Indiana University Bloomington

Indiana, United States

ROLE OF HIGH-PERFORMANCE COMPUTING IN DEEP LEARNING

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Pulasthi Supun Wickramasinghe

2021

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

This dissertation proposal, written by Pulasthi Supun Wickramasinghe, and entitled Role of High-Performance Computing in Deep Learning, having been approved in respect to style and intellectual content, is referred to you for judgment.

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

Minje Kim

Prateek Sharma

Xiaozhong Liu

Geoffrey Fox, Major Professor

Date of Proposal:

The dissertation proposal of Pulasthi Supun Wickramasinghe is approved.

Martin Swany
Luddy School of Informatics, Computing and Engineering

Raj Achariya
Dean of the University Graduate School

Indiana University Bloomington, 2021

To my wife Roshani, and our parents

Acknowledgements

I have been fortunate to be among a set of extraordinary people who have been supporting and guiding me throughout the years dedicated to this research.

I would like to express my deepest appreciation and gratitude to my advisor Prof Geoffrey Fox for all the guidance and knowledge that he shared throughout my research. His remarkable knowledge and experience in parallel and distributed computing and foresight into future trends made this research a possibility and a reality. He is also one of the kindest professors I have come across in my life. It has been a privilege to have worked under his guidance throughout these years.

I would like to thank my research committee members Prof Minje Kim, Prof Prateek Sharma and Prof Xiaozhong Liu, for their valuable guidance and input that made this dissertation possible. I would also like to thank Prof Judy Qiu, who provided guidance and support throughout my research and for being part of my advisory committee.

I also owe my gratitude to Dr Sanjiva Weerawarana and Dr Shahani Markus for the initial guidance and support they provided when I embarked on pursuing my PhD. I would like to extend an especial thanks to Dr Saliya Ekanayake, and Dr Supun Kamburugamuve, whom I had the privilege of working with throughout my research on many research projects, their guidance as fellow researchers have been invaluable for the success of my research. I would also like to thank Dr Jerome Mitchell, who I was fortunate to work alongside under the same advisor during his research years, for his support. I would also like to extend an especial thanks to Vibhatha Abeykoon, whom I am fortunate to have worked alongside on many research projects under the same advisor.

I would like to thank all the other member of the Twister2 project, including Dr Kannan Govindarajan, Dr Ahmet Uyar, Dr Gurhan Gunduz, Chathura Widanage,

Niranda Perera, who have been major contributors to the Twiste2 project which laid the foundation that enabled my research. I would also like to extend my gratitude to all the members of the Digital Science Center, including Gary Miksik, Allan Streib, for the administrative and technical support provided to make my research a success.

I have been fortunate to have a wonderful set of friends who have made my time in Bloomington a pleasant and memorable one. I would like to thank all of my friends in Bloomington for this. I would also like to extend my thanks to all my close friends who have supported me on my journey in various ways, and a special thanks to team CUPPA, which comprises my closest friends who were also my research colleagues during my undergrad years.

I cannot put to words the gratitude and love I have for my parents, who supported and inspired me throughout my life which has led me to this moment. They instilled so many great qualities in me that has allowed me to persevere through this journey. Without their dedication and sacrifices, I would not have been able to come this far.

None of this research would have been even possible if not for my lovely wife Roshani, who has been with me throughout all the highs and lows of my graduate life. Her love and strength allowed me to face and get through all the obstacles that I faced over the years.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

MOTIVATION

The motivation for looking into the role of HPC in deep learning can be viewed from the two major components. The first is to have a high performance distributed data pre-processing framework which is developed using core HPC principles around efficient communication operators and task scheduling strategies. In order to be effective, these highly optimized core operators need to be wrapped and presented using easy to use high-level programming interfaces. While data pre-processing does not have the glamour and interest that attracts people to deep learning and machine learning, it is an integral step before more advanced deep learning and machine learning programs can be applied to the data. In some cases, this step may be where the bulk of the time is spent. Having an easy to use high performance distributed data pre-processing framework allows users to quickly do all the data cleaning and transformation steps that needed to be done so that they can be used as inputs to deep learning/ML algorithms.

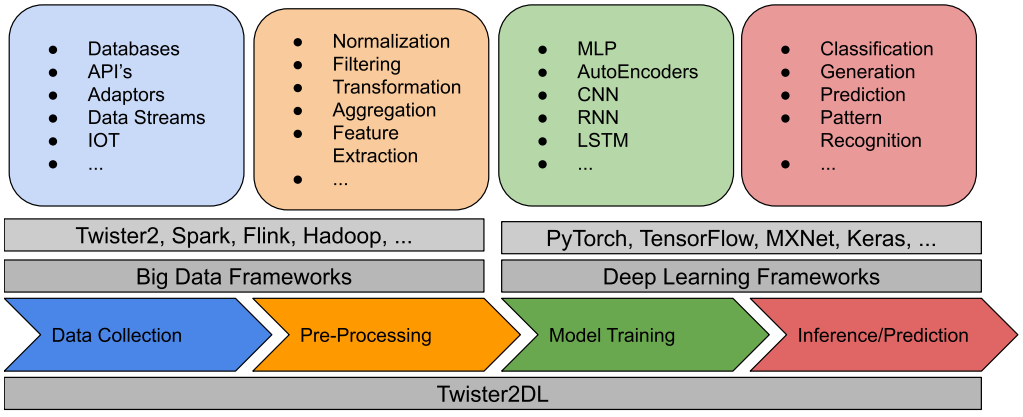


Figure 1.1: Distributed data processing vs deep learning frameworks and what areas each address

