION

The distributed data engineering framework, PyCylon currently contains over 40 dataframe operators with the capability of scaling up to multiple machines or run sequentially. From our current micro-benchmarks conducted on each data engineering operator, it shows that our research effort is promising even in single process execution compared to the state of the art dataframe Pandas. Besides, PyCylon scales well compared to the state of the art distributed data engineering framework Dask under computationally intensive workloads. In terms of usability and adaptability, we have also designed a very familiar API compared to Pandas and provided distributed computing capability on dataframe by changing a few lines of code. Unlike the existing work, our dataframe is specially designed for HPC workloads and works well on HPC hardware. In terms of an external viewer, PyCylon dataframe is basically a dataframe for MPI which was a missing component until now. Besides evaluating the framework just on operators, we also engineered our framework to design a real world scientific workload. Here we designed an end-to-end data analytics aware data engineering workload using PyCylon and obtained better results compared to the original implementation. Since the original implementation was just sequential, we were also able to provide scalability for the application. This effort shows that, our data engineering framework can be a promising tool for scaling data engineering workloads specifically on HPC.

CHAPTER 10

RESEARCH GOALS IN ACTION

The focused research problems and research goals have been transformed to practical research outcomes as follows.

1. *Evaluating the limitations of existing big data frameworks for distributed data analytics:* Regarding Java based data engineering and data analytics, the research work we conducted in the past two years were mainly focused on JVM-based high performance data engineering and analytics[AFK19, AFK$^+$]. We researched on implementing machine learning algorithms in distributed memory using high performance Java and C++. One major discovery was that, even though JVM-based systems can be further enhanced for high performance using high performance kernel interfaces for JVM, an equivalent C++ implementation outperformed JVM-based implementations for large scale data analytics with higher dimensionality. In the related research we used the state of the art BLAS and MPI libraries for high performance computation and communication. This motivated us to investigate deeply into C++ based high performance kernel development for data engineering. The literature review conducted on the existing Python-based data engineering and some of our preliminary research shows that, these systems can be further enhanced [APW$^+$20].Besides our deeper study on the internals of a system, the major data analytic tools like PyTorch[PGM$^+$19] and Tensorflow[ABC$^+$16] show cases that the necessity of standalone data analytics tools specialized for specific tasks. This is a scope beyond just big data computing. Besides, this shows that the importance of being seamlessly integrated with the existing software stack for data analytics and making compatible data engineering systems to run at scale.

2. *Importance and necessity of high performance computing for data analytics aware data engineering:* With a deep analysis on the existing technology based on the rapid growth of data analytics, classical data analytics platforms on low performance Python stack slowly converted to frameworks like PyTorch, Tensorflow and Chainer with the usage of C++ kernels in the core of computation and communication. Our initial literature reviews and existing research pointed out that the high performance computing with a low level system design could be the key towards improving existing systems. Our own research related to high performance data engineering [WPA+20, PAW+20, APW+20] showed that, existing data engineering frameworks in both Java and Python are not scaling well in the high performance computing environments. Furthermore, our preliminary findings on this also showed that a high performance Python approach with C++ shows better scaling than the existing state of the art systems. This shows that our approach is promising for data analytics aware data engineering.

3. *Necessity of a distributed memory oriented dataframe for HPC (MPI) on CPUs for data engineering:* We have implemented an early version of a distributed dataframe abstraction for distributed memory on CPUs. Our initial benchmarks and API definitions shows that, our proposed method is one of the most prominent distributed memory dataframe which exists at the moment. The existing parallel dataframes on CPUs are entirely written on Python which inherently impose limitations in scaling. Our preliminary research and benchmarks provides evidence[APW+20]. By designing a high performance distributed memory dataframe, we can provide better scalability and match up with the high performance data analytics workloads.

4. *Evaluate the necessity of high performance data engineering kernels to im-*

prove existing dataframe operators: We have implemented a set of widely used dataframe operators such as indexing, locate by value, unique finding, duplicate dropping and filtering. These operators are currently performing faster compared to the existing dataframe solutions. We mainly focus our attention towards Pandas, since it is the state of the art dataframe abstraction on CPUs. Our current implementations have shown that sequential performance of our kernels are promising and can be further enhanced for better performance.

5. *Usability of data engineering tools with high performance computing:* High performance Python has been used to many scientific problems to improve the performance of existing data analytical and data processing problems. But we observed that, data engineering as a research problem has not been well studied or researched with a functional data engineering system. Our research has specifically focused on the deep level of avoiding memory copying between programming kernels and user-space. Also, we have focused on retaining the performance and provide the usability at the same time. We have implemented a Cython-based middle layer of high performance Python API which allows us to avoid regular issues found in JVM-based data engineering systems and Python-based data engineering systems. Furthermore, our Cython implementations have been extended to use PyArrow and Numpy in C level via Cython bindings. This allows us to get better performance compared to existing data engineering frameworks and provide ability to seamlessly integrate with existing data analytics systems like Pytorch and Tensorflow.

