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

Dennis Groth
Dean of the University Graduate School

Indiana University Bloomington, 2021

© Copyright 2021 by Vibhatha Lakmal Abeykoon
All rights reserved.

DEDICATION

To my parents, my loving wife and brothers and sisters.

Acknowledgements

I have been privileged to be a part of beautiful people who have helped me throughout life as a student and a human being. First of all, I would like to pay my sincere gratitude to my advisor, distinguished professor Geoffrey Fox for guiding me throughout the journey as a PhD student. Your expertise and guidance have always been a great strength to improve my research career. In the first three years of my PhD life, the PhD advisory committee, including professor Judy Qiu and Professor Minje Kim has been great mentors to improve myself to achieve goals in my PhD life. I would also like to thank Professor Prateek Sharma and Professor Ariful Azad for being my mentors to guide me throughout the rest of the journey in PhD. Their guidance and expertise were always helpful to me.

Digital Science Center staff and the research team members, including Dr Supun Kamburugamuve, Dr Saliya Ekanayake, Dr Gregor Von Laszewski Dr Jerome Mitchell, Dr Kannan Govindarajan, Chathura Widanage, Dr Ahmet Uyar, Dr Gurhan Gunduz, Dr Bo Peng, Bo Feng and Miao Jiang have been a great company to enjoy and learn a lot from each other. Also, it's a privilege to work with my fellow PhD students, Pulasthi Supun Wickramasinghe and Niranda Perera, on many research including Twister2 and Cylon research projects. Their suggestions and ideas always motivated me to improve as a researcher.

I would also like to pay my sincere gratitude to my undergraduate research mentors. Dr Keerthi Gunawickrama, Dr Rajitha Udawalpola, Dr Pasika Ranaweera, Dr Sanjeeva Witharana, Dr Sanath Jayawardena, Hiranya Nuwan Kumara, Samitha Kumara and all the research staff in the Electrical and Information Engineering Department of the University of Ruhuna, Sri Lanka, have been the first motivation towards my graduate studies.

Also, my school teachers from St. Thomas Boys' College, Matara, Sri Lanka, have been an incredible inspiration to improve myself as the student I am today. Special thanks to Chethaka Gamage, Dhanushka Kankanamge, Janitha Samarawickarama, Gihan Lakshitha, Chirantha Kanchana, Najath Akram, Asanka Ranasinghe and all my friends for being there for me from childhood up to this date.

My loving father and mother was a part of my whole life to keep me going and being there for me at all times. Their guidance was always the best thing I had to improve myself as a better person. And my brother is always like a rock to me and has always been there for me. And my loving wife, Kalani Ayodha, was always beside me, every step of the way. She kept me motivated and guided me to improve myself to become the person I am today.

Thanks to everyone who has helped me to come this far!

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. Data analytics contains a wide range of disciplines like statistical modelling, machine learning, deep learning, etc. In our research, we mainly focus on the machine learning and deep learning components of the data analytics ecosystem. 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 preprocessed 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 is 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 communication via MPI. In addition to this, designing optimized compute kernels allows for the use of HPC resources in an effective manner. 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 gives in-depth analysis on a set of data engineering operators implemented to run on HPC resources, and these operators are exposed to the user in terms of a 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 customized 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 effectively on HPC environments.

Some widely used data structures in data analytics are Numpy and Tensors. Seamless integration among data engineering data structures and data analytics data structures provides efficiency in data exploration research. Such tactics, combined with existing data structures like Pandas dataframe, enable facilitation with current data engineering programmes. This thesis investigates how the developed HPC data engineering operators perform compared to existing data engineering operators. This thesis also 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

FIGURE PAGE

CHAPTER 1

MOTIVATION

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 emphasis was on developing algorithms to model such problems in an accurate way. But with the growth of available data, these problems scaled into a much larger issue, which was not only restricted to modelling, but also processing the data efficiently. This was necessary to model vivid scientific curiosities and unknowns into a simple understandable expression. Modelling at the time relied on very accurate statistical models, which were focused on various numerical modelling methods. But these statistical models evolved into a structured data analytics domain called machine learning at the start of the 20th century. Machine learning became a very powerful tool to solve a wide variety of problems very efficiently. Since the dawn of the big data age, the data started growing rapidly, and scientists needed tools to process such datasets efficiently. This was the dawn of big data systems, which began to couple with machine learning workloads. Over time, the machine learning models evolved into deep learning models, which are highly sophisticated algorithms based on neural networks.

Many tools were developed to assist with data processing for efficient data analytics. Data exploration tools like Pandas[M+11] have become a key resource in data processing for small-scale problems. For distributed data exploration, data engineering frameworks like [das] were created on top of Pandas. Frameworks like Apache Spark[ZXW+16], Apache Flink[apaa], Apache Storm [IS15] and Apache Hadoop [apab] now exist to provide data processing for streaming and batch computations. Individually these existing tools are built to perform on specific tasks like data exploration or data processing. But the underlying core problem set is much deeper, and

it requires more involvement from distributed system researchers to build seamlessly integrated tools that can be applied in data analytics-aware data engineering.

As data analytics has grown, so too have the problems it faces based primarily on two factors: increasing data for analytics and model size. In the early days, the model size could fit into a single machine, so using data parallelism was sufficient. With the evolving amount and granularity of issues, scalability for data analytics has become a vital area of research. Frameworks like PyTorch[PGM+19], Tensorflow [ABC+16] and MxNet [CLL+15] are popular options for data analytics, while others like Horovod[SDB18] can be used to perform data analysis at scale with many frameworks in a unified manner. Integrating these tools with data processing is essential to build scientific data pipelines.

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

volves a few options to do data engineering sequentially or in parallel. Frameworks like Pandas provide a definition to represent tabular data and preprocess them with basic dataframe operations widely used by data engineers. Also, frameworks like Modin and Dask were created to scale Pandas on a CPU stack. These frameworks are entirely written in Python and focus on a client-server-based distributed model to support data engineering, and are not entirely focused on a classic HPC software stack to provide efficient kernels for distributed computation. But they are easy to handle and provide users the ability to scale existing Pandas workloads on CPUs. Besides CPUs, frameworks like cuDF were 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 does not posses a BSP mode of execution for data engineering and relies on the classic client-server architecture to scale the dataframe-based workloads. We believe we can design a high performance dataframe which suits the BSP execution and 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. Having a BSP model based on HPC software stack also provides the ability to seamlessly integrate with existing data analytics workloads specifically designed to run on HPC hardware.

Our research focuses on understanding the importance of high performance data analytics and analyzes 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 aim to solve this key problem with a subset of in-depth analyses on the importance of high performance data analytics, efficient and effec-

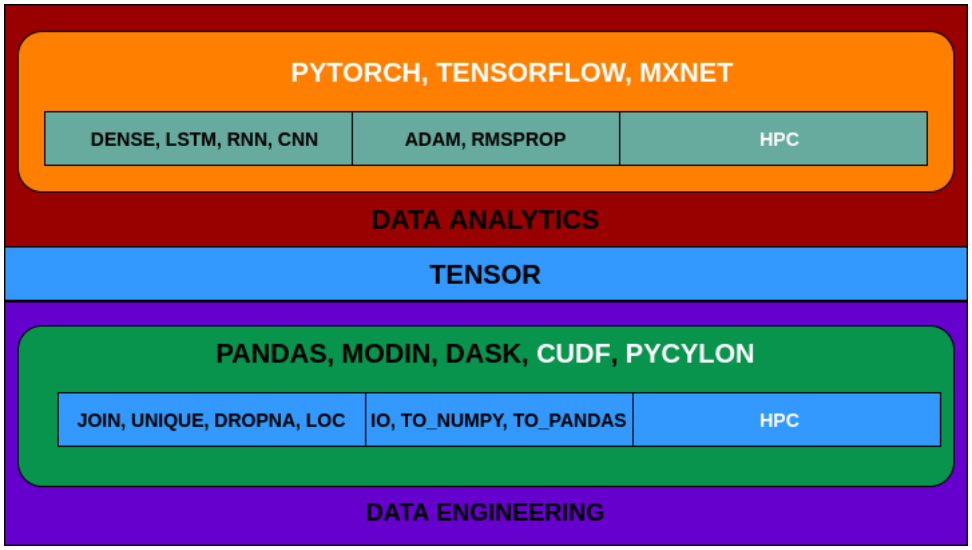

Figure 1.1: Systems overview for data analytics aware data engineering

tive data engineering, and usability for seamless integration with the existing data analytics subsystems.

Our research goals are 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 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 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 workloads depend on structured data with expected data formats and types, which are then ingested by vivid analytical platforms to provide intelligence and harness important information. The majority of these applications rely on tabular data, which is 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 are 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 classic 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 major big data systems designed to preprocess the data for most industrial applications. The main programming languages used to build these frameworks are Java, Scala and Python. They perform very well in cloud environments. These systems tend to be designed for high throughput and scalability. One drawback is their lack of any 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, as well as communication kernels like MPI[SGO+98], PGAS[ZKD+14] and HPX[KHAL+14]. This is crucial owing to 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 one of the earliest precursors, 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 analytics 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 became easy. Pandas also supports Numpy[num] which is a 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 it.

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

The Python research community adopted 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 do not scale well when compared to others 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. 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 research efficiency. In old-school scientific research, the most important tool might be 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 do not provide a better usability for distributed computing. There are existing commercial products offering support on cloud environments, but there is a lack of usability in the existing open source frameworks like IPyParallel[VOS18] when dealing with the data representation in high performance computing environments. Figure 2.1 shows a higher level view of data analytics-aware data engineering.

We believe that data engineering on CPU stacks 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 the Cylon[WPA+20, PAW+20] data engineering system we designed. With this system, our focus is to integrate high performance and high usability in data engineering.

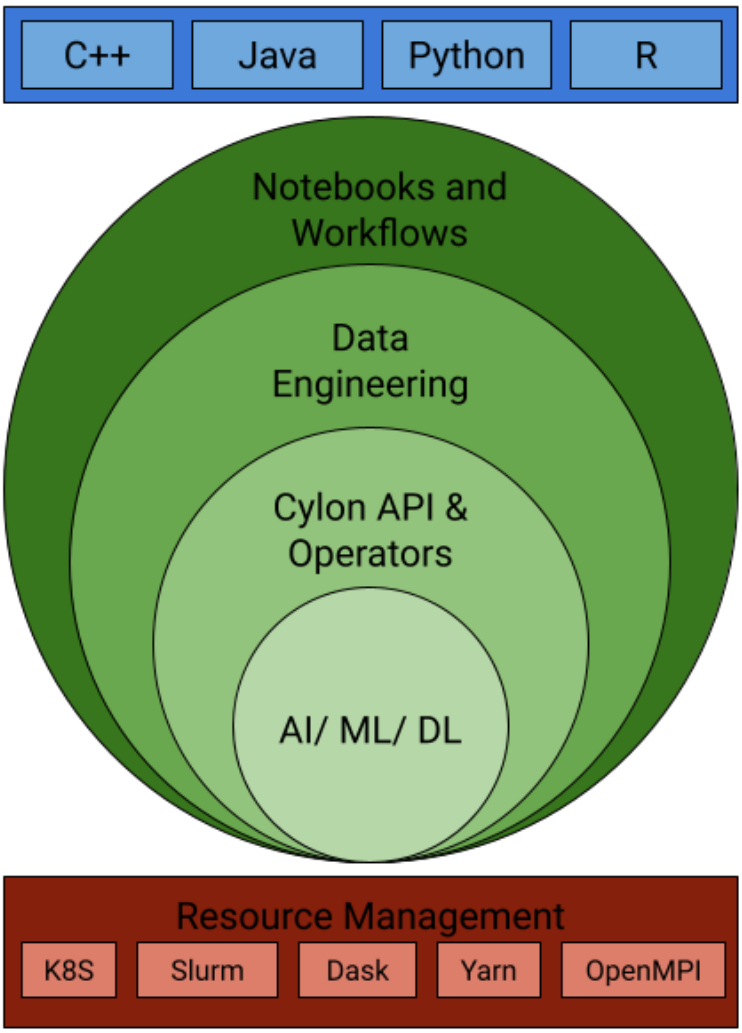

Figure 2.1: Higher Level View of Data Analytics-Aware Data Engineering

CHAPTER 3

LITERATURE REVIEW

The first big data systems were a breakthrough in data engineering. 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] are recognized as examples of such big data systems capable of data engineering. They support both batch and stream data computation on the distributed computing paradigm. Apart from the big data systems, the HPC community from academia created frameworks like MPI[SGO+98], PGAS[ZKD+14] and HPX[KHAL+14] running in supercomputing environments. These are specialized models with high performance compute kernels for math-based and distributed memory computations. HPC systems are mostly favourable for compute-intensive workloads with basic compute and communication kernels. The major difference between big data and HPC systems is the way they are designed and the tasks they are specialized to perform. For a better understanding of the general big data use case, it is vital to examine the core values of both systems and design a hybrid system which can do both. Big data systems are easy to use and offer a variety of compute kernels abstracted by layers of application programming interfaces (APIs) and allow easy access for users to design systems. On the other hand, big data systems only provide the major compute kernels and communication kernels to build such APIs. But HPC systems are much faster in most cases. To overcome this, 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 fashion.

Since developing that system, we have been closely analysing its capabilities and limitations when it comes to computation-intensive applications from modern data-related instances. Data science wrapped in machine learning and deep learning are one such example which require a special set of requirements. These analytical problems consist of two major aspects: an efficient system is required to do the computation, and an effective system is needed to model the problem. The accelerated data processing inherently becomes an HPC issue, and existing knowledge can be extended towards designing an efficient system. Considering the effectiveness, 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 the nature of data, data scientists and engineers 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 by big data systems, and the effect is enhanced with the Python programming layer added on top of existing big data systems. These three characteristics allow for the capability to efficiently and effectively prototype a scientific analysis and design the end system for production. We came upon this simple pattern, which has already 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 became a solution to data engineering problems to a certain extent.

However, we observed a set of major drawbacks in the existing systems. For big data systems with Python, it is the massive serialization-deserialization cost when data is moved back and forth from Python to Java. Since core compute 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 have to be continuously serialized and deserialized. In addition, 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 regular compute kernels on eager execution. Furthermore, the learning curve for maintaining 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 most prominent tools used in data science. In addition, Cudf is a high performance GPU dataframe which executes sequentially. Similar to Dask scaling Pandas, CuDf is also using Dask to scale[HSY+20]. In contrast, the big systems contain a complete eco-system to perform various data engineering tasks. These frameworks are designed to process larger data sets by partitioning the data and processing sub tasks by executing them remotely in computing resources. Among the data engineering frameworks, Dask provides a distributed dataframe for Pandas by having a global view as a dataframe and having partitions of dataframe across multiple tasks. Dask distributed execution format is based on a client-server execution architecture where a scheduler is dealing with the workload and scheduling tasks by specifying the number of partitions specified by the user. This execution model is the same model used Apache Spark for distributed computation. Apache Spark eco-system contains various data abstractions like Spark-RDD, Dataset and Dataframe. But the usability of Apache Spark is more complex than using Dask to do data processing in parallel. The limitation of this model is the centralized nature

since the scheduler is dealing with scheduling tasks. This execution mode is prone to performance issues in communication and overheads in scheduling tasks. Besides, this execution model doesn't fit well with the BSP execution model in HPC-based applications. Modin, another framework focusing on accelerated data engineering for Pandas also provide an easy mode of usage for the user. Underneath it uses Ray to parallelize the tasks based on a actor-based model. This is another remote-tasks based model. Eventhough in terms of execution it is a bit different from the client server model, the actor based execution is also not a natural paralle model to scale application in HPC resources. Another limitation in these systems is the less usage of high performance kernels. Pandas itself contains a set of high performance Cython kernels to improve the performances of joins and other time consuming operations. But both Dask and Modin doesn't such kernels to improve the local operations. For time consuming operations like Joins, duplicate handling and search operations, having high performance kernels can provide better performance over Pythonic algorithm implementations in most of the data engineering frameworks like Dask or Modin. Besides, the Apache Spark has these kernels written in Scala or Java which could be implemented for better performance using C++. These Python systems are highly effective in designing, but suffer from performance issues mainly owing to their having compute 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 results over classic Python systems. This reveals a very significant point in designing better systems.

Although most of the data analytics aware data engineering systems are focusing on the in-memory distributed computations for data exploration based applications, the database community used distributed computation for database queries for a

very longer time. The difference between the database systems and the data exploration systems like Pandas, Dask, Modin or PyCylon is that the operators are not persistent and not fault tolerant. For instance, distributed database systems like CockroachDB highly fault-tolerant and works well even in worse conditions [TSM+20]. Dask and Modin has some integration with the disks, but these features are not widely used for data engineering applications since, most of the workloads are focusing on the in-memory data exploration approach rather than querying data by writing SQL queries. What database systems offer and what data exploration data engineering systems offer are completely different in terms of user experience. But it is a clear fact that, PyCylon and other systems can use existing algorithms optimized by the database commmunity to create efficient implementations for some of the database operations like select, join, filter, etc. Besides, features like replication, transcation-based execution are not commonly seen in in-memory exploration tools like Pandas. These features are important for applications which are motivated by data persistence and accuracy in atomic level. But for data analytics applications, what current data engineering offers is adequate and matching with the requirements. These are also limitations of data engineering systems, but expected outcome as far as data analytis aware data engineering considered.

When we discuss about systems in terms of just doing data processing, in terms of a distributed system the main attention is paid to systems like Databrick's Spark because of the broad eco-system it supports. Databrick's Spark version is a commercial version of Apache Spark scaling data processing operations in cloud environments. The programming model of these applications can be in-memory or utilize the disk depending on the problem size the limitations in the hardware. The data analytics aware data engineering frameworks and these big data systems mainly differ in terms of the commercial usage support, third-party library support for scaling

in clouds and data procesing model. Big data systems like Apache Spark focus on the entire data processing stack starting from stream processing to batch processing with vivid data structures. Overall, such big data systems provide a solution to a much larger problem. But as far as data analytics data engineering is considered the qualities like JVM-oriented design, non-BSP computation model, in-efficient Python bindings and complex APIs takes away the productivity from the data scientists. The main objective of systems like Pandas, Modin and Dask is to provide the performance and easy way to write data processing for data analytics applications. So the main difference is the domains and the set of problems focused by each framework. Apache Spark can solve all these problems, unlike the data analytics specific data engineering systems, but not very well designed to meet the specific requirements like high performance, easy programming model and adaptability with less system knowledge. Such qualities dilutes down the efficiency and effectiveness of the systems.

One aspect of the motivation behind an efficient and effective data engineering system is expanding intensive data analytical workloads. Today's workloads are mainly focused on machine learning and deep learning approaches. They are composed of layers and layers of classic data analytical kernels focused on extracting as much useful information as possible from the raw data processed by data engineering systems. These analytical systems are based on two principles, namely 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 a high performance factor, deep learning frameworks like PyTorch[PGM+19], Tensorflow[ABC+16], MxNet[CLL+15]

and Chainer[TOHC15] made available high performance compute kernels written in C/C++ and exposed the kernels via efficient Python bindings. A major part of the computation workload runs 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 they are limited to a math-based data structure called tensors. Tensors are the form in which data is injected into these data analytic 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 a scientist might be working on evaluating the feature extraction-based analytical model convergence, it is crucial to keep an efficient data pipeline when processing large datasets. Even for the disk-based approach, we have to store them in tensor-compatible data structures rather than other types. Focusing on data structures, the big data systems are in favour of structured data in tabular format. In modern 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 element. Having an efficient and effective methodology for this conversion must be achieved when 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 difficult to visualize the in-

termediate data structures in distributed memory. A medium which is acceptable and widely used by scientists must also be on hand 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 practice, interactive notebooks like Jupyter notebooks[KRKP+16, GG16] have been widely used by many scientists. To provide enhancements for usability like Python, Java, C++, Scala, Ruby and Julia have been added to such notebooks by scientific research communities. Many of the industrial research communities have extended this to support various requirements. Netflix has develped an open source version of Scala-driven interactive notebooks called Polynote [LDMG20] to match their industrial needs. Apache Zeppelin[CLJ+18] is another such open source project 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 in response. It supports both MPI models and task-based models. Dask also provides their notebook [Hay20] extending from the IPyParallel kernels. Still, the main issue is that 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.