Once all the data pre-processing is done, machine learning and deep learning algorithms need to be applied to extract useful information and patterns from the

data. For machine learning, it has been proven that leveraging the primitive operations provided by the distributed data processing frameworks to develop parallel implementations of ML algorithms can be done efficiently. With a more optimized set of core operators, such ML programs will also see higher performance numbers. For deep learning, this has not been the case to a vast extent. Developing, training and inferencing deep learning models are mostly done through a set of deep learning frameworks such as Caffe[JSD+14], Keras[GP17], PyTorch[PGM+19], etc. that have been optimized for deep learning models. These provide easy to use high-level interfaces which can be used to develop complex deep learning models and then leverage CPU's and/or GPU's to execute. While some such frameworks do provide the ability to train/run, deep learning models in a distributed manner, they are not well suited for the pre-processing data; hence that needs to be performed using distributed big data frameworks. This means processed data needs to be migrated between different frameworks. Figure 1.1 show how the current framework echo system is divided between distributed data processing frameworks and deep learning frameworks. In order to have an efficient end-to-end workflow, it is important to have seamless integration between these two framework echo systems. This will allow developers and data scientists to efficiently build a complete data processing and analysis pipeline without having to move intermediate data between frameworks and possibly compute clusters.

It is also important to note that even with the current state of the art frameworks such as PyTorch, setting up distributed training and evaluation can pose major technical challenges and require expert knowledge to get done correctly. This leads to the motivation of having a seamless integration of deep learning frameworks with distributed data processing framework. If both data pre-processing and deep learning can be done through a single framework, it would make the end to end data

analytics process much easier.

1.1 DAMDS

One of the main motivations for extending the Twister2 framework to support DeepLearning came from the DAMDS[RF13] application that was utilized to analyze and cluster gene sequence data by the authors. DAMDS is a parallel Multi-Dimensional Scaling (MDS)[Kru78, BG05] algorithm. In essence, MDS is a non-linear optimization problem that tries to optimize the mapping in the target dimension based on the original pairwise distance information. Since it works on pairwise distances, DAMDS can be broadly applied to many datasets. This is especially important in gene sequence data analysis since such datasets do not have associated feature vectors. Pairwise distance, on the other hand, can be calculated using sequence alignment algorithms such as Smith-Waterman[SW$^+$81]. DAMDS is a complex algorithm with computation and memory complexity of $O(N^2)$ where N is the number of data points. Therefore using DAMDS on even modest data sets require a considerable amount of distributed computing power. Results discussed in section 4.6.2 shows that a DAMDS implementation executed on Twister2 framework runs on par with highly optimized OpenMPI implementation. DAMDS takes in a distance matrix of size NxN as an input, which means the raw input data such as gene sequence data needs to be pre-processed to generate the input distance matrix. Even with a highly optimized parallel implementation of DAMDS it is still not practical to run it for datasets with millions of data points.

1.2 Autoencoder based MDS

In order to analyze gene sequence data with DAMDS, a considerable amount of pre-processing needs to be done. For a gene sequence dataset of N sequences, N^2 Smith-Waterman calculations need to be performed. Smith-Waterman algorithm itself has a complexity of $O(ML)$ where M and L are the lengths of the gene sequences being aligned. For example, the fungi gene sequence dataset analyzed by the authors contained 170K sequences, which meant $\sim$14.5 Billion Smith-Waterman calculations needed to be done to generate the input distance matrix for the DAMDS algorithm. Such large data pre-processing requirements are not uncommon in distributed data processing pipelines. Because of the high resource requirements of DAMDS and the resource constraints surrounding DAMDS, the authors looked into the applicability of deep learning to perform multi-dimensional scaling for large datasets. In [WF] we propose an multi-dimensional scaling algorithm based on autoencoders for gene sequence data. One of the main challenges with applying an autoencoder based solution for gene sequence data is the nature of the data itself. First, gene sequence data is typically presented in RNA format, where each sequence is a string of characters. The character alphabet for RNA sequences are A, T, G and C. Secondly, sequences in a data set can vary in length significantly. In order to train an autoencoder, the input needs to be of fixed size numerical vector. Therefore raw gene sequence data needs to be converted into fixed-length numerical vectors which accurately represent the original gene sequence.