6. *Research on the seamless integration of end-to-end scientific data engineering and data analytics workloads on high performance Python stack:* With a high performance distributed dataframe API and a seamless integration to numer-

ical data structures like Numpy and tensors allows us to seamlessly integrate with existing data analytics workloads. This provides us the capability to create an entire data pipeline in Python but retain high performance on the CPU stack and transfer data seamlessly to GPU stack for high performance data analytics.

BIBLIOGRAPHY

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[AFK+] Vibhatha Abeykoon, Geoffrey Fox, Minje Kim, Saliya Ekanayake, Supun Kamburugamuve, Kannan Govindarajan, Pulasthi Wickramasinghe, Niranda Perera, Chathura Widanage, Ahmet Uyar, Gurhan Gunduz, and Selahattin Akkas. Stochastic gradient descent based support vector machines training optimization on big data and hpc frameworks [accepted]. *Concurrency and Computation: Practice and Experience*.

[AFK19] Vibhatha Abeykoon, Geoffrey Fox, and Minje Kim. Performance optimization on model synchronization in parallel stochastic gradient descent based svm. In *Proceedings of the HPML Workshop in International Symposium in Cluster, Cloud, and Grid Computing, Larnaca, Cyprus*, pages 1–10, 2019.

[apaa] Apache flink - stateful computations over data streams.

[apab] Apache hadoop project.

[APW+20] Vibhatha Abeykoon, Niranda Perera, Chathura Widanage, Supun Kamburugamuve, Thejaka Amila Kanewala, Hasara Maithree, Pulasthi Wickramasinghe, Ahmet Uyar, and Geoffrey Fox. Data engineering for hpc with python. In *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 13–21. IEEE, 2020.

[AZR17] Bilal Akil, Ying Zhou, and Uwe Röhm. On the usability of hadoop mapreduce, apache spark & apache flink for data science. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 303–310. IEEE, 2017.

[Bis19] Ekaba Bisong. Google colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 59–64. Springer, 2019.

[CLJ+18] Yanzhe Cheng, Fang Cherry Liu, Shan Jing, Weijia Xu, and Duen Horng Chau. Building big data processing and visualization pipeline through apache zeppelin. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–7. 2018.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[cud] Cudf gpu dataframes.

[das] Dask framework.

[DL17] Tomasz Drabas and Denny Lee. *Learning PySpark*. Packt Publishing Ltd, 2017.

[Dona] Jack Dongara. Lapack: daxpy. http://www.netlib.org/lapack/explore-html/de/da4/group__double__blas__level1_ga8f99d6a644d3396aa32db472e0cfc91c.ht (Accessed on 06/07/2020).

[Donb] Jack Dongara. Lapack: ddot. http://www.netlib.org/lapack/explore-html/de/da4/group__double__blas__level1_ga75066c4825cb6ff1c8ec4403ef8c843a.htm (Accessed on 06/07/2020).

[Eta19] Leila Etaati. Azure databricks. In *Machine Learning with Microsoft Technologies*, pages 159–171. Springer, 2019.

[Fox17] Geoffrey 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*, UCC '17, pages 1–1, New York, NY, USA, 2017. ACM.

[GG16] B Granger and J Grout. Jupyterlab: Building blocks for interactive computing. *Slides of presentation made at SciPy*, 2016.

[Hay20] Wolfgang Hayek. Parallel computing with dask. 2020.

[ipy] ipython/ipyparallel: Interactive parallel computing in python. https://github.com/ipython/ipyparallel. (Accessed on 09/10/2020).

[IS15] Muhammad Hussain Iqbal and Tariq Rahim Soomro. Big data analysis: Apache storm perspective. *International journal of computer trends and technology*, 19(1):9–14, 2015.

[KHAL+14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[KWG+18] 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)*, volume 00, pages 383–391, Jul 2018.

[LDMG20] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11. IEEE, 2020.

[M+11] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[mod] Modin dataframes.

[num] Numpy - the fundamental package for scientific computing with python.

[PAW+20] Niranda Perera, Vibhatha Abeykoon, Chathura Widanage, Supun Kamburugamuve, Thejaka Amila Kanewala, Pulasthi Wickramas-

inghe, Ahmet Uyar, Hasara Maithree, Damitha Lenadora, and Geoffrey Fox. A fast, scalable, universal approach for distributed data reductions. *arXiv preprint arXiv:2010.14596*, 2020.

[Per18] Jeffrey M Perkel. Why jupyter is data scientists' computational notebook of choice. *Nature*, 563(7732):145–147, 2018.