CHAPTER 4

DISTRIBUTED MACHINE LEARNING

This chapter gives an overview of integrating HPCs for distributed machine learning algorithms. Here we will discuss how distributed machine learning algorithms can be efficiently implemented for better scalability and usability with HPC-based big data analytics. The objective of this study is to showcase the importance of high performance computing for data analytics and how this could be very valuable for much more complex problems. We briefly discuss two machine learning algorithms implemented in batch and streaming mode execution on an HPC-based implementation and hybrid big data/HPC implementation. We illustrate how HPC solutions outperform traditional big data solutions designed for data analytics with big data. Our findings further reinforce the necessity of optimized data analytics systems for complex data analytics and the need to move data engineering towards this goal.

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 science models. They ran on hundreds of machines with high performance capabilities, eventually leading to HPC. But later on, the trend of analysing data moved towards big data systems with 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 traditional 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 HPC hardware. Internally Twister2:Net [KWG+18] uses MPI point-to-point communication to build a communication abstraction possessing state-of-the-art collective communication used in HPC. This provides the ability to incorporate the classic HPC communication model into big data by bridging the dataflow model with HPC communication. Pursuing this novel approach, we developed a set of applications and advanced programming models to fit well with 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) are one of the most prominent machine learning algorithms used for classification prior to the deep learning models which now dominate artificial intelligence applications. With larger datasets, SVM requires more computing resources to train efficiently. There are multiple implementations which provide distributed computation for SVM. Among these, Apache Spark and MPI-based implementations are dominant. Since our focus is to improve the performance and retain the big data attributes in programming, we applied a distributed version of SVM on Twister2.

4.1.1 Anatomy of the SVM Algorithm

There are a few optimizations algorithms often found in SVM. Sequential minimal optimization, chunking algorithm and gradient descent (GD) are some of these variations. Recently, the use of gradient descent has been widely considered with the growth of deep learning algorithms. We selected a gradient descent-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.

4.1.2 Parallel Gradient Descent SVM

We used a parallel gradient descent SVM algorithm based on the sequential version. After the completion of each epoch, a model synchronization is performed by doing an MPI_Allreduce call. Here the model weights across each process are aggregated

Algorithm 1 Gradient Descent SVM

1: $INPUT : [x, y] \in S, w \in R^d, t \in R^+, b \in Z^+$
2: $OUTPUT : w \in R^d$
3: **procedure** GRADIENT DESCENT(S, w, t, b)
4: **for** $i = 0$ to n with step size b **do**
5: **if** $(g(w; (x_i, y_i)) == 0)$ **then**
6: $\nabla J^t = w$
7: **else**
8: $\nabla J^t = w - Cx_i y_i$
9: $w = w - \alpha \nabla J^t$
return w

and averaged over the number of processes involved. Algorithm 2 is the parallel algorithm implemented on the sequential algorithm in 1. Here K refers to the number of processes, S_i indicates the i^{th} batch and T is 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

To determine the performance of the algorithm in terms of data sparsity, we chose three datasets: number of features, training data size and testing data size. 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

For 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]. Equations 4.6, 4.7 and 4.8 refer to *daxpy* BLAS operation which perform constant times a vector plus a vector [Dona]. Additionally, the inc_x and inc_y refer to the storage space between the elements in the x and y arguments of the *daxpy* notation where x and y refer 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 advanced computing engines specialized for distributed computing. We designed two sets of experiments discussing the performance of distributed SVM. The first was set up to compare the performance of implementations on Java and C++ along with BLAS integration. The second set compared the performance of big data systems, MPI systems against big data and MPI hybrid systems, and Twister2. The experiments were performed in 16 nodes of Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz and the maximum number of processes per node was set to 16.

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

Figure 4.2 shows the experiments conducted on the second set. These experiments were meant to evaluate the performance of a distributed SVM algorithm implemented with Apache Spark (as a big data system), MPI (as an HPC system) and Twister2 (a hybrid big data and HPC system). The objective was to emphasise the importance of integrating machine learning algorithms with HPC and big data hybrid systems compared to classic big data systems. These results show that the performance of Twister2 is very similar compared to the implementation on MPI. Also, Twister2 outperforms Spark-based implementations at scale.

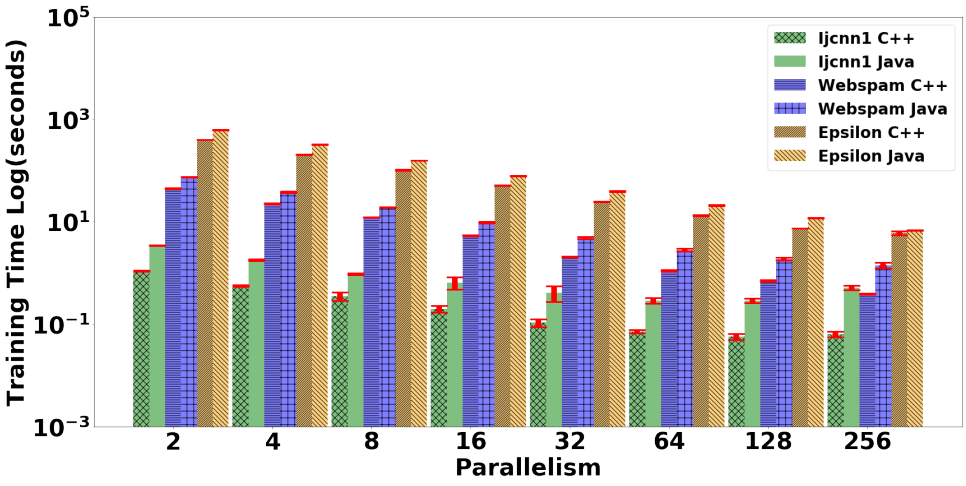

Figure 4.1: SVM Distributed Data Parallel Training with BLAS Optimizations with MPI

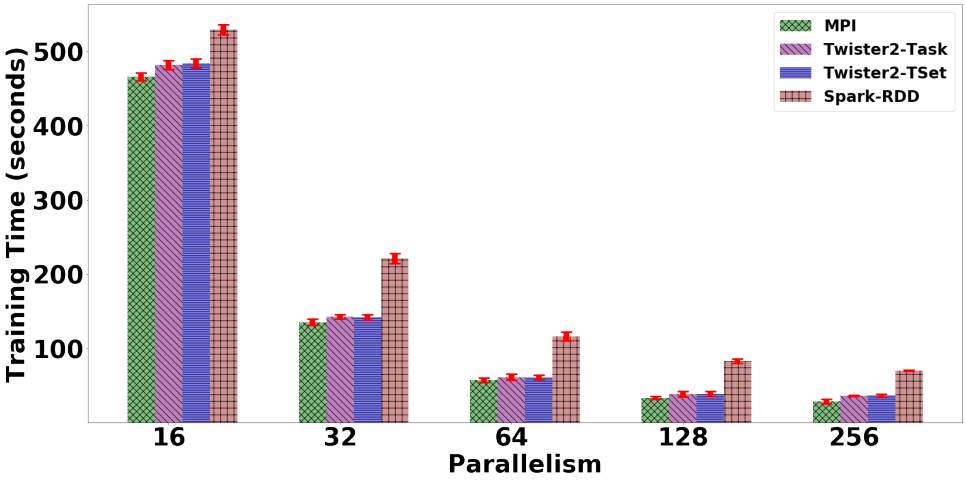

Figure 4.2: SVM Distributed Data Parallel Training with BLAS Optimizations with Big Data HPC Overlap

4.2 Iterative Streaming for Data Analytics

The impact of big data processing is not only limited to batch data, but also 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 performed on batch applications for data analytics. When we consider streaming applications, one way is to just compute a given data point once, create a state, and use it for the next data point. But when it comes to accuracy and various computational requirements, sometimes the streaming data can be converted into mini-batches and computed iteratively. This is the simple idea behind 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 are gathered by means of a windowing schema. Windowing schemas 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 a number of elements in the window or time taken to acquire elements in 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 for 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 train machine learning algorithms online.

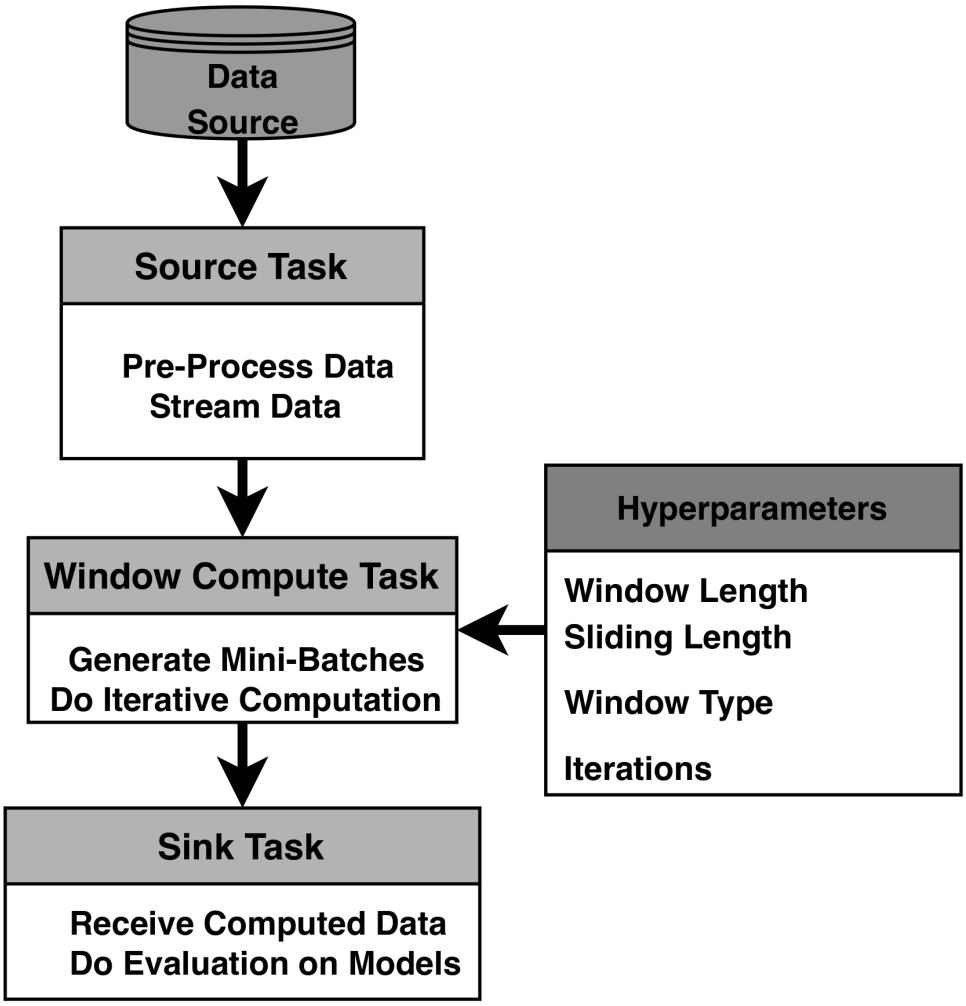

Figure 4.3: Twister2 Iterative Streaming Workflow for an ML Application

Initially the data are loaded from a source, which can be a messaging queue or a stream of data coming from storage. The source task does the preprocessing required to formulate the expected features in 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 hyper-parameters. They are window length, sliding length, window type 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 a 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 window and do an iterative computation on each. 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.

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

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

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

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 is 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 applied 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 utilized a basic version of the online KMeans algorithm. V refers to the cluster centroids, k to the number of centroids, and n 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. We determine the new centroid as shown in the algorithm. But in the initialization step, the centroids can be either handpicked from the dataset or randomly selected. Here we choose 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 most sophisticated 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.

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

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

the all-reduce model. Apache Flink does not support an all-reduce-like communication for synchronizing models globally. In Apache Spark, reduce function and RDD broadcast are used to synchronize the model. With Apache Storm, all-grouping is able to generate a synchronized model. Twister2-HPC model uses MPI-AllReduce collective communication to achieve the same result. Twister2-Dataflow model employs 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 used a distributed cluster with 8 physical nodes. We scheduled 16 tasks per node to run the experiments. Each node consisted 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 were used. For the experiments, we only had a

finite stream to evaluate training accuracy and performance. The implementations for the performance evaluation were Apache Storm 1.2.8, Apache Flink 1.9.0 and Twister 0.3.0.

Streaming SVM

For streaming SVM model, we selected a dataset with two classes and 22 elements per data point. We used an iterative computation on windowed elements for the experiments. This operation was 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 the aforementioned frameworks. The main constraint was that it only provided windowing considering the notion of time. This made it hard to do a stress test on the stream engine because, by the notion of time, the minimum number of elements that could be set per batch was in millisecond level. Furthermore, it did not 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 only considers the notion of time. We did not implement that model in this research as it is a very different case concerning the other implementations. In the Conclusion section, this will be explained in detail.

Figure 4.4 shows the experiment results for tumbling window. From these, it is clear that the Twister2 model outperforms both Apache Storm and Apache Flink implementations. Figure 4.5 is the sliding window-related experiments. Similar to tumbling windowing, with sliding windows, Twister2 implementations outperform Apache Flink and Apache Storm. Twister2 possesses a faster stream processing

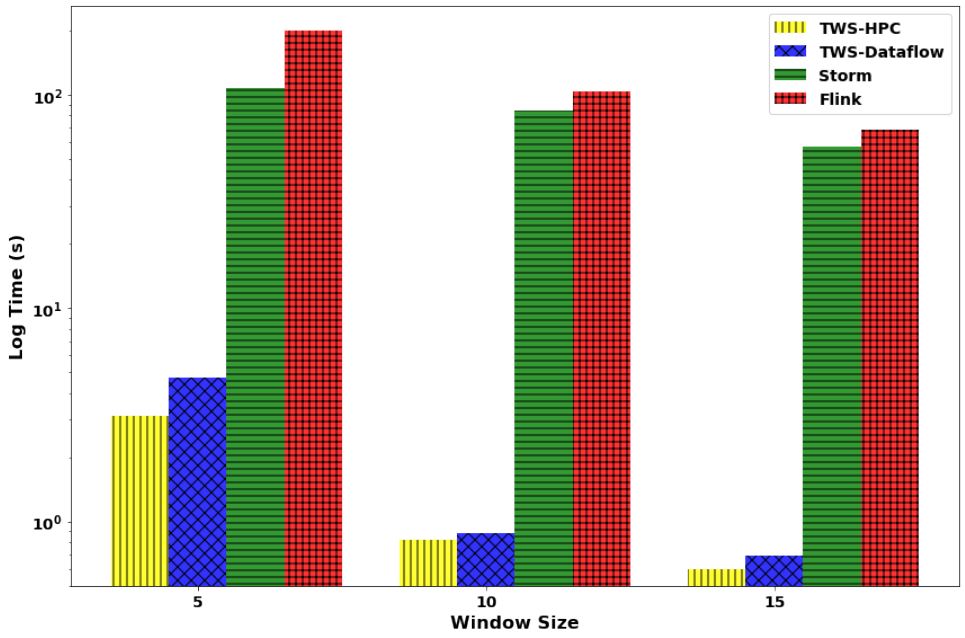

Figure 4.4: Streaming SVM with Linear Kernel-based experiments for tumbling window is recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

capability through a strong MPI-based backend. This provides a scalable solution for iterative stream processing on a window. With Apache Flink, the main bottleneck is the reduce task doing the model synchronization. In Twister2 and Apache Storm, the all-reduce and all-grouping mechanisms are involved in providing all-to-all 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 exceed Apache Flink performance.

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 do not include the graphs here because the number of iterations required to get the same accuracy is much higher.

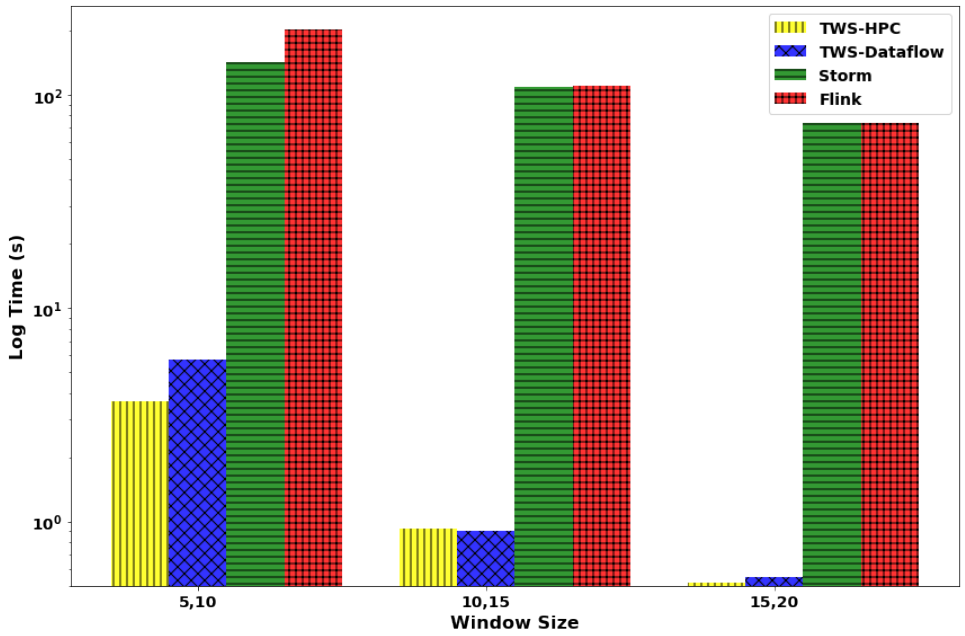

Figure 4.5: Streaming SVM with Linear Kernel-based experiments for sliding window is recorded for HPC model and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x-axis in the right figure is labeled with the pair of (window length, sliding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

The main issue for this is Spark streaming API is not designed with iteration compatibility. Also, it does not provide a window function to capture the elements belonging to a window. This functionality is available in Apache Storm, Apache Flink and Twister2. Apache Spark only provides basic element-wise operators like map, flatmap, etc. If this was attempted with a forEachRDD function, the user would have no capability to synchronize the model as it is a sink function. In addition, Spark only provides a windowing functionality with the notion of time and 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 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 illustrates the tumbling window-based experiments carried out on streaming KMeans model, while Figure 4.7 details the sliding window-based experiments carried out on streaming KMeans model. Similar to the streaming SVM results, Twister2 models yield better results than 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 appli-

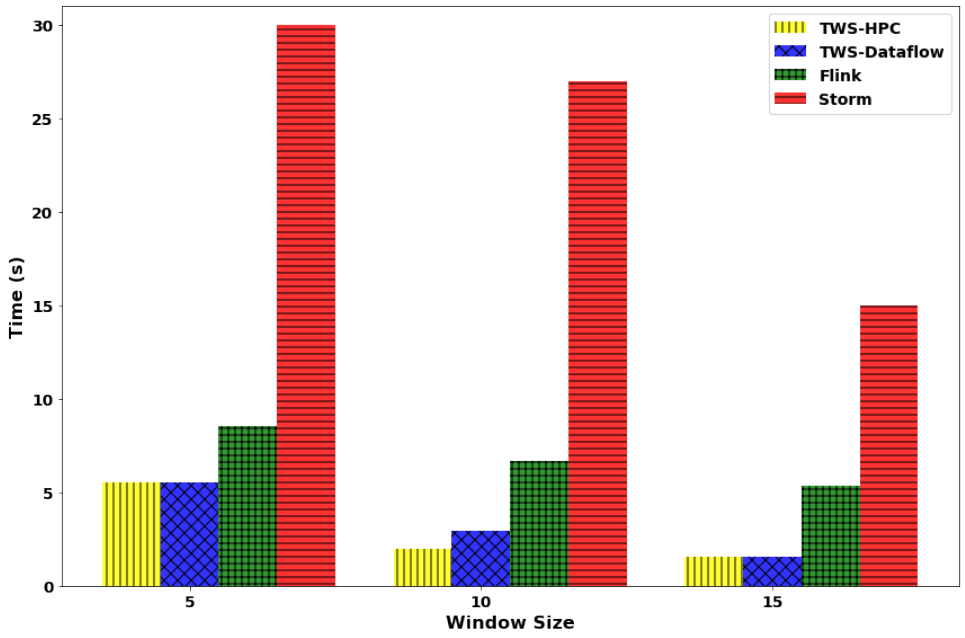

Figure 4.6: Streaming KMeans Results for 1000 cluster-based experiments for tumbling window recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence.

cation. But Apache Flink outperforms Apache Storm. This model is non-iterative and the pressure exerted on communication is less. This leads to quite faster data progress from the windowing task to the reduce task.

Both the algorithms implemented in batch and streaming execution model show that the HPC-based implementations are much better as far as performance is concerned. Furthermore, these experiments confirm that the classic big data approach for data analytics is not very suitable for complex data analytics workloads like deep learning. The more complex the algorithm, the more computational resources required. With most machine learning applications being converted to deep learning applications, less and less usage of big data systems for data analytics is beginning to occur. But it is clear that by simply evolving with HPC-based computation models, the performance can be improved. The research value of these experiment

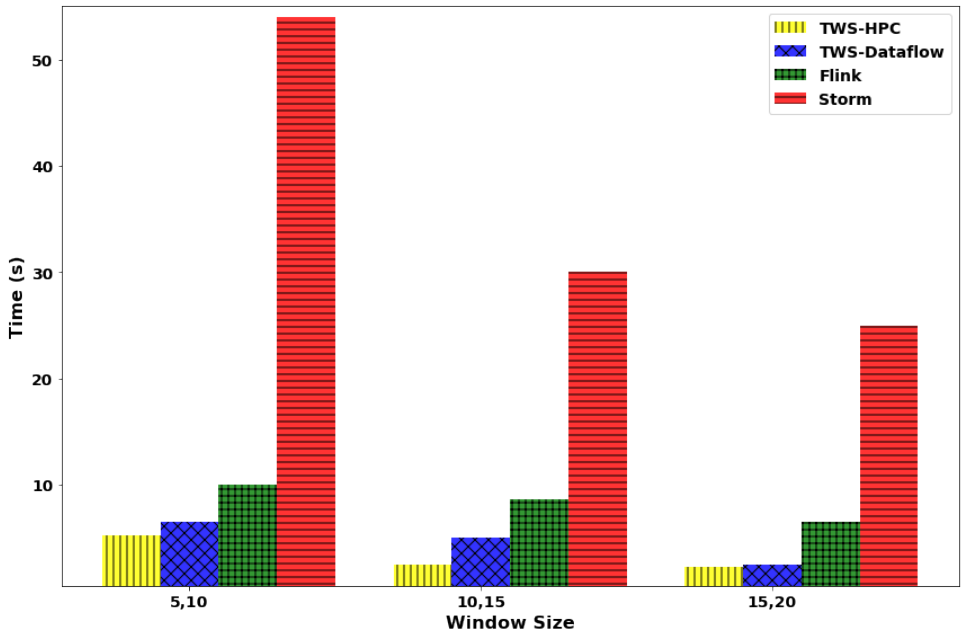

Figure 4.7: Streaming KMeans for 1000 cluster-based experiments for sliding window recorded for both HPC and Dataflow programming models. The time recorded is the streaming training time until expected convergence. The x-axis in the right figure is labeled with the pair of (window length, sliding length).

sets shows us the need to move data analytics stacks on big data for better performance with the evolving deep learning systems. In terms of big data systems and deep learning systems, the mode of execution, ease of programming, and flexibility are much favored in deep learning systems along with higher performance. Most of these algorithms for big data systems and big data-HPC systems are written in Java or Scala. These programming models are not very intuitive and the programming knowledge required to write such data engineering and data analytical workloads is far higher when it comes to complex applications. The mode of execution is lazy and involves complex task graphs and unnecessary additional details in deploying programs. Even though Python bindings are available, they are not as intuitive as Pandas. For these reasons, most of the big data systems have become obsolete as far as data analytics-aware workloads go. More and more researchers have focused on data engineering frameworks specifically designed for data analytics workloads. This brings us to the main theme of our research: data analytics-aware high performance data engineering.

CHAPTER 5

HIGH PERFORMANCE DATA ANALYTICS-AWARE DATA ENGINEERING

Recognizing the importance of data analytics and 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 data analytics workloads can be efficiently executed in large scale to train vivid scientific models to a far greater extant than without. Another important aspect that we chose not to analyze in-depth is the data engineering operations that are being widely standardized and used in parallel with evolved data analytics workloads. Even though big data systems played a major role in data analytics in the better half of the last decade, more systems took over data analytics by specializing into sub-domains, providing not just distributed communication, but also application development capability. This is accomplished by writing much fewer lines of code. In addition, these data analytics frameworks were built in such a way that data scientists can easily write programs and analyze the data very efficiently. Frameworks like PyTorch [PGM+19] and Tensorflow [ABC+16] are examples of such dominant frameworks specialized for data analytics. Also, the user base, scientific applications and research efforts grew exponentially utilising these data analytics frameworks to implement problems rather than just traditional big data systems. With the emergence of these frameworks, it is clear that the best option is to provide improved support for the data engineering portion of data analytics, and is more important than building data analytics components with big data systems. Taking all this into consideration, we dive deep into investigating 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 essential to understanding 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 available tools with much easier programming abstractions for improved application development. This is because most 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 available on top of the JVM-based big data systems to provide this usability. But using existing systems is a very time-consuming exercise for data scientists to preprocess the data. Additionally, these APIs are not seamlessly integrated to support HPC-oriented data science systems and thus 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 fashion data science applications of a higher caliber.

In the modern data engineering domain, most Python-based data engineering systems are developed by considering a data abstraction called a dataframe. Dataframes are simply 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 helpful to support a wide range of data engineering applications. Pandas[M+11] dataframe can be considered as a state-of-the-art dataframe representation in Python-oriented data engineering. Pandas only support data processing in a single process at the present. There have been many efforts from the Python community to provide distributed computing for dataframes. This abstraction has been adopted by many

distributed data engineering frameworks like Modin[mod] (structured Pandas for parallel computing), Dask[das] (distributed Pandas), and cuDF[cud] (dataframe for GPUs). Our data engineering interface mimics a high-end dataframe and extends towards an HPC-compatible computation to provide both distributed and pleasingly parallel operators.

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

5.1 Methodology

To support data analytic applications that are rapidly evolving on Deep Learning, a 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 the data analytic engine (DA). For a better data engineering design, it is necessary to understand the requirements of DA frameworks and backtrack to create efficient solutions, an approach drawn from existing applications. Also, we expand the requirements for designing the DE solutions by understanding future demands based on 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 vital components. Figure 5.1 outlines the data

movement from a data source towards a data analytic workload. 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, we must take into account their data loading components.

Data engineering can be viewed under three criteria, namely data extraction, transformation, and loading. The 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 data sets, efficient data reading and data movement in distributed computing environments cannot be overstated. Data in this phase are heterogeneous, where both numerical and non-numerical data are 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 them into an efficient in-memory format where it can be transformed with ease.

In the scope of this research, the main focus is to accelerate the CPU workloads on data preprocessing and data movement. Preprocessing can be defined into two categories. First is the raw data processing to extract features. The second category is numerical data augmentation done to reshape and transform the data into data analytic models. Here some of the data transformation kernels are known to run much faster in GPUs. But the limitation of 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 developed systems, high quality scientific applications and workloads are benchmarked. In the initial phase of the benchmarks, we micro-benchmark system-level performance compared to the exist-

ing systems. For evaluating an end-to-end workload, we benchmark state-of-the-art scientific workloads written on the proposed system.

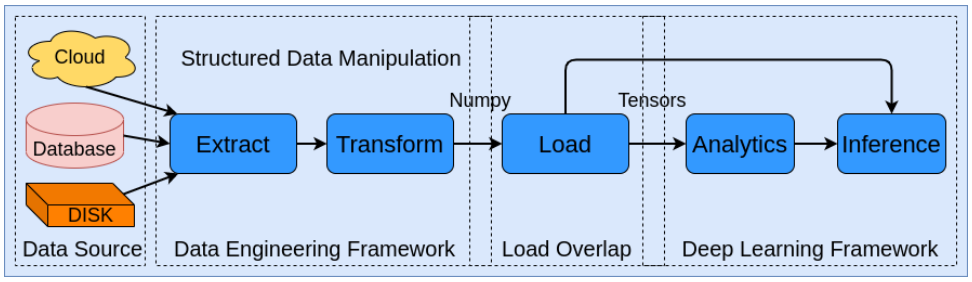

Figure 5.1: Data analytics-aware data engineering workload

5.2 System Architecture

Figure 5.2 refers to the high-level architecture of our 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 offer access 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 often written with a Python user interface. On top of the language bindings, the usability APIs and sub-algorithms are developed. DataFrame API is the most important feature, providing high performance data engineering. The DataFlow API is the gateway towards external systems like machine learning and deep learning, allowing data movement from a data engineering workload to a data analytics workload. The highest level of abstraction is focused on an annotated Python API which allows users to write distributed or sequential code without consideration for 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 classic bulk-synchronous-parallel (BSP) model, in some advance applications the user needs to handle parallelism-aware local computations by dealing with the rank or process ID.

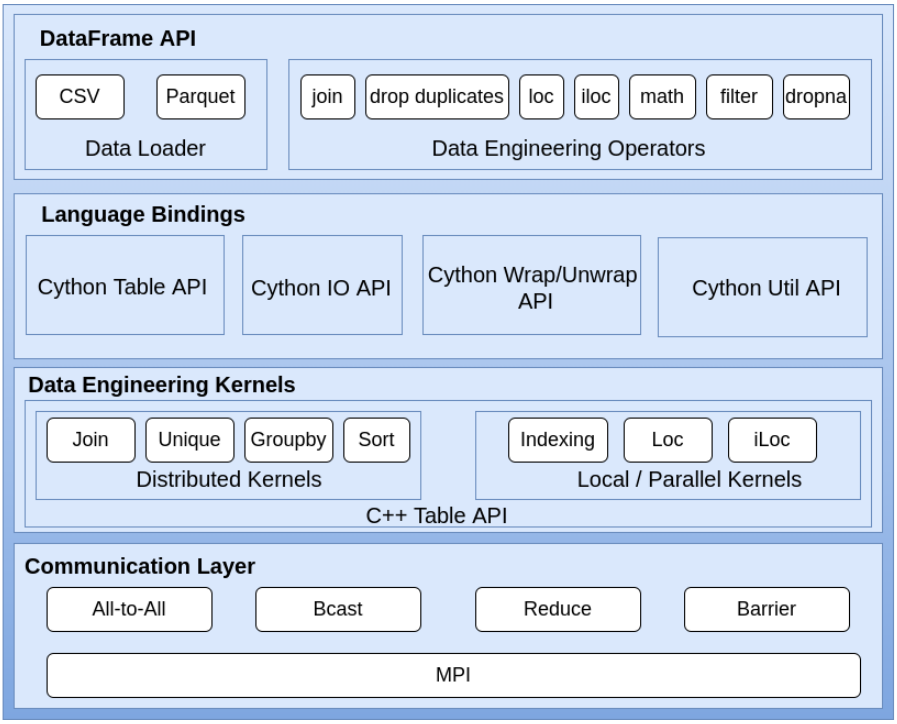

Figure 5.2: System Architecture

These data engineering operators are built in the Cylon framework. Below are the key contributions from our research therein.

- 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

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 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 of the data set. Join requires a specific column to do the join, and the data type of that column and the value of the data are needed to perform the required computation to join two tables. In the distributed setting, when comparing such values, hashing becomes a very prominent technique, which allows the moving of values with the same hash to a single machine so as to perform 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 categories of data engineering kernels supported in the system.

Kernel	Operation
Relational Algebra Kernel	Join, Union, Intersect, Difference and Project kernels
Indexing Kernels	Hash, Vector and Range indexing kernels
Search Kernels	Hash, Vector 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 three 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 fall back to local computation when used in a $parallelism = 1$ setting.

Operator	Local	Data Parallel	Pleasingly Parallel
Relational Algebra	Yes	Yes	Yes
Indexing	Yes	Not Implemented	Yes
Search	Yes	Yes (simple)	Yes
Filter	Yes	Yes (simple)	Yes
Duplicate Handling	Yes	Yes	Yes
Null Handling	Yes	Yes (simple)	Yes
Linear Algebra	Yes	Not Implemented	Yes

5.4.1 Relational Algebra Kernel

When considering dataframes, the main data relational algebra kernel used is the join operation. In the dataframe domain, the join operation requires a set of parameters from the user: join columns, join type, join prefixes (optional) and join algorithm (optional). Join prefixes allow for readability when visualizing the join outputs. Join type falls under these 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.

Join algorithm is an additional feature supported from our implementation to enable users 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) by keeping them in a hashmap, scanning through the other relation join-column, and computing the hash to build the resultant table by comparing hashes. In a distributed setting, before performing the join, we do hash-based data shuffling. The hashes for join columns are computed and data is reshuffled 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.

5.4.2 Indexing Kernel

Indexing kernels support fast data querying. We have implemented three 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 indices.

In the vector indexing implementation, a column is selected and dropped off the table depending on a user argument. The idea is that this particular column can be a data column and index, or it can just be a column 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 hashed in such a way that hashes with the same value map a set of rows. Compared to vector indexing implementation, 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 does not exist in actual memory until a user wants to get index values. It only records the starting index and end index and generates the index as per the user's requirements 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 search happens by considering a local index in each process. For distributed operations, at present the user is required to re-implement the indices after the distributed computation. When using vector indices, the time taken to generate them is negligible. We will discuss more on performance in the benchmark Section 6.1.

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

5.4.3 Search Kernel

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

- Search by value range (start value and end value): This retrieves a set of records beginning from the start value and terminating at the 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 the search table, the user can specify a range of columns or a list as well. These combinations altogether give six 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 every 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 can be a table with multiple columns or a single column table. There are many ways to generate this filter table. A boolean-value table can be generated by comparing a table or subset of table values to given values. 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
- Less than
- Less than or equal

- Equal
- Not Equal

In implementing these comparator operators, we have applied two types of implementations to support vector operations. Since we use Arrow-tabular format to represent the data internally, Arrow compute operations provide the vectorized filters rather than implementing them from scratch. Also we have included Numpy kernels internally to support optimized vector operations for comparators. We 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 to be a pleasingly parallel operator. Each process will provide the filtering operator along with the filter value. The value can be different for each process or it can be the same as far as a BSP program is concerned.

5.4.5 Duplicate Handling Kernel

Duplicate records can be widely located in a raw dataset. The objective of the duplicate handling depends on a keep policy, namely retaining the first value found as a duplicate in records or the last value found. This implementation consists of a hashset where we iterate through each row and create a hash. When new rows are inserted into the hashset, a row comparator is used to identify whether the value is already present or not. This provides the ability to differentiate between a unique record and a duplicate. In default setting, all the columns are considered when searching for a duplicate record, but we have also provided the option to include a subset of columns to determine 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 them with meaningful values is a useful function. Also, depending on data representation, null values can be represented as a string or a 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, 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 chose based on the existence of null values row-wise and column-wise based on whether there were any null values or all null values. For column-wise operations, we checked for nulls and did an algebraic sum of boolean values to determine whether any or all values were null. Any corresponding row indices were dropped. But in the column scenario, we designed a couple of heuristics. The reason is that, since we depend on columnar data, we cannot physically do row-based computations directly. For row-wise null check implementation, a column-wise null check is performed. The obtained boolean array is then cast to $int32$ array and the addition of all columns is taken. Then we observe the following heuristics in deciding whether any null values are present.

- Heuristic 1: If the resultant sum/array contains an element with value equal to 0, it implies 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 all elements in that row are None.

Selection criterion is determined for how the dropping is done; when 'any' is selected, an addition value greater than 0 means that corresponding 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 factor 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) examples. To support efficient vector operations for linear algebra, we have implemented the support by using Numpy and Arrow compute kernels. Currently we do not support an internal version of linear algebra kernels, but instead rely on highly optimized kernels written in Numpy and Arrow. With Arrow we use 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 very important aspect of our system. Since we focus on providing the highest usability and seamless integration to the data science ecosystem, the ability to write programs in Python is essential. For this we introduce the

framework PyCylon, the Python API for data engineering workloads, as one of the major contributions from our research.

The 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 of our research is to make these tools available in Python. To enable C++ kernels for Python, the most effective and developer-friendly mode is to have Cython as the interface between C++ and Python. Cython has been widely used to write efficient Python bindings in scientific computing libraries like Numpy, SciPy, Pandas and many other research projects.

Figure 5.3 depicts a high level API abstraction in our data engineering framework. The lowermost layers consist of the core kernels for data engineering and communication. On top of these, 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 combined the 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 communicate with other libraries, providing Cython interfaces. Atop the core Cython API, we developed the PyCylon API. This is purely Python, and calls to the Cython interface when computations must be done on data engineering operators.

5.5.1 Cython for Python Bindings

Cython is a special language created to design Python programs for high performance computing. The advantage of using Cython is that it provides the ability to closely work with C/C++ data structures using the internal Cython APIs. When

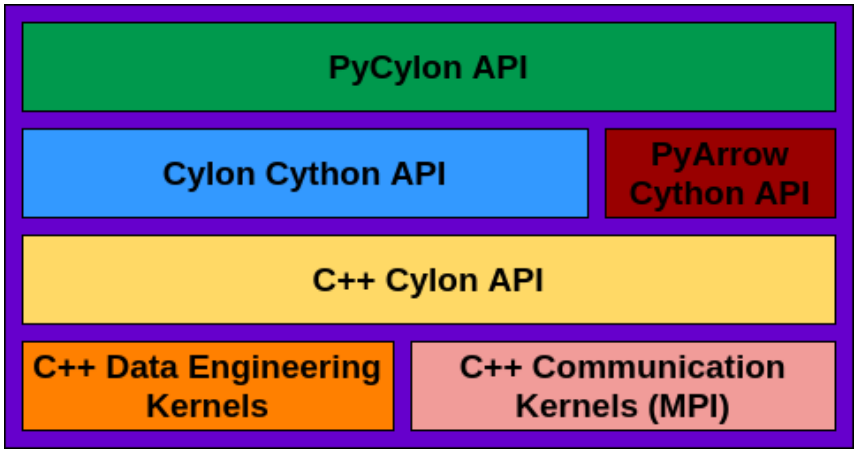

Figure 5.3: High-Level API Abstraction

building HPC systems for Python, Cython provides a wide variety of APIs to integrate C/C++ kernels such that C++ functions can be called from Python very efficiently.