[PG07] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in science & engineering*, 9(3):21–29, 2007.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[Roo20] Chat Room. Apache beam. *system*, 11(17):24, 2020.

[SDB18] Alexander Sergeev and Mike Del Balso. "horovod: fast and easy distributed deep learning in tensorflow". *arXiv preprint arXiv:1802.05799*, 2018.

[SGO+98] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[Tes16] Federico Tesser. Distributed message passing with mpi4py. In *Euroscipy 2016*, 2016.

[TOHC15] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[twi17] Twister2: Design of a big data toolkit, 2017. Technical Report.

[VdWSNI+14] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[VOS18] AARON VOSE. Interactive distributed deep learning with jupyter notebooks. 2018.

[WKG+19] Pulasthi Wickramasinghe, Supun Kamburugamuve, Kannan Govindarajan, Vibhatha Abeykoon, Chathura Widanage, Niranda Perera, Ahmet Uyar, Gurhan Gunduz, Selahattin Akkas, and Geoffrey Fox. Twister2: Tset high-performance iterative dataflow. In *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, pages 55–60. IEEE, 2019.

[WPA+20] Chathura Widanage, Niranda Perera, Vibhatha Abeykoon, Supun Kamburugamuve, Thejaka Amila Kanewala, Hasara Maithree, Pulasthi Wickramasinghe, Ahmet Uyar, Gurhan Gunduz, and Geoffrey Fox. High performance data engineering everywhere. *arXiv preprint arXiv:2007.09589*, 2020.

[WYMY+] Justin M Wozniak, Hyunseung Yoo, Jamaludin Mohd-Yusof, Bogdan Nicolae, Nicholson Collier, Jonathan Ozik, Thomas Brettin, and Rick Stevens. High-bypass learning: Automated detection of tumor cells that significantly impact drug response.

[XAB+21] Fangfang Xia, Jonathan Allen, Prasanna Balaprakash, Thomas Brettin, Cristina Garcia-Cardona, Austin Clyde, Judith Cohn, James Doroshow, Xiaotian Duan, Veronika Dubinkina, et al. A cross-study analysis of drug response prediction in cancer cell lines. *arXiv preprint arXiv:2104.08961*, 2021.

[ZKD+14] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE, 2014.

[ZXW+16] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shiv-

aram Venkataraman, Michael J Franklin, et al. "apache spark: a unified engine for big data processing". *Communications of the ACM*, 59(11):56–65, 2016.

Vibhatha Lakmal Abeykoon

EDUCATION

2016	B.Sc., Electrical and Information Engineering Faculty of Engineering, University of Ruhuna Galle, Sri Lanka

INTERNSHIPS

2019	Research Intern Argonne National Laboratory Lemont, Illinois, United States
2020	Research Intern Microsoft Redmond, Washington, United States

PUBLICATIONS

1. **Abeykoon, Vibhatha** and Fox, Geoffrey and Kim, Minje and Ekanayake, Saliya and Kamburugamuve, Supun and Govindarajan, Kannan and Wickramasinghe, Pulasthi and Perera, Niranda and Widanage, Chathura and Uyar, Ahmet and Gunduz, Gurhan and Akkas, Selahattin, *Stochastic Gradient Descent Based Support Vector Machines Training Optimization on Big Data and HPC Frameworks*, Concurrency and Computation: Practice and Experience [ACCEPTED], 2021.

2. **Abeykoon, Vibhatha** and Perera, Niranda and Widanage, Chathura and Kamburugamuve, Supun and Kanewala, Thejaka Amila and Maithree, Hasara and Wickramasinghe, Pulasthi and Uyar, Ahmet and Fox, Geoffrey, Workshop on Python for High-Performance and Scientific Computing, Supercomputing, 2020.

3. Widanage, Chathura and Perera, Niranda and **Abeykoon, Vibhatha** and Kamburugamuve, Supun and Kanewala, Thejaka Amila and Maithree, Hasara and Wickramasinghe, Pulasthi and Uyar, Ahmet and Gunduz, Gurhan and Fox, Geoffrey, *High Performance Data Engineering Everywhere*, 2020 IEEE International Conference on Smart Data Services (SMDS).

4. Perera, Niranda and **Abeykoon, Vibhatha** and Widanage, Chathura and Kamburugamuve, Supun and Kanewala, Thejaka Amila and Wickramasinghe, Pulasthi and Uyar, Ahmet and Maithree, Hasara and Lenadora, Damitha and Fox, Geoffrey, *A Fast, Scalable, Universal Approach For Distributed Data Reductions*, International Workshop on Big Data Reduction, IEEE Big Data 2020.