In terms of data copying across languages, if the data are created on Python data structures (entirely Python) and if the data are primitive types, they 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 data created on PyArrow or LibArrow using Cylon. This provides an advantage since we do not need to serialize or deserialize data.

Referring to the function calls made from the Python interface, the actual computation takes place only in C++ based memory allocation formulated when loading data via Arrow. Here Python actually does not do any memory allocation, but calls Cython bindings to exercise 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 illustrates how a particular Python interface has been integrated

via language bindings to the core compute kernels.

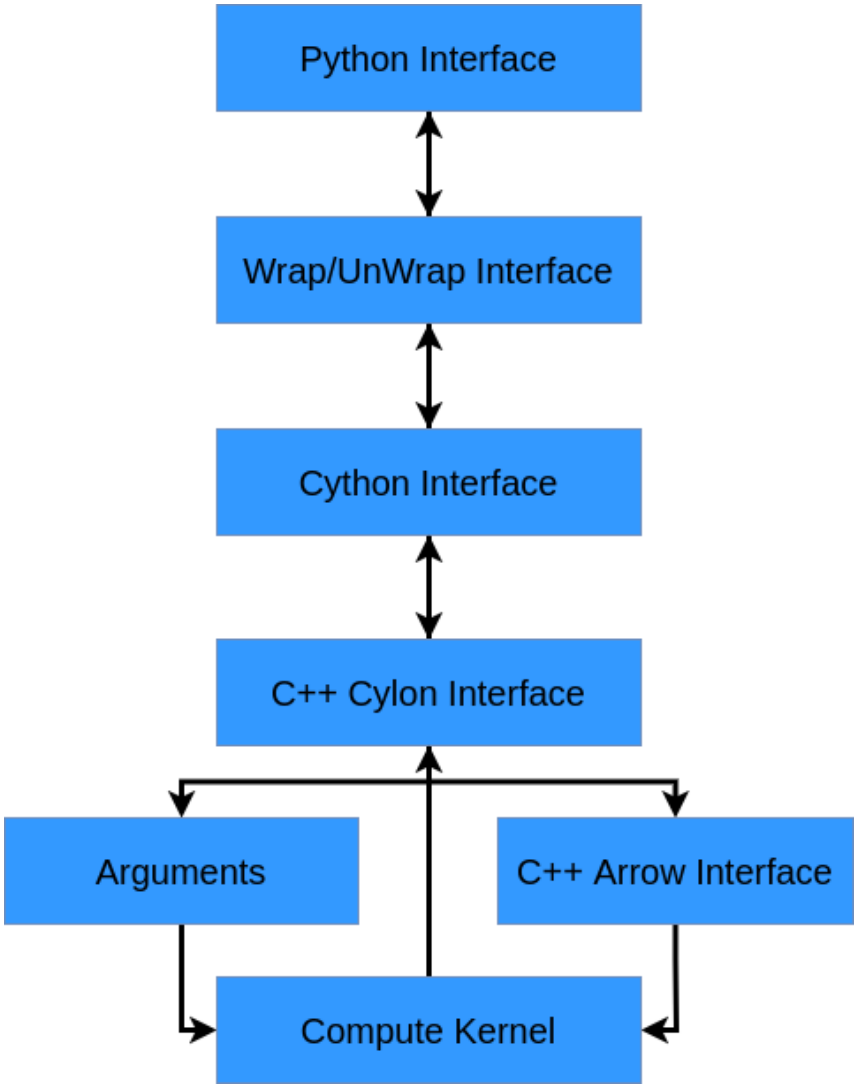

Figure 5.4: Cython Interfacing with Computing

Here any compute operator or compute interface that needs to be executed using C++ core kernels requires the flow shown in Figure 5.4. Such interfaces can be unwrapped to access the Cython interface object, which is generally a shared pointer in C++. The Cython interface provides an 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 a Cython object and extracts the underlying shared pointer in C++ implementation. The wrap interface uses 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 referring to the shared pointer created when the initial data were formed in memory. 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

As with the discussion in Section 5.5.1, the objective of this layer is to provide the efficient execution of C++ kernels and output to Python interface without copying data by instead using the underlying memory allocated from C++ kernels. For extended and advanced usage, 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 follow:

- 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
- I/O API : Input and output modules related to data (read, write, convert)
- Compute API : A high level wrapper for subsets 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. Similarly, 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. It 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

The highest level of API abstraction in our system is the dataframe. This dataframe API is designed such that it mimics the functionality of a state-of-the-art Pandas dataframe representation. The major difference is the functions with distributed computing interfaces and a few additional interfaces provide endpoints to get information about the distributed runtime (number of workers, worker ID, etc). The dataframe API is built on top of the Table API exposed in the Cython API. Table API contains all the core data engineering operators abstracted to be used as a simple Python class. But dataframe API abstracts away the distributed compute function calls and other internal details and provides streamlined function definitions. The supported dataframe operations are grouped into sequential (also function as pleasingly parallel operators) and distributed operators as shown in Tables 5.2 and

5.3 respectively. Here we mainly support the widely used data engineering operators. Our intention is to improve and add more operators based on the scientific applications developed as an outcome of this research.

In the dataframe representation, most of the operators are pleasingly parallel because the nature of the computation mimics a sequential computation that can be executed across all 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 on initializations, joins and duplicate handling.

5.7 Interoperability Among Python Data Structures

When it comes to application development, one of the most important features is the ability to seamlessly integrate with other existing data structures which are widely used in interdisciplinary domains. The data structures mainly represent the data in a useful abstraction with vivid compute functions helpful in manipulating data. As far as data engineering is concerned, the most widely used data formats are CSV, Parquet, HDFS and other binary formats like HDF5. When it comes to tabular data representation, the most prevalent data formats are CSV or Parquet. Parquet is an efficient columnar representation for data storage on disk. In data loading (discussed in Section 5.9), we provide support to load data into a PyCylon dataframe. To understand how we achieve data interoperability, it is necessary to showcase how we have interfaced the tabular data representation.

From Figure 5.5, the underlying data structures in the PyCylon framework can be seen. Here the dataframe is the aforementioned higher level API for data engineering operations. Table refers to the C++/Cython table abstraction integrated

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 from 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 less than
gt	Comparator for greater than
le	Comparator for less 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

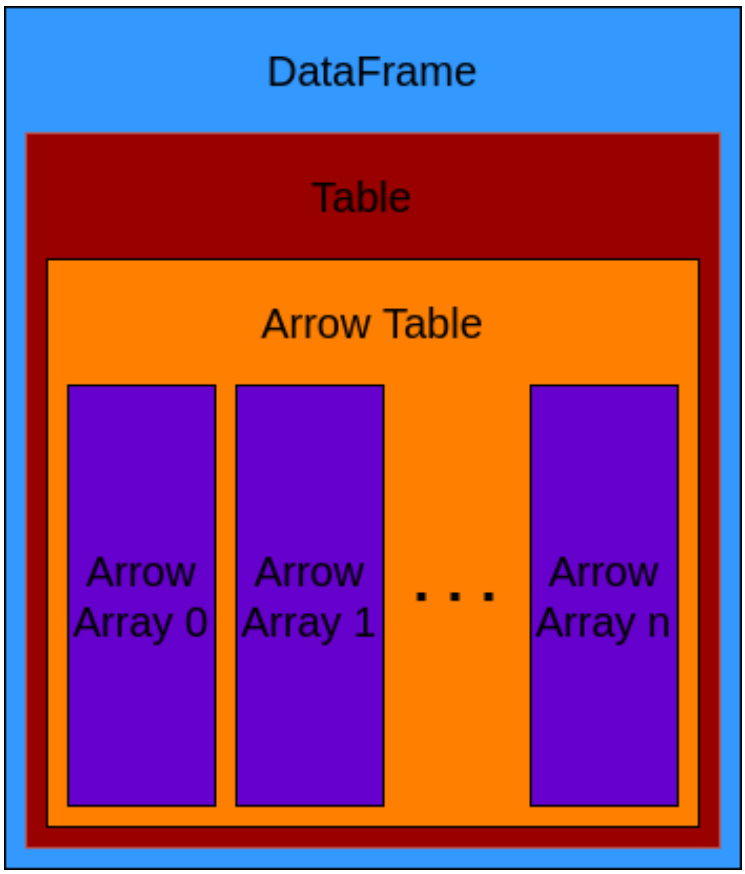

Figure 5.5: Data Structure Hierarchy

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 them to compute functions. One important feature with Arrow is that the Arrow table is immutable, so when we do a particular data engineering operation, we have to create a new table with updated data even though it is viewed as an in-place operation in the higher level. Here we replace the underlying shared pointer of the table from the original table with that created from the computation results.

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

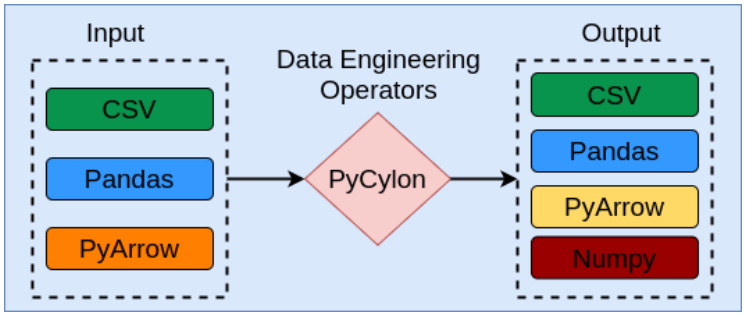

Figure 5.6: PyCylon Data Interoperability

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 workload. Figure 5.7 details 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 internally uses 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, such an 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 $chunk\_size = 1$.

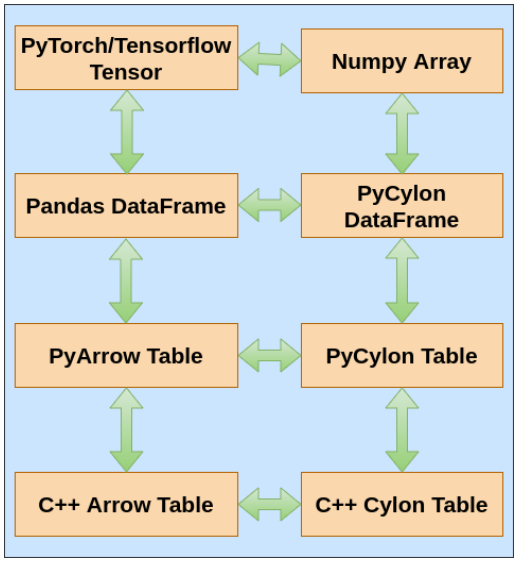

Figure 5.7: In-memory data conversion

5.9 Data Loaders

Data loading is the entry point for any data engineering application. In our data engineering framework, we currently support loading CSV and Parquet-formatted files via Arrow readers, Pandas readers and 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 possible 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 widely used deep learning frameworks.

5.10 Productivity and Usability

Productivity and usability play major roles in prototyping applications and designing efficient production frameworks. In recent years, Python has allowed us to

achieve such objectives. Specifically for data engineering workloads, Pandas operator specification has become the preferred operator specification. In our research contribution, our objective is to minimize 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 only be modifying an import statement and adding a distributed context to enable efficient data engineering.

The usage of data engineering operators with the sequential relational algebra operators can be seen in 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 sequential and distributed operation is the function call and the way the context is initialized. This is clearly seen in Algorithm 7. Here we provide a similar API to Pandas and optimize the API to allow for minimum code change and thus perform code migrations efficiently. The availability of local and distributed operators allows a user to develop an application with dynamic capabilities and solve complex data engineering problems more easily. Another advantage is that it allows the user to use pleasingly parallel or local operators along with distributed operators with minimum code changes.

Algorithm 8 shows how data filtering operators are designed in PyCylon DataFrame API similar to Pandas. In addition to the use of context, the rest of the data engi-

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()
```

neering code is configured to match with existing Pandas definitions for data filtering. Similar to a regular Pandas program, our APIs provide the ability to retrieve a subset of data by including a slice of row indices like $1:3$, giving a column name and retrieving those values, filtering out a subset of data by using a comparator operation, and filtering a dataframe using another dataframe with boolean values on the entire table.

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
```

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

set the index. Then *loc* operation can be executed by providing a slice of start 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 details math operations used on the dataframe with scalar values and dataframes with similar shapes. Currently PyCylon supports operations on a sequential mode and embarrassingly parallel mode. We have not yet implemented distributed operations for 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 done using the dataframe. This semantic is similar to a Pandas routine.

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

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 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 few benchmarks by grouping our compute operations in parallel and sequential execution modes. For sequential operations, we benchmarked our system with Pandas since it is considered a superb dataframe implementation on CPUs. For distributed operations, we compared the performance with Dask distributed dataframe, which is an implementation to scale Pandas on CPUs. We also conducted another set of benchmarks on NVIDIA GPU devices with Rapids cuDF. Apart from this, we tested the performance of our language bindings against the C++ core to identify the overhead from each binding.

6.1 Indexing and Searching

When indexing experiments, we conducted three sets to showcase the indexing performance, search performance and overall performance for a search followed by indexing. Generally an indexing operation is done only once or just 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 were unique in the indexing column. The data distribution provides us with the ability to query large numbers of data pointing to the same key. In the search operation, we searched for all unique keys in the data distribution. Figure 6.1 illustrates the performance results gathered from this experiment. From it we can see that PyCylon vector indexing mode outperforms Pandas, while PyCylon hash indexing mode is much slower compared to Pandas. The main reason for this is that internally, when we build the hash index, we populate a multimap (a C++

standard multimap), which takes more time to build when a large number of hash collisions are present.

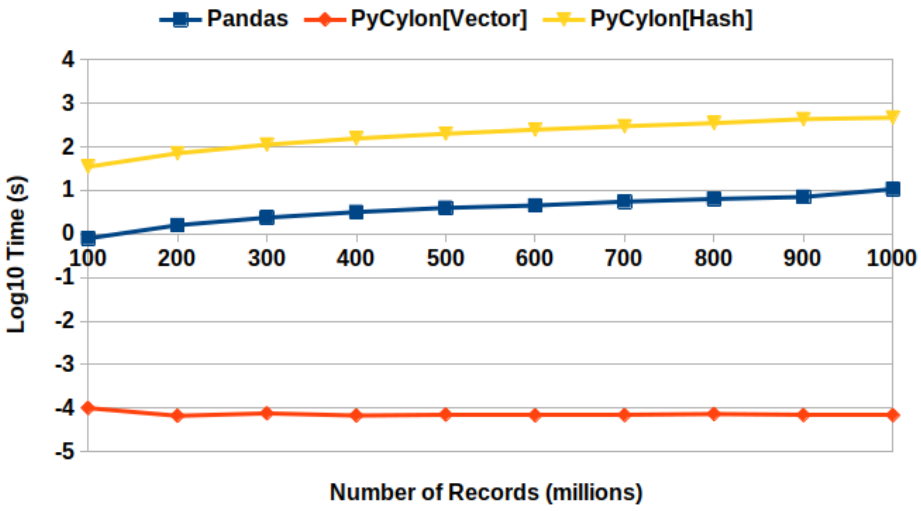

Figure 6.1: Indexing Operation Performance

Considering the search experiments, we used the aforementioned data distribution with the same index. Figure 6.2 has the results obtained for loc operation or search operation. It shows that both PyCylon search implementations perform the search much faster than Pandas.

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 experiment by taking the indexing factor into consideration and calculating 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 greater than or equal to 1. Figure 6.3 details the experiments conducted under these conditions. Even though the hash indexing performance is slow in PyCylon, it is clear that a search operation followed by indexing is still faster compared to the time taken by Pandas to do the same operation.

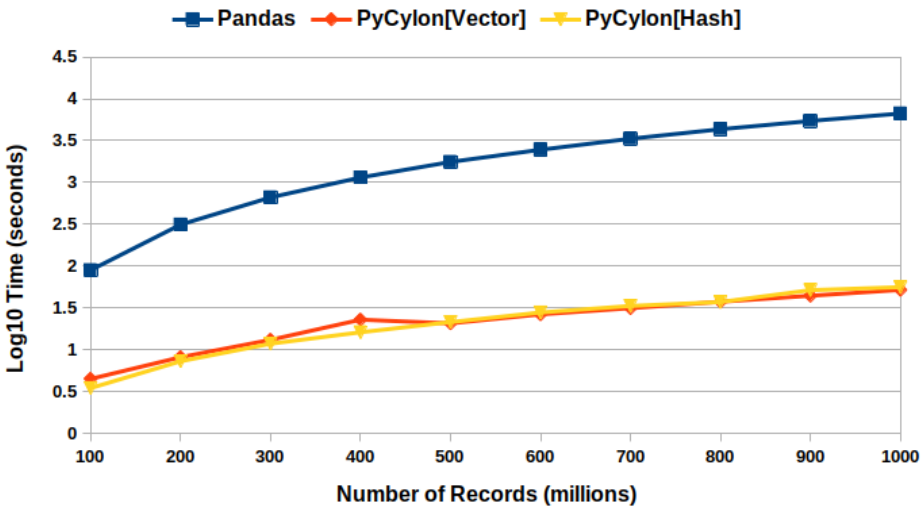

Figure 6.2: Search By Value Operation Performance

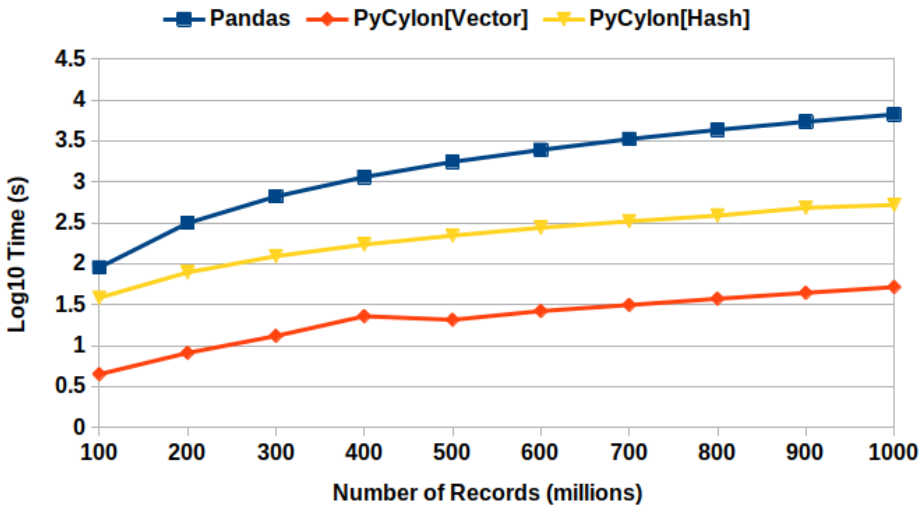

Figure 6.3: Indexing and Search By Value Operation Performance

6.2 Comparator Operations

In comparator operations, we refer to the operators which determine whether a particular value is greater, less, greater than or equal, less than or equal, not equal, and equal operators. With respect to a dataframe, these operators mean the same idea. Figure 6.4 has the results from the conducted experiments. We observe that the PyCylon comparator operator is slower in performance compared to that of Pandas and Modin. We did a micro-benchmark and observed that the internal performance for computation of the filter is as fast as Pandas or Modin. But the boolean value 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 enhanced in a future Arrow release. In addition to this, we also recommend these computations be done using either Arrow or Numpy compute kernels.

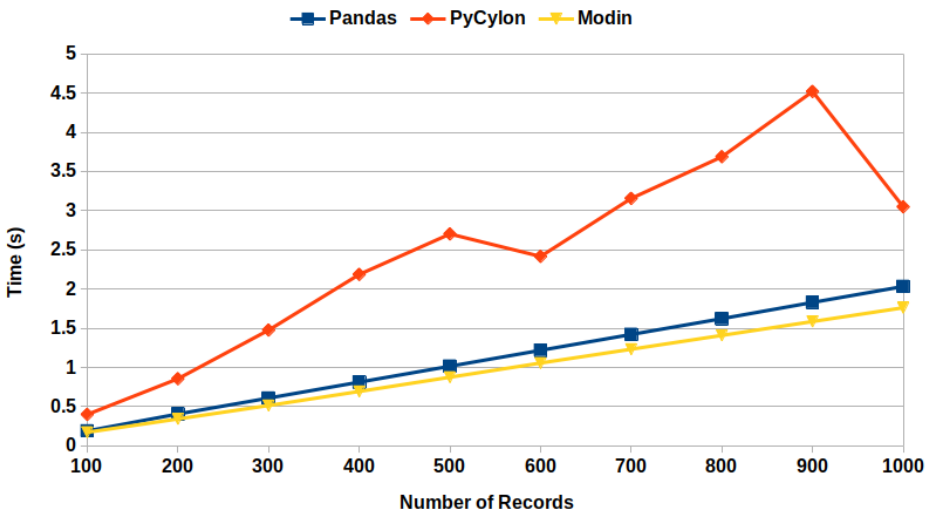

Figure 6.4: Comparator Operation Performance

6.3 Math Operations

The math operations we benchmarked are basic operations like scalar addition, subtraction, multiplication and division. In this case the benchmark refers to a scalar addition to the dataframe created with the variable number of records. Our findings are that the performance of Pandas and PyCylon is very close, but it is slightly slower compared to Modin. We internally use Numpy and Arrow interface as the compute engines during 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.

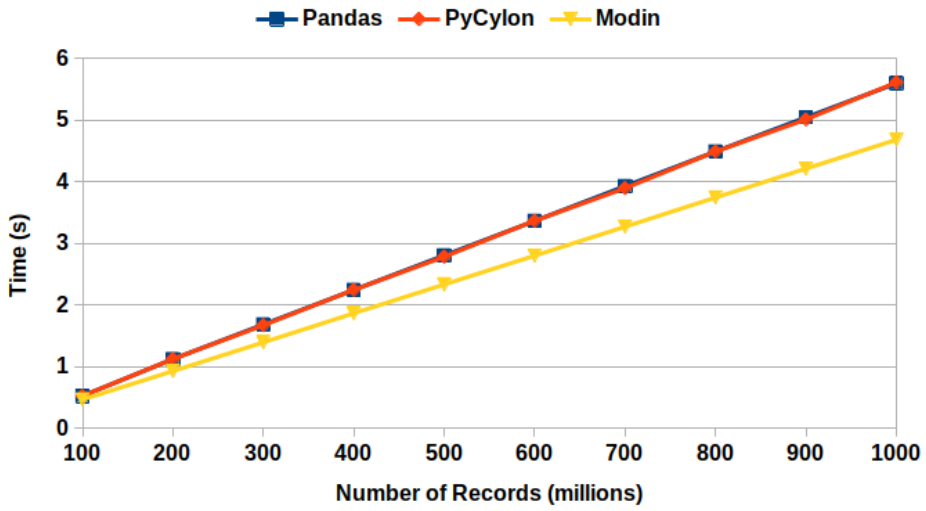

Figure 6.5: Math Operation Performance

6.4 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. The performance benchmark is shown in Figure 6.6. The results show that our null

handling implementation executes faster than the Pandas and Modin operator. This operation is executed column-wise.

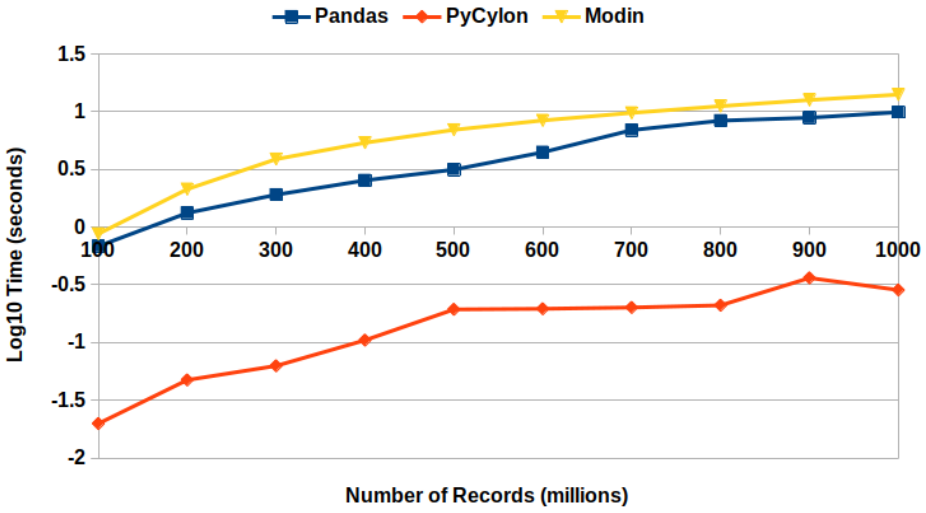

Figure 6.6: Null Handling (dropna) Performance

6.5 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. Random data were generated by considering the uniqueness of data to be 10% such that the join performs under higher stress considering hash functions and hash-based shuffles. In the parallel experiments, each process will be loading an equal amount of data such that the total amount is limited to 200M records. The results from Figure 6.7 show that our distributed join implementation is faster than Dask and Modin implementations. Also, the scalability in Dask and Modin is not very strong compared to scaling provided by PyCylon. Also, the Modin couldn't be scaled beyond single machine and failed in the execution. Figure 6.8 shows the speed up for the corresponding

experiments. Here we can see that the PyCylon speed up is significant compare to both Modin and Dask performance.

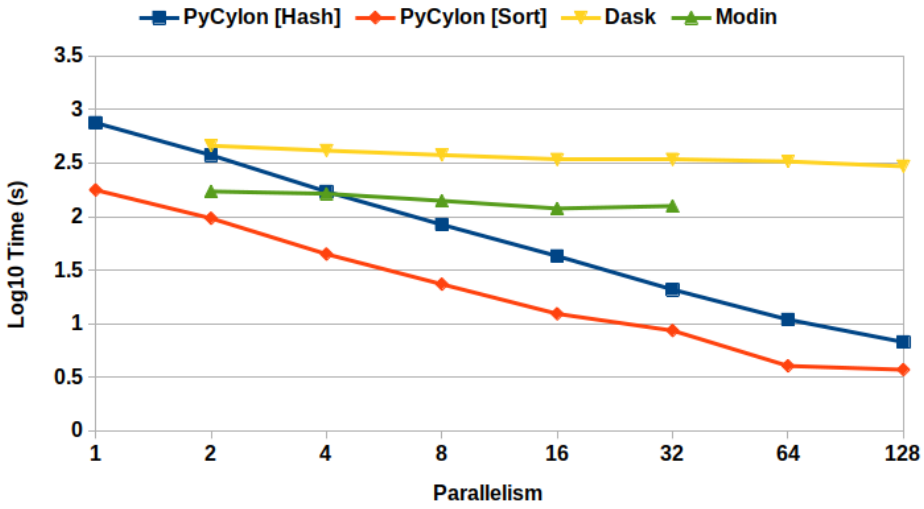

Figure 6.7: Distributed Join Performance

6.6 Distributed Drop Duplicates

Distributed drop duplication operation is a widely used operator to organize data such that overall data in a distributed computation must be unique. This is an essential operator when distributed deep learning is done in data parallel 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 and Modin distributed shows that PyCylon scales better than Dask and Modin. We also observe that the Dask workload does not scale after Parallelism 32. Also, the Modin couldn't be scaled beyond single machine and failed in the execution. Also figure 6.10 shows that the PyCylon has a significant speed up compared to both Modin and Dask.

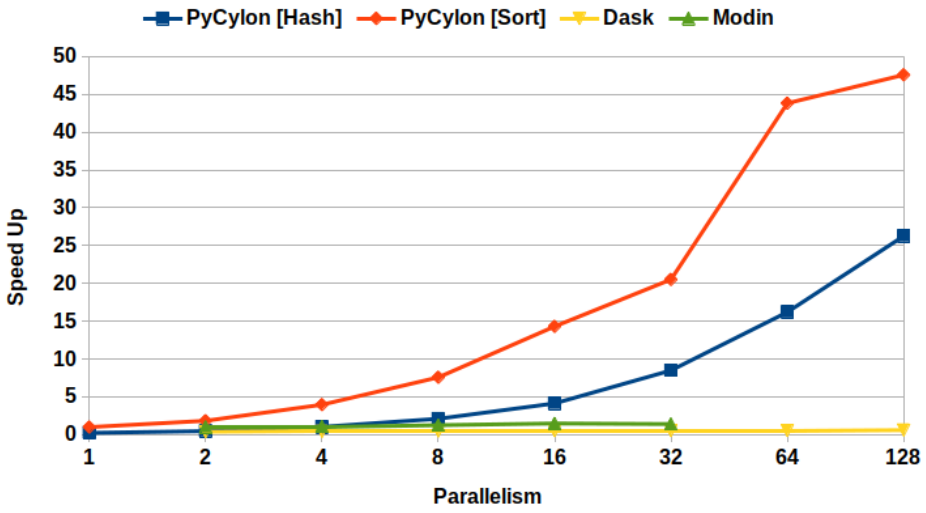

Figure 6.8: Distributed Join Speed Up

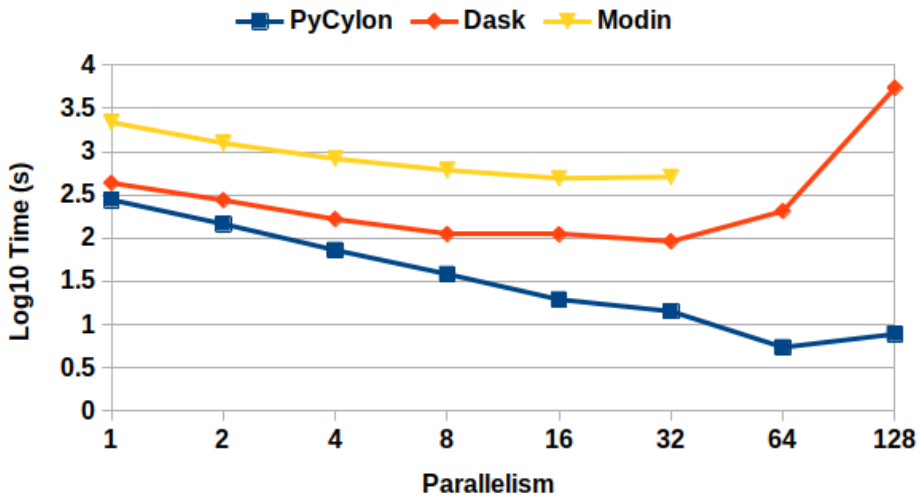

Figure 6.9: Distributed Drop Duplicates Performance

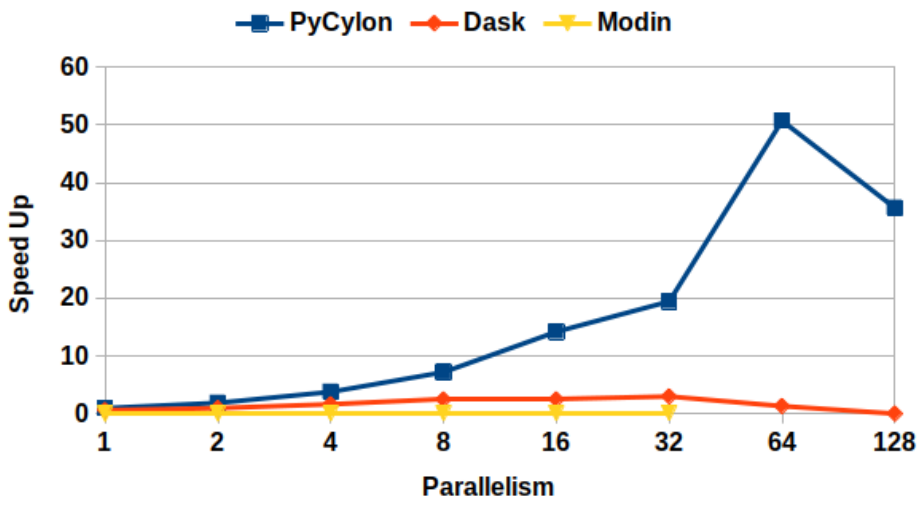

Figure 6.10: Distributed Drop Duplicates Speed Up

6.7 Join with CPU and GPU

Since PyCylon only supports CPU-based computations, we conducted an experiment to compare the join performance with cuDF on GPUs. Also, we used the Modin with sequential and multi-core experiments and Pandas with sequential execution. For this we selected an experiment criteria where the full resources of a CPU node were 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 our CPU and NVIDIA TESLA T4 for GPU. The CPU base experiments were executed on 48 cores, while GPU-based experiments consumed all CUDA cores (this depends on cuDF internal operations) of the GPU device. We selected the sort algorithm of PyCylon for the join benchmark. Figure 6.11 shows that the performance of a single GPU device vs. the selected multi-core CPU had similar performance. Alhough the CPU with a single core performs slower compared to a fully Cuda core-utilized GPU, the multi-core response provides the same result. This shows that CPUs can still be used to get decent performance compared to a GPU. An important point

to take away is that the time taken for join operation is much higher compared to other data engineering operators. For instance, the math operators in GPU are much faster than that of CPUs. Still, in a data engineering query, if a single join query is present, the time taken for that query is much higher compared to other vector operations. This also shows how resources can be utilized in an optimum manner. Another relevant detail is the memory limitation. The memory limit in the CPU node we used was 250 GB, while the limit of the GPU device was 16 GB. In that particular CPU, we could execute a much larger data engineering workload compared to the single GPU device for the same performance.

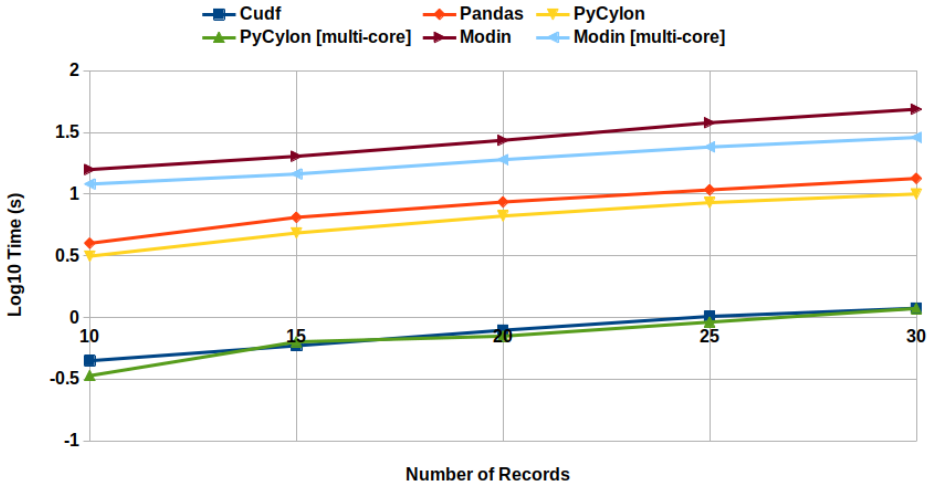

Figure 6.11: Join CPU vs. GPU Performance

Besides individual operator performance we also evaluated the gain obtained by each operator compared to Modin since Modin usage is similar to what PyCylon is offering for parallel application implementations. Figure 6.12 shows the average performance gain over each operator. Here we can see that most of our operations have significant performance gain compared to Modin. Here we calculated the gain by comparing the best result for each operator for sequential execution. PyCylon

gain for comparator and math operator is quite low. But, if we consider the execution time of a join or distributed drop duplicate function compared to a math operator or a join, a join roughly takes 100s of seconds for millions of records computation but the math or comparator operators take 2-3 seconds even for billion records. If we consider the execution time of an end-to-end application, always the computationally intensive operators consumes the majority of the time. In terms of overall execution, PyCylon can be still faster for a general application. But we are actively working on improving the gain for math and comparator operators.

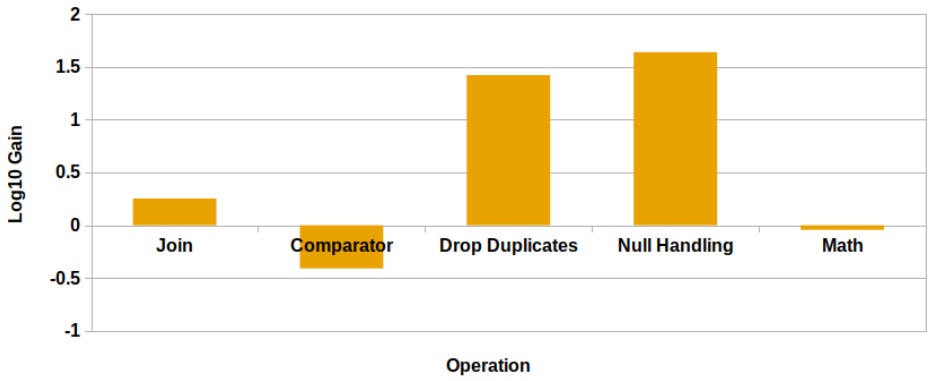

Figure 6.12: PyCylon Gain vs Modin

6.8 Overhead from Python

One of the most important attributes for creating high performance data engineering libraries is understanding the overheads caused by language bindings. Especially in the case of Python, there are some overheads when systems are designed inefficiently. As such, we conducted a set of tests to evaluate the performance of the language bindings by considering a set of operators. Figure 6.13 shows that the overhead caused by language bindings is very small compared to the core C++ API. Essentially 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.

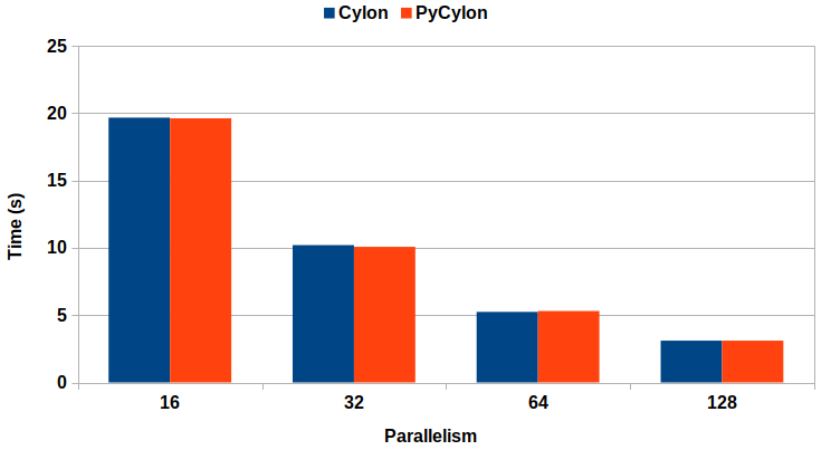

Figure 6.13: Performance Overhead by Language Bindings

CHAPTER 7

INTEGRATION WITH DEEP LEARNING FRAMEWORKS

Since the objective of our research is to make available an 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 core of PyCylon. In a data analytics-aware data engineering workload, there are three main factors that govern 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 exploration with data analytics. For such workloads, feature engineering and data engineering components are extensively modified to see how the data analytics workload performs for different settings. In such cases the data scientist must have room to write the usual Python script and run the data analytics workload efficiently, not only in single node, but across multiple nodes. Simple execution mode refers to running the workload with a simple mode to spawn the processes to run in parallel.