5. Wickramasinghe, Pulasthi and Perera, Niranda and Kamburugamuve, Supun and Govindarajan, Kannan and **Abeykoon, Vibhatha** and Widanage, Chathura and Uyar, Ahmet and Gunduz, Gurhan and Akkas, Selahattin and Fox, Geoffrey, *High-Performance Iterative Dataflow Abstractions in Twister2: TSet*, Concurrency and Computation: Practice and Experience, 2020.

6. **Abeykoon, Vibhatha** and Fox, Geoffrey and Kim, Minje, *Performance Optimization on Model Synchronization in Parallel Stochastic Gradient Descent Based SVM*, Proceedings of the HPML Workshop in International Symposium in Cluster, Cloud, and Grid Computing, Larnaca, Cyprus, 2019.

7. **Abeykoon, Vibhatha** and Liu, Zhengchun and Kettimuthu, Rajkumar and Fox, Geoffrey and Foster, Ian, *Scientific Image Restoration Anywhere*, Proceedings of Technical Consortium On High Performance Computing, Xloop, Supercomputing 2019.

8. **Abeykoon, Vibhatha** and Kamburugamuve, Supun and Govindrarajan, Kannan and Wickramasinghe, Pulasthi and Widanage, Chathura and Perera, Niranda and Uyar, Ahmet and Gunduz, Gurhan and Akkas, Selahattin and Von Laszewski, Gregor, *Proceedings of IEEE Big Data 2019, Streaming ML Workshop*, 2019.

9. Wickramasinghe, Pulasthi and Kamburugamuve, Supun and Govindarajan, Kannan and **Abeykoon, Vibhatha** and Widanage, Chathura and Perara, Niranda and Uyar, Ahmet and Gunduz, Gurhan and Akkas, Selahattin and Fox, Geoffrey, *Twister2:TSet High-Performance Iterative Dataflow*, Proceedings of the International Conference on High Performance Big Data and Intelligent Systems, Shenzhen, China, 2019.

10. Kamburugamuve, Supun and Wickramasinghe, Pulasthi and Govindarajan, Kannan and Uyar, Ahmet and Gunduz, Gurhan and **Abeykoon, Vibhatha** and Fox, Geoffrey, *Twister: Net-communication library for big data processing in HPC and cloud environments*, Proceedings of Cloud 2018 Conference, 2018.

11. Kamburugamuve, Supun and Govindarajan, Kannan and Wickramasinghe, Pulasthi and **Abeykoon, Vibhatha** and Fox, Geoffrey, *Twister2: Design of a big data toolkit*, Concurrency and Computation: Practice and Experience, e5189, 2017.

12. Govindarajan, Kannan and Kamburugamuve, Supun and Wickramasinghe, Pulasthi and **Abeykoon, Vibhatha** and Fox, Geoffrey, *Task Scheduling in*

Big Data-Review, Research Challenges, and Prospects, 2017 Ninth International Conference on Advanced Computing (ICoAC), 2017.

13. **Abeykoon, Vibhatha** and Kankanamdurage, Nishadi and Senevirathna, Anuruddha and Ranaweera, Pasika and Udawalpola, Rajitha, *Electrical Devices Identification through Power Consumption using Machine Learning Techniques*, International Journal of Simulation: Systems, Science and Technology, 2017.

14. **Abeykoon, Vibhatha** and Kankanamdurage, Nishadi and Senevirathna, Anuruddha and Ranaweera, Pasika and Udawalpola, Rajitha, *Real Time Identification of Electrical Devices through Power Consumption Pattern Detection*, Proceedings of the International Conference on Micro and Nano Technologies, Modelling and Simulation, Kuala Lumpur, Malaysia, 2016.

CONFERENCE TALKS

2020/11 Presented the paper on Data Engineering for HPC with Python (Nov 11-13, 2020) **PyHPC, Super Computing 20** [**Virtual**]

2019/12 Attended the conference and presented the paper, Streaming machine learning algorithms with big data systems (Dec 9-13, 2019) **Stream-ML, IEEE Big Data 2019** [**Los Angeles, California, United States**]

2019/11/18 Attended the conference and presented the paper, Scientific Image Restoration **Xloop, Super Computing 2019** [**Denver, Colorado, United States**]

2019/05/14 Attended the conference and presented the paper, Performance optimization on model synchronization in parallel stochastic gradient descent based SVM, **HPML, CCGRID 2019** [**Larnaca, Cyprus**]

2016/06/14 Attended the conference and presented the paper, Real-Time Electrical Device Identification with Machine Learning Techniques, **IET Present Around the Globe, Sri Lankan Chapter** [**Galle, Sri Lanka**]

OPEN SOURCE SOFTWARE DEVELOPMENT

1. Cylon: A Lead developer and researcher. [*https://cylondata.org/*]
2. Twister2: A Lead developer and researcher. [*https://twister2.org/*]
3. MLCube Applications: Contributor. [*https://mlcommons.org/en/mlcube/*]