In regards to vivid frameworks, various ways to execute the framework on multiple nodes are provided. For instance, frameworks like Dask require we start the workers and schedulers on each node and provide host information for distributed communication. On top of that, MPI allows for 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. The majority of the frameworks sup-

port both CPU and GPU execution, so it is vital to provide the means to seamlessly integrate with these execution models to support data analytics workloads.

Figure 7.1 highlights the high level component overlay of a data analytics-aware data engineering workload. 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 or network information 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 workloads, the worker information, network information, chosen accelerator and data must be provided from the corresponding data engineering process. This mapping is 1:1 for data engineering worker to data analytics worker. But this can also be a many-to-many relationship.
- Stage 4: The worker information, network information and data will be used to execute the data analytics workload in distributed or pleasingly parallel mode.

Considering this generic overview on deploying deep learning workloads with data engineering workloads, we have integrated PyCylon with distributed data-parallel models for PyTorch, Horovod-PyTorch and Horovod-Tensorflow. Horovod has become an advanced distributed deep learning framework which supports a unified API for handling distributed deep learning on multiple frameworks. PyTorch, Tensorflow and MXNet are supported by Horovod. In our research, we paid close

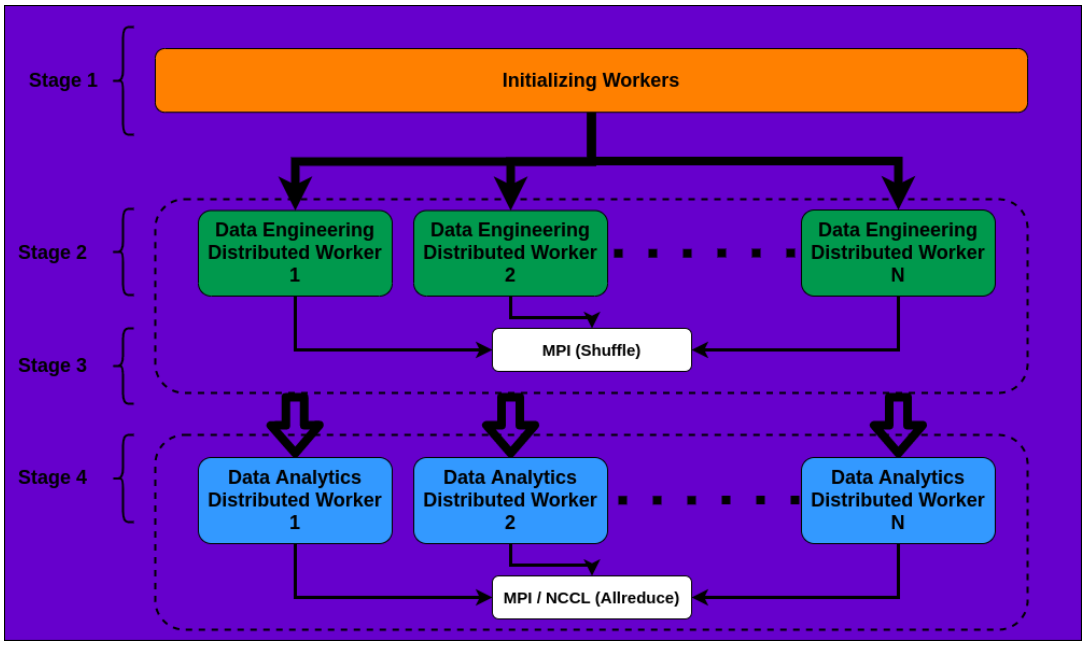

Figure 7.1: Integrating Data Engineering Workload with Data Analytics Workload

attention to PyTorch and Tensorflow. Horovod internally uses mpirun to spawn the processes, and this model fit very well with PyCylon internals as we relied on mpirun to spawn the processes. This makes PyCylon uniquely qualified as a supportive data engineering framework for Horovod.

7.1 PyTorch

PyTorch offers a distributed data parallel (DDP) module from the distributed runtime, which allows the user to initialize an existing model using DDP to make distributed computing easily available. But the key factor is choosing the distributed runtime, whether it be 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 especially on CPUs, the PyTorch initialization must be called since PyTorch internally does not handle the MPI initialization check. But if we use NCCL as the back-end, this constraint does not exist. This is one of the bugs we discovered from our previous research. For the PyTorch DDP, the master address and port must be provided because the NCCL back-end needs to identify which worker is going to be designated as the master worker to coordinate the communication. In addition, the initialization method has to be set. After the distributed initialization in PyTorch, PyCylon context must be initialized to set to distributed mode. After this stage, we complete the requirements for Stage 1 and partial requirements for Stage 3 (network information is also passed along with data in Stage 3, which is initialized in this step). Figure 7.2 is 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 in PyCylon, assuming the distributed mode initialization. We first join two tables and use the join response for a deep learning workload. The distributed join is called by providing the initialized context information to the join function. At the end of this stage, we create the resultant dataframe, and later on in Stage 3 this dataframe can be used to generate the Numpy array required for deep learning. This stage is common for any framework, including PyTorch, Tensorflow, etc. Figure 7.3 details 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 Stage 3, the data from Stage 2 is used to create tensors required for the deep learning stage. We also perform the data partitioning for training and testing. This stage is different than framework to framework since the tensor creation and data partitioning steps can have various internal utils. We do not use data loaders or data samplers, but note that these tools can be used to generate both. Figure 7.4 is 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, as well as the DDP model using the sequential model. We pass along device information such that tensors and models are copied to the corresponding devices (if accelerators are involved) for training and testing. This initialization part varies from framework to framework depending on the requirements and APIs. Figure 7.10 highlights 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. This is signigicant because PyTorch does not need to be compiled from the source to get MPI capability, as Horovod has already offloaded the distributed trainer, optimizer and allreduce communication packages so that the internal DDP mechanism in PyTorch is offloaded.

7.2.1 Stage 1

In Stage 1, the Horovod init method must be called to initialize the environment. After that the Cylon context can be initialized with distributed runtime true. If GPUs are used, the correct device must be set to PyTorch CUDA configs. To obtain the device IDs, we can either use the rank from Horovod initialization or PyCylon initialization, but at the moment Horovod supports local rank as well, and it is more suitable in terms of effortlessly integrating with the distributed runtime for Horovod-PyTorch. Figure 7.6 shows a sample code snippet demonstrating how this is accomplished.

```
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

Again, as with 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 the Horovod runtime and data can be prepared for a deep learning workload.

7.2.4 Stage 4

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

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.

```
# 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

7.3.1 Stage 1

To start this run, we initialize Horovod and PyCylon. As with PyTorch, we also need to decide how the device is selected depending on the accelerator. The Tensorflow config API provides a listing of GPUs, and this information is added to the Tensorflow configurations to make all the GPU devices available. Figure 7.8 is 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

As in 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 simpler and more structured. The Tensorflow dataset API can be used to create tensors from Numpy arrays, and

this API can be used to shuffle and create mini-batches, as expected by the deep learning workload. Figure 7.9 contains a code snippet detailing this step.

```
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 requires a set of initialization steps to train a Tensorflow deep learning model. Just like with PyTorch, the Tensorflow loss function, optimization function and neural network model are compatible with Tensorflow-Horovod internals. The gradient tape from Tensorflow autograd can be used, and for this Horovod provides a DistributedGradientTape operator which takes the gradient tape instance as a parameter. In addition, prior to training, this DistributedGradient-

Tape must be initialized with the model parameters and loss function, and the optimizer values must be set to initial values. Again, the model parameters and optimizer values must be broadcast using designated Horovod broadcast functions. Figure 7.10 illustrates this.

```
# 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 an end-to-end workload containing data engineering and data science. Our objective is to showcase how a sequential workload can be designed in a distributed manner using PyCylon and run a deep learning workload seamlessly on only a single script with a unified runtime. For this, we selected an application from academic science which involves Pandas dataframe for data engineering and PyTorch for data analytics. The original application is sequentially executed, and we have implemented a distributed version of this application with PyCylon and Distributed PyTorch with unified execution.

8.1 UNOMT

UNOMT application is part of CANDLE[WYMY+, XAB+21] research conducted by Argonne National Laboratory, focusing on automated detection of tumor 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 approach 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 an efficient data pipeline for a high-end scientific problem.

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

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

UNOMT application consists of two main components. The first is a data engineering workload which cleans the raw data to formulate the trainable parameters. The second component is the deep learning modelling with the preprocessed data taken from the data engineering workload. There are multiple networks involved working on small and large datasets in the training process. In our research we focus our attention on the distributed network, which is designed to calculate the drug response based on the cell-line information. This network is a regression network supported by two others which provide gene configuration-based and drug feature-based networks.

8.2 Deep Learning Component

UNOMT refers to a unified deep learning model with multi-tasks to predict drug response as a function of tumor and drug features for personalized cancer treatment. Precision oncology focuses 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 exposure to a treatment with a specific drug concentration. In this application, a set of drug data obtained from NCI60 human tumor cell line database is used to predict the drug response by considering gene expression, protein and microRNA abundance. As per the considered scope, the UNOMT application we focus on in the study is conducted on

single-drug response prediction done using NCI60 and gCSI datasets. We used 1006 drugs from NCI60 database for this evaluation and gCSI for the cross-validation.

To evaluate the drug response predictions (regression model), the metrics selected 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 are 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 features RNA-sequence data, drug feature data and drug concentration feature sets as input features. The RNA-sequence data becomes an input to a pre-trained model called a 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 taught to calculate the drug response. Figure 8.1 refers to the gene network which is being trained prior to becoming an input to the main drug response model. The gene network only contains three dense layers, each followed by a ReLU. Figure 8.2 refers to the drug network, which also has 3 dense layers. This network is pre-trained prior to being used in the drug response regression network.

The drug and gene networks 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,

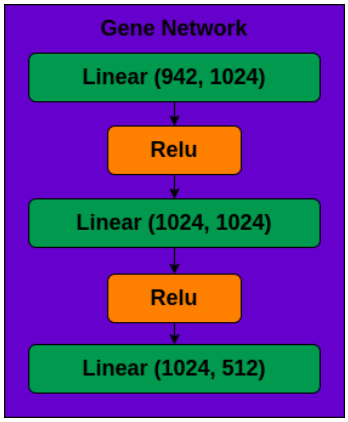

Figure 8.1: Gene Network

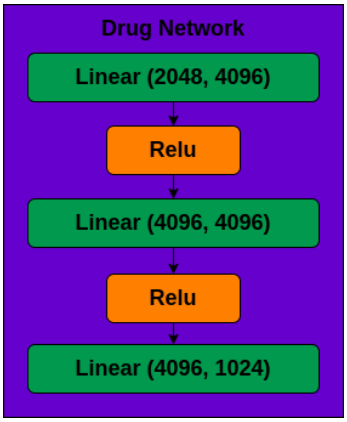

Figure 8.2: Drug Network

there is another residual block being used repeatedly. This layer is called the drug response block module, which contains 2 dense layers followed by a dropout layer and a ReLU activation layer. Figure 8.3 depicts the response block module.

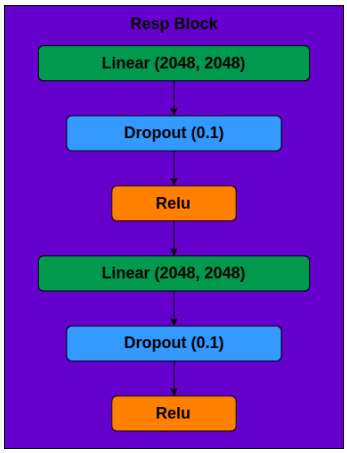

Figure 8.3: Response Block Module

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 as well. Finally, the regression layer contains a single output dense layer. The number of response blocks can be customized dynamically, as well

as the number of dense layers that follow 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.

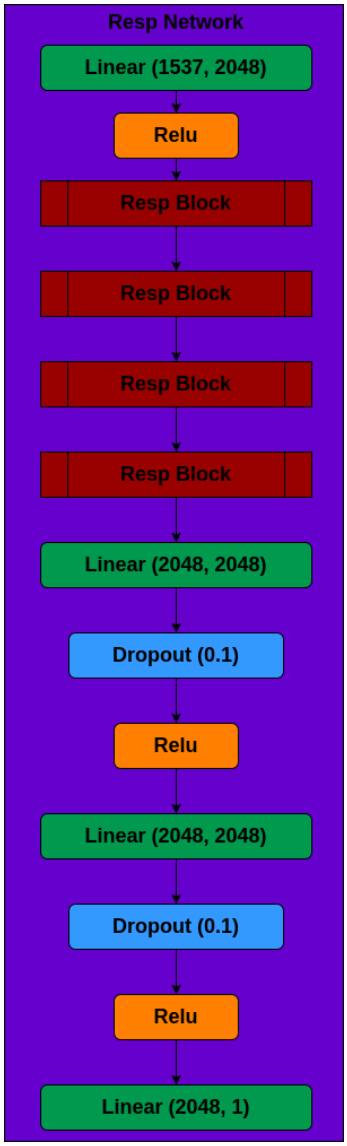

Figure 8.4: Response Network

Note that this network is trained in a distributed data parallel model since it contains a very large dataset and a complex network compared to the other

examples trained simultaneously. The corresponding data engineering component is also distributed data parallel, which is discussed in detail in Section 8.3.1.

UNOMT deep learning component consists of five other auxiliary networks trained on smaller datasets to predict a few features. These networks are trained to achieve a broader view about the data related to drugs and cell information. In our research we paid more attention to the network with larger data to be trained efficiently. The other networks are as follows:

- Cell-line category classifier: Tissue category (normal vs. tumor) classification
- Cell-line types classifier: Tissue-type classification (melanoma, gynecological, 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 classifier, the number of hidden units, activations and output parameters vary.

8.2.2 Cell-Line Category Classifier

The cell-line category classifier relies on ClfNet to predict whether the cell category is a tumor, fibroblast or normal. Figure 8.5a shows the network architecture design for this network. The data used in this network are RNA sequence data and cell metadata. 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 for this classifier. Note that this

network is trained sequentially and the corresponding data engineering component is also a sequential as described in Section 8.3.2.

8.2.3 Cell-Line Types Classifier

The cell line types classifier also uses ClfNet to predict the cell-line types. This is a network customized by using the generic network created for cell-line-based analytics. Figure 8.5b refers to the cell-line types classifier. The difference from the generic network 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

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

8.2.4 Cell-Line Sites Classifier

The cell-line sites classifier is used to predict cancer sites. For this classification, the same generalized network is used but the number of classes is 17. Figure 8.5c lists 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

(a) (b) (c)

Figure 8.5: (a) Cell-Line Category Classifier, (b) Cell-Line Type Classifier, (c) Cell-Line Site Classifier

- endocrine and neuroendocrine
- digestive/gastrointestinal
- breast
- hematologic/blood
- eye
- liver/bile duct
- neurologic
- unknown

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 Section 8.3.3. The network is trained sequentially as is the corresponding data engineering workload.

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 the analysis. Figure 8.6b depicts the network designed to calculate drug likeliness. This network is trained sequentially as per this application, and the corresponding data engineering workload is discussed in Section 8.3.3.

(a) (b)

Figure 8.6: (a) Drug Target Family Classifier, (b) Drug QED Regression Network

8.3 Data Engineering Component

UNOMT application uses 2.5 million samples of cancer data across six research centres. This model analyses the study bias across these samples to design a unified drug response model. Before building this model, the application computes 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 does not scale. We reengineered this application to a scalable data engineering workload and designed a seamless integration between data analysis and data engineering workload consuming state-of-the-art high performance computing resources. We also integrated a Modin-based implementation to showcase the performance comparison with our implementation. The data engineering workload is executed in CPU-based distributed memory and the data analytical workload can be either executed in CPU or GPU. We use Pytorch for data analytics workload and extend it to PyTorch distributed data-parallel training. Our objective is to integrate an HPC-based full stack of data analytics-aware data engineering for scalability. This feature is only

supported by PyCylon at the moment. Also, we stress the importance of designing a BSP-based model for deep learning workloads associated with data engineering components for better performance and scalability in HPC hardware.

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 8.7a refers to the main dataset, which contains the drug response. The raw dataset possesses 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 preprocess a drug ID column to remove symbols from the columns and create a consistent drug ID. Once the data are cleaned, they are scaled with the Scikit-learn preprocessing library for scaling numerical values. After this the data are fully converted into a numeric type to provide numeric tensors at the end for the deep learning workload. In the parallel mode, we partition this dataset with the set parallelism, upon which it is passed to the corresponding operators.

To formulate the global dataset, we require two other datasets which act as metadata to filter and process the the main drug response dataset. The first is the drug feature raw dataset, which contains drug features required to be located in the drug response data. There are also two sub-datasets that contribute 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 them as a numeric array which is later converted to a numeric tensor for deep learning. This data processing workflow is shown in Figure 8.7b.

The other dataset required is the RNA sequence dataset containing information about RNA sequences. Here the dataset is first processed to remove specific symbols by a map operation, and then duplicate records are dropped by way of a drop duplicate operator. Then an index is set for this dataset, and later on scaling is done on the numeric data using the Scikit-Learn preprocessing library. Finally, the data is cast to a numeric type and preprocessed RNA-sequence data are formulated as a Numpy array, which is later converted into a numeric tensor for the deep learning workload. This data processing pipeline can be found in Figure 8.7c.

Once the drug response initial dataset, drug feature data and RNA-sequence data are preprocessed, the final dataset for drug response model is engineered as shown in Figure 8.8. The processed drug response data are further feature-selected and a unique operation is applied. Then the RNA sequence data is filtered by checking whether specific drug-related RNA sequences are present. The same is done for the drug feature dataset. These two operations are done by the isin operator. Afterwards the common drug set is selected by performing an and operation, and later these common drug-related drug response data filters are used to get the final drug response data.

Among the operators applied, since we partitioned the data, each data engineering operator can work independently in a pleasingly parallel manner. But we can rely on the distributed unique operator to 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 metadata, and we use them to filter a very large dataset which is converted to formulate the expected input for the drug response model.

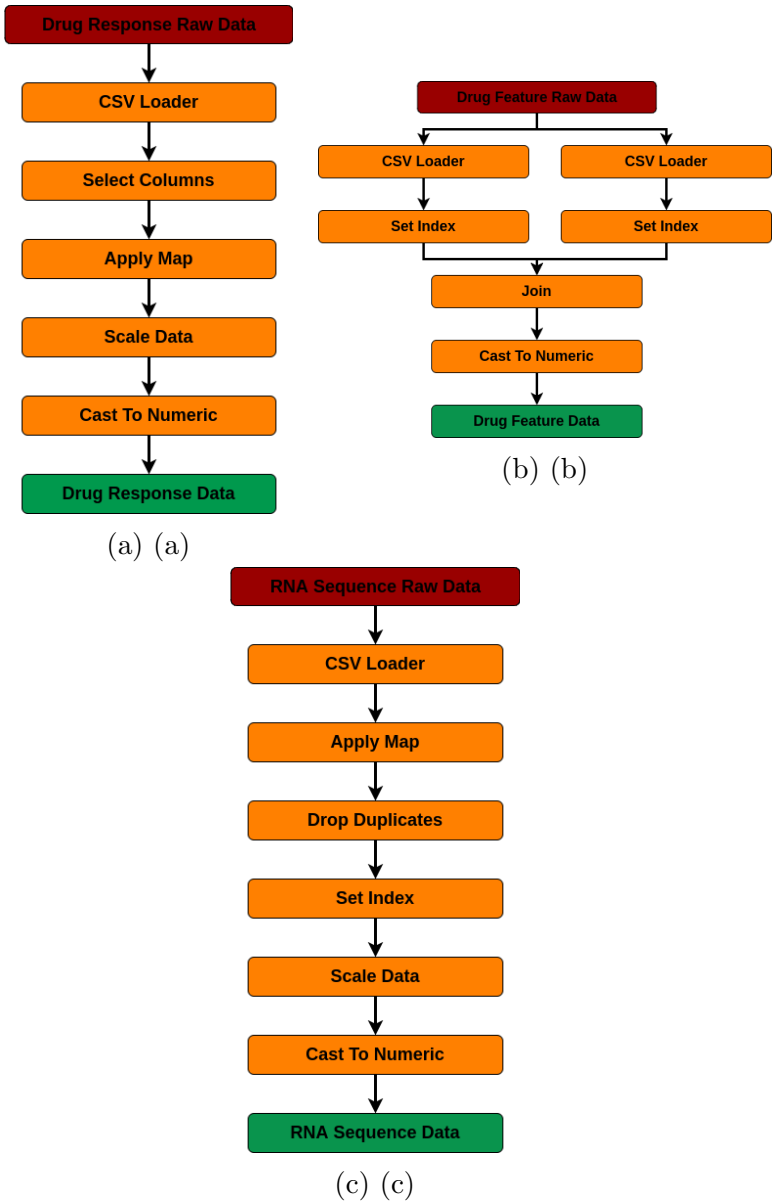

Figure 8.7: (a) Drug Response Data Processing, (b) Drug Feature Data Processing, (c) RNA Sequence Data Processing

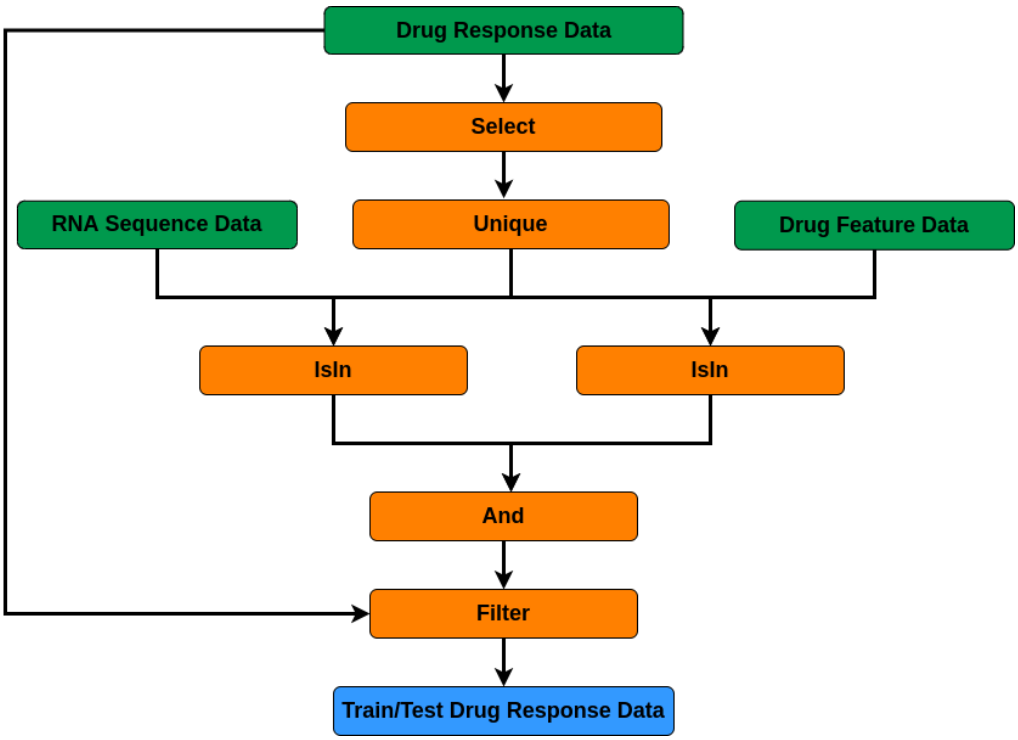

Figure 8.8: Drug Response Overall Data Processing

8.3.2 Cell-line Data Processing

The cell-line data processing component produces the numerical data required for data analytics in the neural networks discussed in Sections 8.2.2, 8.2.3, 8.2.4 and 8.2.5. Here the first step is to load the cell-line information by preprocessing the raw data for cell-line information. Figure 8.9a contains the initial steps to preprocess the cell-line metadata to extract the cell-metadata. The operations are executed in a sequential fashion followed by a set of filtering operations. An encoding operation is deployed to convert the string naming conventions to a numeric category for classification. In the final step, the data are converted to a Numpy array and then into a tensor for the deep learning workload.

Using the preprocessed cell-line metadata, the overall feature set required for the deep learning component is engineered as shown in Figure 8.9b. Here we use the preprocessed RNA-sequence data along with the cell-line metadata to formulate the final feature vector by performing a join operation on the drug sample column information in RNA-dataset.

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 preprocessed by loading the raw data CSV, 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 are preprocessed as seen in Figure 8.10.

The preprocessed drug property dataset can obtain the dataset required for the drug QED regression network. Figure 8.11a illustrates the corresponding work-

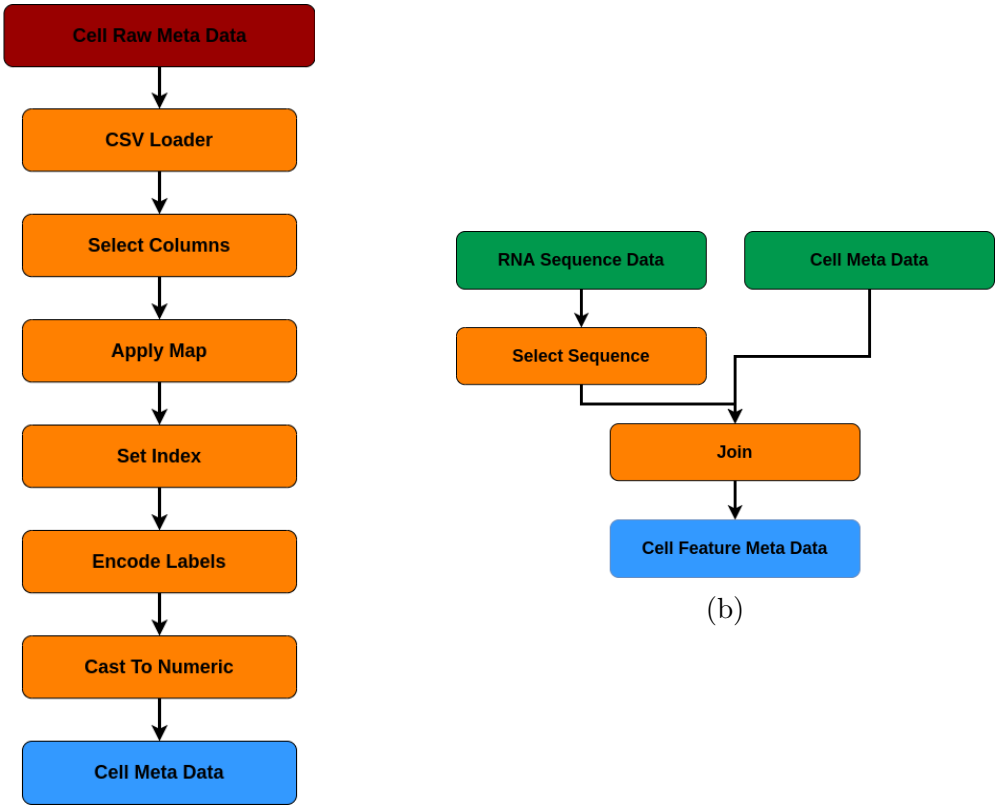

Figure 8.9: (a) Cell-Metadata Processing, (b) Cell Feature Metadata Overall Processing

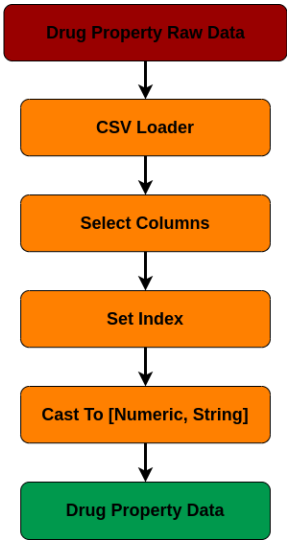

Figure 8.10: Drug Property Data Processing

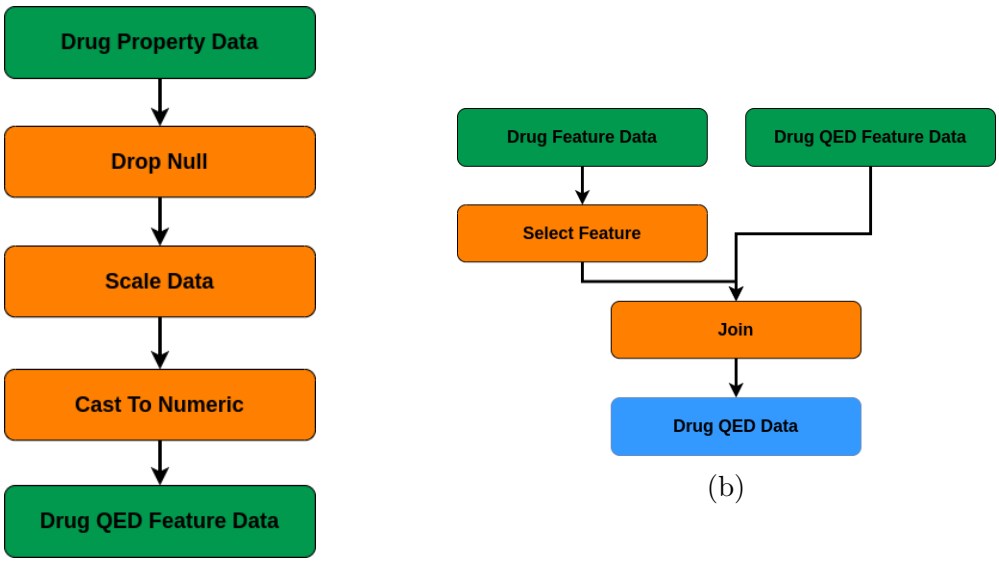

Figure 8.11: (a) Drug QED Feature Data Processing, (b) Drug QED Data Processing

flow. Here the drug property data is used to drop all the null values, scaled using Scikit-learn preprocessing library, then finally cast to numeric types and used in the deep learning network to form tensors. With the preprocessed drug feature data (a sub-input to the drug response network) and drug QED feature data, the dataset required for drug QED regression network is formulated as can be seen in Figure 8.11b.

8.4 Performance Evaluation

The original application was a single threaded application 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 could scale our workload on CPUs for data engineering. We also extended the deep learning

component of this application by integrating with PyTorch distributed execution framework on both CPUs and GPUs using MPI and NCCL respectively. In this benchmark, our goal 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 used the drug response network-related larger data distribution for the application benchmark, while the smaller networks require a much smaller execution time compared to this larger model.

For the experiments, we had two sets of clusters for CPUs and GPUs. For CPUs, there was 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 node. GPU experiments had Tesla K80s with 8 GPU devices on Google Cloud Platform. For single-node single-process executions, we used the same Victor nodes. For the sequential performance comparisons, Pandas, PyCylon (single core) and Modin (single core) were deployed. Finally, for the distributed performance comparisons, we used PyCylon and Modin on single node multi-core scaling. We selected Modin instead of Dask because it is closer to the data engineering stack proposed by PyCylon as a result 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 proposed systems PyCylon, Modin and Pandas. Modin provides the ability to convert a Pandas data engineering workload by means of a single line of code, whereas PyCylon offers a dynamic API allowing 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 preprocessing workload used for the drug response regression network. Figure 8.12 has the single core performance for the aforementioned data engineering workload. We observe that the performance of PyCylon and Pandas are very similar, while Modin is much 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.

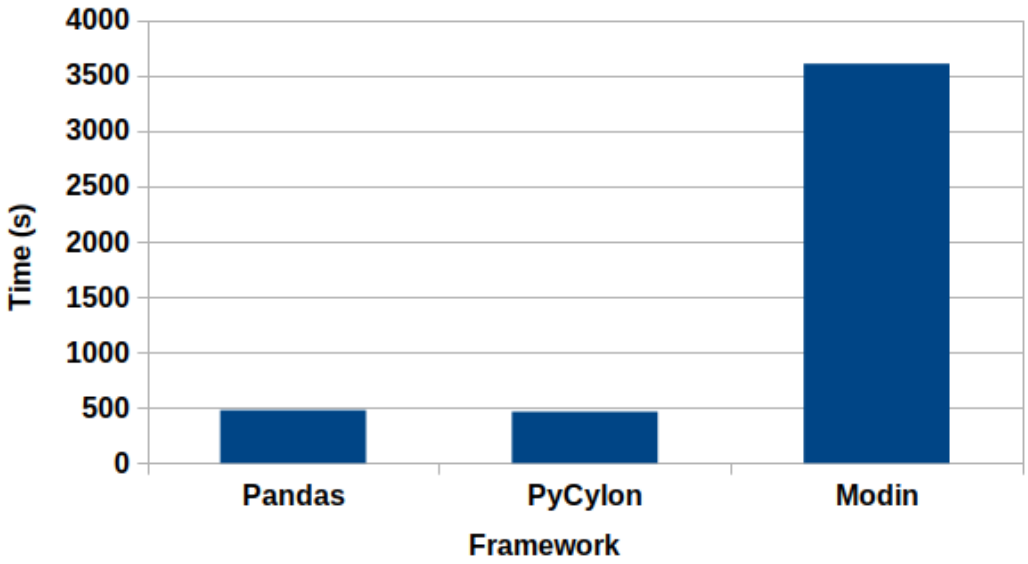

Figure 8.12: Sequential Data Engineering

We investigated the underlying sub-components to learn why the sequential performance in PyCylon was better compared to Pandas. Table **??** lists the time taken for significant sub-components. The time breakdown shows that a set of components

took much longer for the sequential execution. The drug response data loading, timing data, drug analysis and data split lasted 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 component. The drug analysis component contains an iteration through all the data in the dataframe to create a subset of data by doing a statistics calculation using Scikit-learn preprocessing library. Looping through the dataframe is quite slow in Modin compared to both Pandas and PyCylon. The split operation uses the Scikit-learn preprocessing 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 so 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 very slow in Pandas when it comes to the UNOMT 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 metadata 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. We compared the multi-core performance of Modin to PyCylon. We encountered an issue in scaling the Modin

dataframe across nodes (Modin mentions advanced applications with distributed cluster is experimental. It involves spinning up a ray cluster and this component was not functional as per our experience with Modin for this application). The current Modin documentation also mentions that the distributed component is experimental. In addition, we found most of the Modin benchmarks in the published research are done on multi-core by comparing with Pandas. Figure 8.13 showcases the multi-core data parallel data engineering time breakdown for PyCylon vs. Modin. It is apparent that the PyCylon scales relatively well compared to Modin.

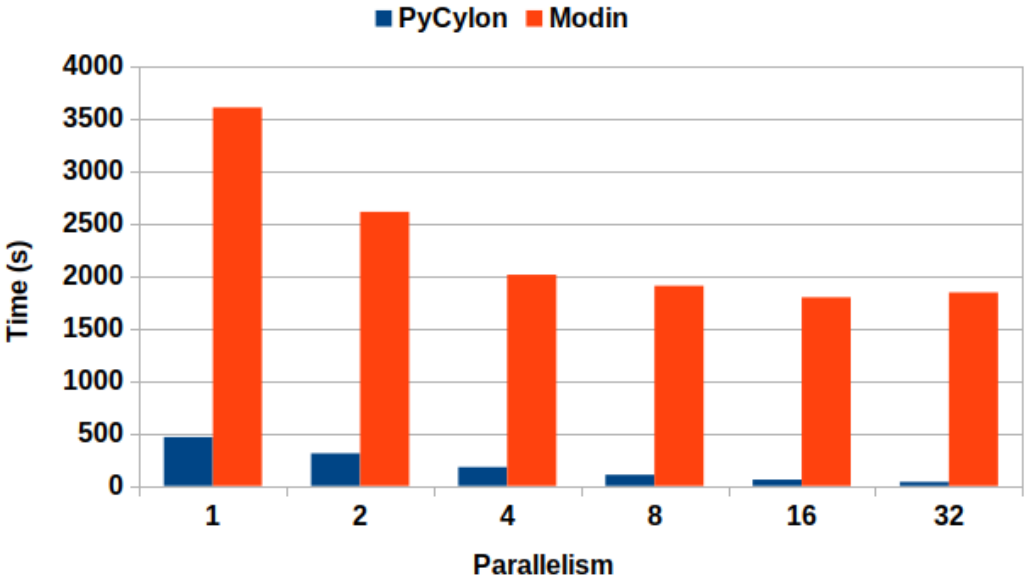

Figure 8.13: Multi-Core Data Parallel Data Engineering Performance

Considering the speed-up gain compared to the base implementation of each framework, we plotted the speed-up graphs as depicted in Figure 8.14. The speed-up from Modin is relatively low compared to PyCylon. The Modin dataframe internally uses Ray to scale up the dataframe. This is the default execution engine for Modin. By design, Modin does not have its own distributed execution engine, but relies on Ray to do the distributed computation. In PyCylon, we have multiple modes of

executing the application, such as distributed data parallel, pleasingly parallel and sequential. Given the nature of this application, we decided on a pleasingly parallel approach. We also investigated whether Modin can provide dynamic parallelism as required by the application, only to find Modin does not have this capability. In the case of Modin, the output is shown as a sequential view or one dataframe, and operators executed 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 were using the correct number of drugs and cell-line information at intermediate stages of the distributed computation.

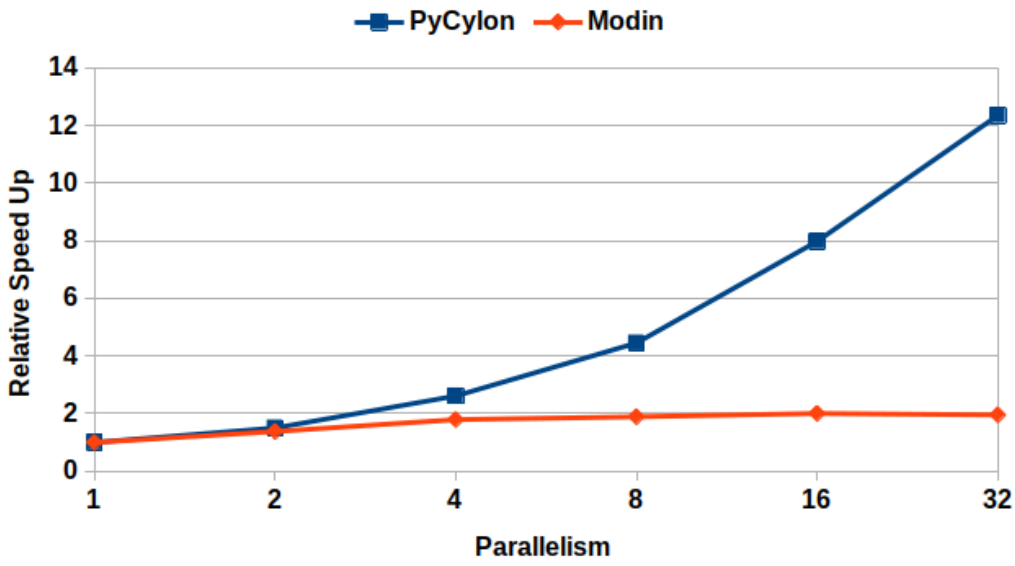

Figure 8.14: Multi-Core Data Parallel Data Engineering Speed-up

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 a main component which produces sub-datasets. Figure 8.15 compares the time breakdown for these data engineering components. The 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 due to an iterative computation happening on Python with a Scikit-Learn data processing library. We also observed that much time is spent on iterating the loop in Python. Here 2.5M samples were iterated to do the calculation. We did not improve this performance by doing further forking processes since it hindered distribution execution of threads along with an MPI processes. This execution can be improved by offloading it to a C++ kernel, although in general data engineering practices, we cannot introduce generic kernels to do such operations, which are application-dependent.

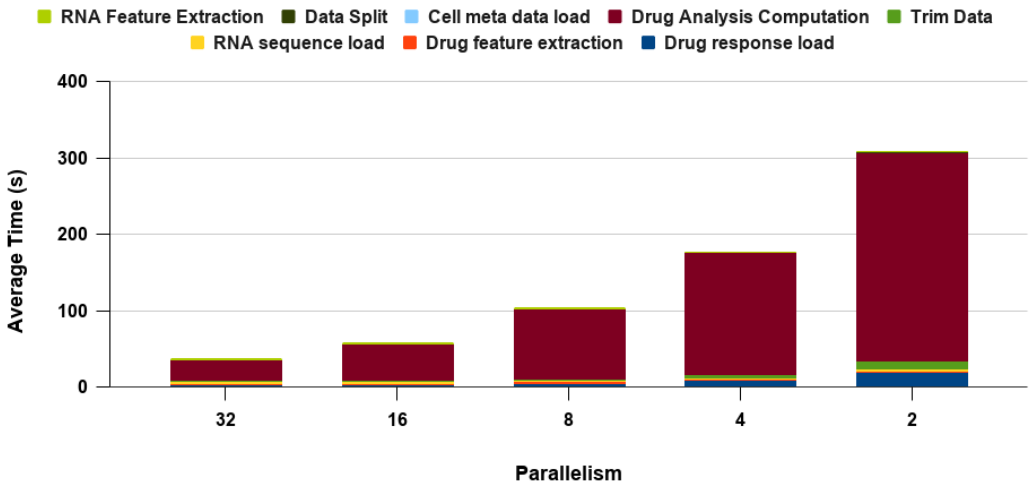

Figure 8.15: PyCylon Distributed Data Engineering Time Breakdown

Figure 8.16 plots the time breakdown for the same experiment based on total time as a percentage. It is clear that distributed components reduce as parallelism increases and non-parallel components related to metadata preprocessing are a constant component of the overall workload. The results demonstrate that the majority of the time is spent on drug analysis computation.

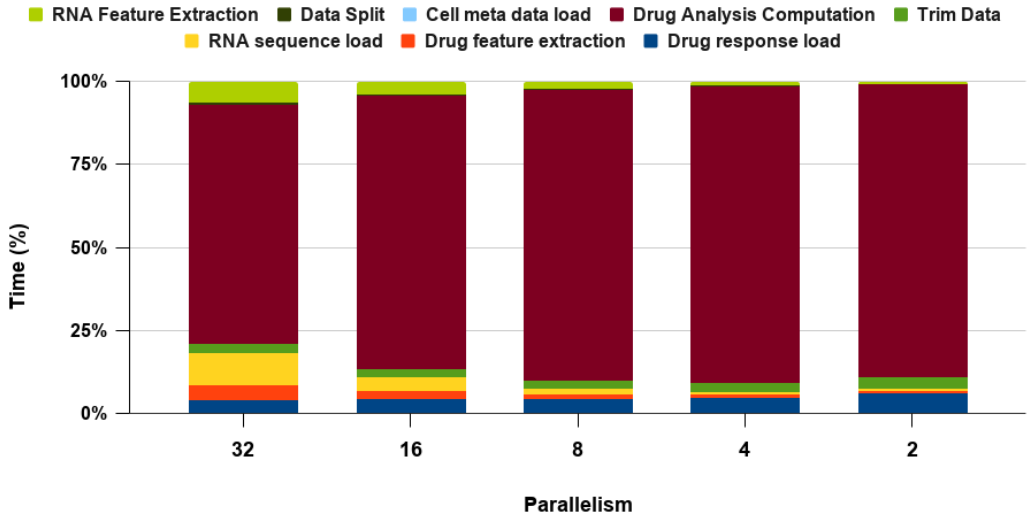

Figure 8.16: PyCylon Distributed Data Engineering (CPU) Percentile Time Breakdown

Figure 8.17 contains the data engineering time for distributed experiments across multiple nodes. We employed 6 physical nodes of the Victor cluster, and each node used 16 processes for the computation. Note that even though the workload was scaling, the scale-up factor was not that significant. The major reason for this is the dominant drug analysis computation in the data engineering component. Even though the data was partitioned, more time was spent on this component than any other. We encountered problems in scaling Modin dataframe across multiple nodes. Also note that Modin framework is an evolving framework similar to PyCylon in the parallel dataframe domain.

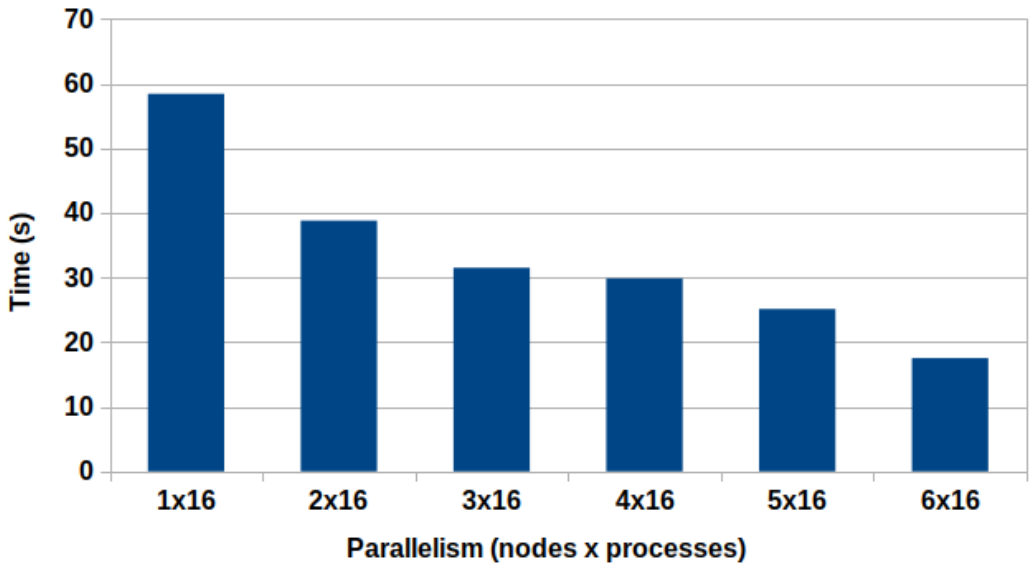

Figure 8.17: PyCylon Distributed Data Parallel Data Engineering

8.4.3 Data Analytics Distributed Performance

For the data analytics scaling experiments, we selected PyTorch distributed communication framework with MPI for CPUs, and NCCL for GPUs. The single process experiment results are the same for both PyCylon and Pandas, and both have the same PyTorch code base. Furthermore, all the data were in-memory prior to deep learning workload, so there was no overhead in loading data to create minibatches. The experiments conducted on CPUs scaled well across multi-nodes, but we observed a slight memory overhead causing the application to scale below the ideal point. We conducted more experiments to evaluate if there was an overhead from the data engineering framework, but we observed no significant overheads causing less scaling on CPUs. Figure 8.18 highlights the single process and distributed experiments carried out on CPUs. We used PyTorch built 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 an MPI backend for distributed computation.

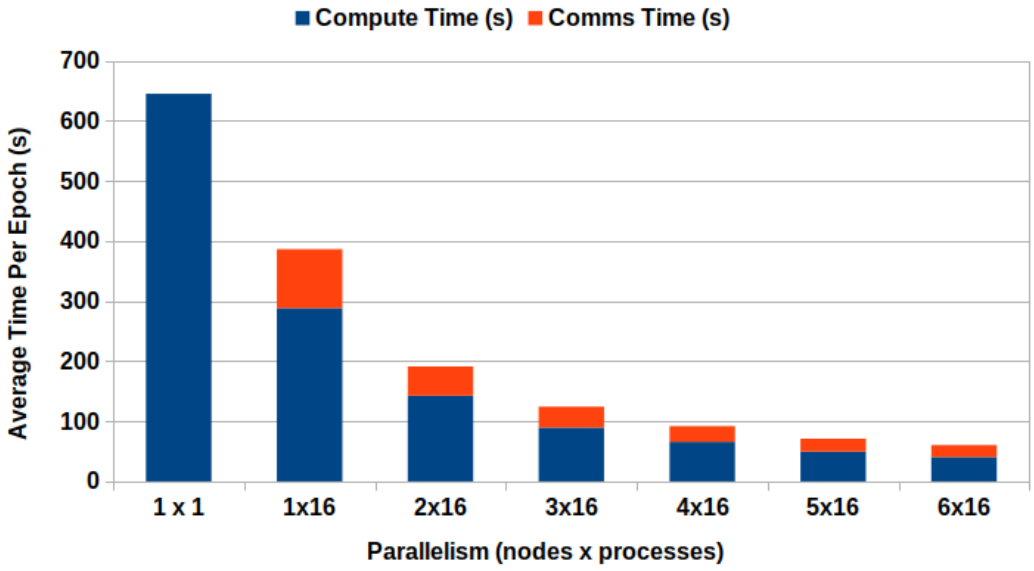

Figure 8.18: Distributed Data Parallel Deep Learning on CPU

The GPU-based experiments were handled with a single-node multi-GPU experiment setting to see how the data analytics workload could be scaled on NCCL execution framework with PyTorch. Figure 8.19 displays the results for single GPU and multi-GPU experiments. We observed that the execution time was dominated by the communication time. With the increase of parallelism, the number of communications across devices increases, but the number of batches that has to be sent across devices lowers. This gives an advantage in scaling. When we consider the computation time, we saw that scaling happens closer to the ideal point of scaling in all parallel settings. In addition, the computation is much faster in Parallelism 2 compared to Parallelism 1 where the memory overhead is 50% less compared to the sequential execution. When considering CPU vs. GPU performance for the deep learning workload, the speed-up from GPUs is 2x compared to CPUs in this network.

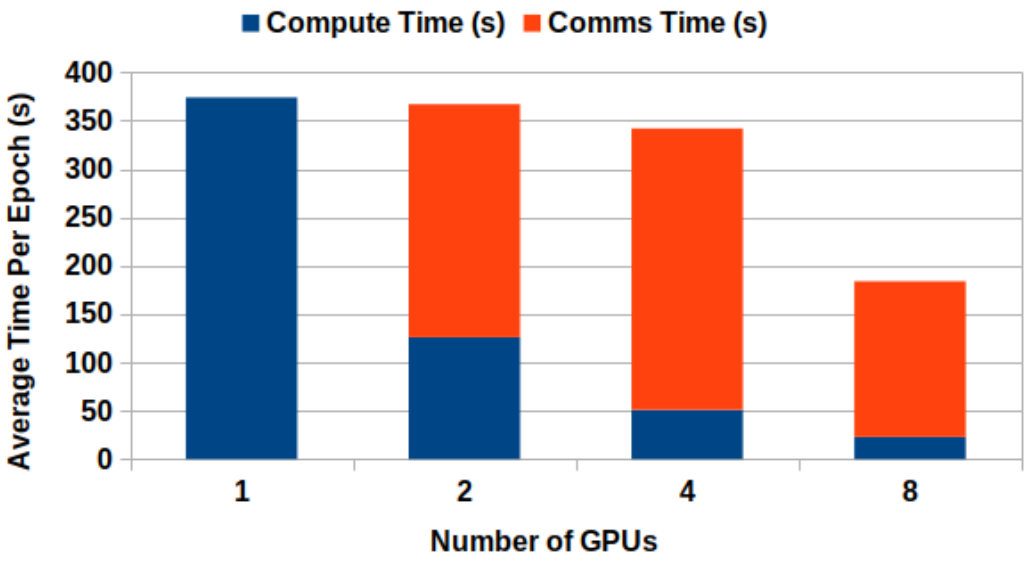

Figure 8.19: Distributed Data Parallel Deep Learning on GPU

CHAPTER 9

CONCLUSION

PyCylon currently contains over 40 dataframe operators with the capability of scaling up to multiple machines or run sequentially. Our research effort shows promise even in single process execution from our current micro-benchmarks on each data engineering operator compared to the currently predominant dataframe Pandas. Not only that, PyCylon scales well compared to Dask and Modin 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 dataframes by simply changing a few lines of code. Our dataframe is specially designed for HPC workloads and works well on HPC hardware, unlike the existing options. In terms of an external viewer, PyCylon is a dataframe for MPI, which was a missing component until now. Besides evaluating the framework just on operators, we also engineered it to fashion a real-world scientific workload. We designed an end-to-end data analytics-aware data engineering workload using PyCylon and obtained better results than the original implementation. Since the original was just sequential, we were also able to provide scalability for the application. Besides, we implemented the data engineering workload of this application using Modin, and the performance comparison shows that PyCylon is scaling much better than Modin. The significance of PyCylon is that it is not a framework to accelerate Pandas dataframe similar to Modin or Dask but to create an accelerated dataframe optimized for high-performance computing along with the usability of Pandas. These facts depict that our data engineering framework can be a promising tool for scaling data engineering workloads specifically on HPC and seamlessly integrate with machine learning and deep learning workloads for large scale applications.

CHAPTER 10

FUTURE WORK

Currently PyCylon supports about 40 data engineering operations and we are working on adding more operations to the dataframe API while improving the performance of existing operators. Besides, there are a few operators which are quite slower compared to existing data engineering frameworks and we are working on impoving those operators. The current version of the dataframe API doesn't have specific abstraction to represent columnar data. We are working on representing columnar data as a Series similar to Pandas. Additionally, the scientific work load we implemented can be benchmarked as an end-to-end application using a custom API written by using MPI operators. But we are planning on expanding this benchmark to be done using Sciml-bench[stf] and package this for easier usage using MLCube[MLC]. For advance application development we are working on exposing communication layer operators like Allreduce, Allgather, Broadcast, etc similar to what Pytorch is offering for tensors. Our objective is to be an advanced framework for data engineering and be compatible with frameworks like PyTorch in the highest level. At the moment we only focused on the CPU stack for distributed-memory parallel computation, but we are working on expanding our scope to GPUs by utilizing a BSP mode of execution and using Cudf to do local operations.

Considering a complete system, the next challenges we face are scaling better with dynamic computation requirements. Specifically, in using HPC resources in a cloud setting (auto-scaling), our main challenge is how to auto-scale with an MPI-backend. Because, MPI implementations are not designed for fault-tolerance. This must be externally provided or rely on implementations for this. From the MPI community there are some initiations for fault tolernace for MPI-workloads. This one prospect to deal with the problem. For scaling, we need to make sure the system

can pick-up where it left-off. This is an active research even for HPC-based deep learning at scale. Frameworks like PyTorch elastic and Horovod elastic are focusing on these problems. We belive that seamlessly integration to deep learning auto-scale would be beneficial and provide a good foundation to design a scalable system.

PyCylon is at the moment only support in-memory computation using a distributed-memory parallel approach. But when the data doesn't fit the memory, spilling to the disk becomes an inevitable outcome, in order to support applications with such requirements. The current execution model of PyCylon only relies on representing the data in-memory and if the frameworks needed to be extended to this scope, the Parquet (in-memory columnar representation for disk) support be provided for operations that need to be spilled to the disk. This requires, repogramming a generic layer to support all operators to run with disk an depending on the file-system configurations like shared-file system or not, we need to design operators to abstract away these details. But a known fact is spilling to disk, the operations become significantly slow because the main memory access time is quite low compared to that of disk access. To mitigate this issue, efficient disk data representation formats can be used. Specifically, NVMe (non-volatile memory express) are designed to access non-volatile memory rather efficiently using PCI-express interface connected directly to the CPU. This is relatively a new protocol to access non-volatile memory. Integrating such hardware optimizations to access disk faster can provide better performance when we need to spill operations to disk.

CHAPTER 11

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 over the past two years mainly focused on JVM-based high performance data engineering and analytics[AFK19, AFK$^+$]. We studied 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, an equivalent C++ implementation outperformed JVM-based implementations for large-scale data analytics with higher dimensionality. In the related research, we used BLAS and MPI libraries for high performance computation and communication. This motivated us to investigate more closely 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$^+$20]. Besides our deeper study on the internal workings of a system, major data analytic tools like PyTorch[PGM$^+$19] and Tensorflow[ABC$^+$16] showcase the necessity of standalone data analytics tools specialized for specific tasks. This is a scope beyond just big data computing. In addition, it illustrates the importance of seamless integration with the existing software stack for data analytics and making compatible data engineering systems 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, classic data analytics platforms on low performance Python stack slowly when 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 revealed that 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 how existing data engineering frameworks in both Java and Python are not scaling well in high performance computing environments. Furthermore, our preliminary findings on this also revealed that a high performance Python approach with C++ shows better scaling than existing state-of-the-art systems. This indicates 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. Initial benchmarks and API definitions show that our proposed method is one of the most prominent distributed memory dataframes in existence at the moment. The existing parallel dataframes on CPUs are entirely written on Python, which inherently imposes limitations in scaling. Our preliminary research and benchmarks provide evidence[APW+20]. By designing a high performance distributed memory dataframe, we can offer better scalability and match up with 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 focused our attention on Pandas, since it is the most advanced 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 adapted to many scientific problems to improve the performance of existing data analytics and data processing. However we observed that data engineering as a research problem has not been well studied or researched with a functional data engineering system. Our research specifically focused on the deep level of avoiding memory copying between programming kernels and user-space. Also, we have emphasized retaining performance and providing usability at the same time. We 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 provides the 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:* A high performance distributed dataframe API and a seamless integration to numerical

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 while retaining high performance on the CPU stack and transfer data seamlessly to the 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/. (Accessed on 06/07/2020).

[Donb] Jack Dongara. Lapack: ddot. http://www.netlib.org. (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.

[HSY+20] Benjamín Hernández, Suhas Somnath, Junqi Yin, Hao Lu, Joe Eaton, Peter Entschev, John Kirkham, and Zahra Ronaghi. Performance evaluation of python based data analytics frameworks in summit: Early experiences. In *Smoky Mountains Computational Sciences and Engineering Conference*, pages 366–380. Springer, 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.

[MLC] Mlcube — mlcommons. https://mlcommons.org/en/mlcube/. (Accessed on 02/08/2021).

[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 Wickramasinghe, 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.

[stf] stfc-sciml/sciml-benchmarks-prerelease: A suite of machine learning benchmarks for ai for science. `https://github.com/stfc-sciml/sciml-benchmarks-prerelease`. (Accessed on 06/03/2021).

[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.

[TSM+20] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[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, Shivaram 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

2021	M.Sc., Luddy School of Informatics, Computing and Engineering Indiana University Bloomington United States
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 Wickra-

masinghe, 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, Ge-

offrey, *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/*]