1.2.1 Adapting gene sequences for Autoencoders

The conventional method to convert gene sequences into fixed-length numerical vectors is to use either One Hot Encoding (OHE) or ordinal encoding. In [WF] we

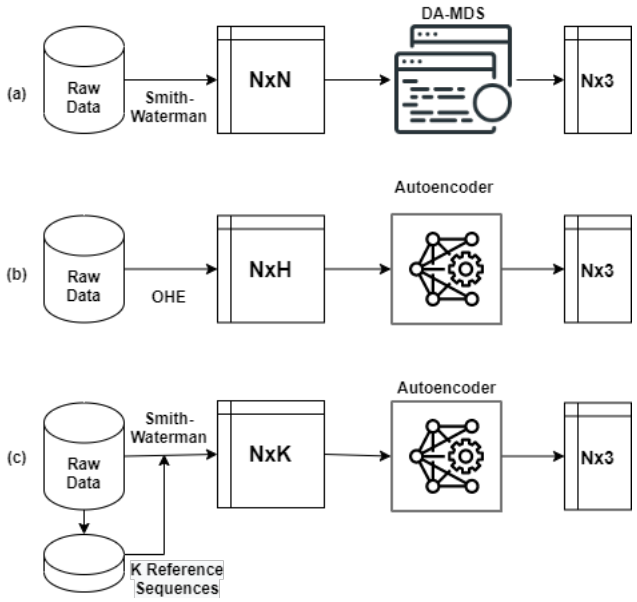


Figure 1.2: Dimension reduction approaches. (a) DAMDS. (b) Autoencoder with OHE, "H" is the length of the vector once it is encoded with OHE, (c) Autoencoder with reference sequences, "K" is the number of reference sequences

introduce an additional approach based on pairwise distances to generate a representative numerical vector. Later evaluation results show that this method generates better results compared to OHE based approach.

One Hot Encoding

In OHE, the character alphabet of the sequence is represented by a vector. The vector has a length that is equal to the number of characters in the alphabet, and the value of the position representing a single character will be '1' the other location will be '0'. For example, the 'A' of the 'ATGC' alphabet in RNA sequences is represented as [1,0,0,0]. In order to make the length of the OHE vectors equal for a given dataset, the vectors need to padded with '0'. One drawback of the OHE approach is that while it does create a fixed-length vector (with padding), it would vary between datasets. OHE also does not encode any biological distance data related to the gene sequences, which is an important aspect when performing

dimension reduction on gene sequence data. The resulting input vector will have a length of '*Alphabet Size * Length of longest sequence*'.

Ordinal Encoding

In ordinal encoding, each character in the alphabet is assigned a specific numerical value. To construct the numerical vector, the sequence values are replaced with the assigned numerical value. For example, characters "A,T,G,C" can be assigned values "0.25, 0.5, 0.75, 1.0" respectively. This would mean the encoded result of the sequence "GGTAC" would be [0.75, 0.75, 0.5, 0.25, 1.0]. This approach will produce a smaller vector since the length of the numerical will be equal to the length of the longest sequence in the dataset. One major issue with ordinal encoding in the context of deep learning is that during training, the network may give a higher significance to characters assigned with higher numerical values, while in the biological sense, there is no such significance to those characters. Therefore out of OHE and ordinal encoding, OHE presents a more accurate representation for gene sequence encoding.

Pairwise distance - K Reference encoding

The aim of the novel approach we introduced in [WF] for gene sequence encoding for multidimensional reduction was to generate a fixed-length numerical vector that encapsulates the biological distance data of sequences. In this approach, the representative numerical vector is calculated by first selecting 'K' reference sequences and then calculating the Smith-Waterman distance to a given sequence and the 'K' reference sequences. This results in a numerical vector of length 'K', which contains biological distance information of the dataset embedded within it. The correct value for 'K' would depend on the dataset and the structure of the dataset. A logical

estimate of K would be “10xC” or “20xC,” where “C” is a rough estimate of the number of clusters in the dataset.

Figure1.2 summarizes how the different approaches for multidimensional scaling operate and the intermediate data structures that are generated in the process. Figure 1.3 shows the heat-maps generated for each approach. The heat-map plots the comparison between the original ”Smith-Waterman” distance and euclidean distance in the projected 3 dimensional space. As seen in the heat-maps the ’K Reference’ based approach generates heat-maps that have data points more concentrated along the diagonal, which is the desired outcome. Based on the heat-maps ’K Reference’ based autoencoder solution produces results that are slightly better than DAMDS based approach. The findings show that the autoencoder based solution is able to produce results similar to DAMDS with significantly lesser compute resources. This solution does not contain $O(N^2)$ memory or computation complexities allowing it to be scaled to millions of data points. The proposed solution is based on a set of K reference sequences, which are used to create the input vector for the autoencoder.

Implementing the solution proposed by the authors in [WF] requires two main stages. The first is a data pre-processing step where a representative vector for each gene sequence is generated based on the K reference sequences. This means that for each gene-sequence, K Smith-Waterman calculations need to be performed. To put this in perspective for the 170K sequence dataset with K=1000, pre-processing needs to calculate 170 million Smith-Waterman calculations (opposed to $\sim$14.5 Billion in DAMDS pre-processing). While the amount has been reduced drastically for larger datasets, the pre-processing step also needs to be distributed. In the second stage, the calculated input vectors need to be used to train an autoencoder. This also needs to be executed as a distributed application for large datasets. In such an application, the need for seamless integration between distributed data processing

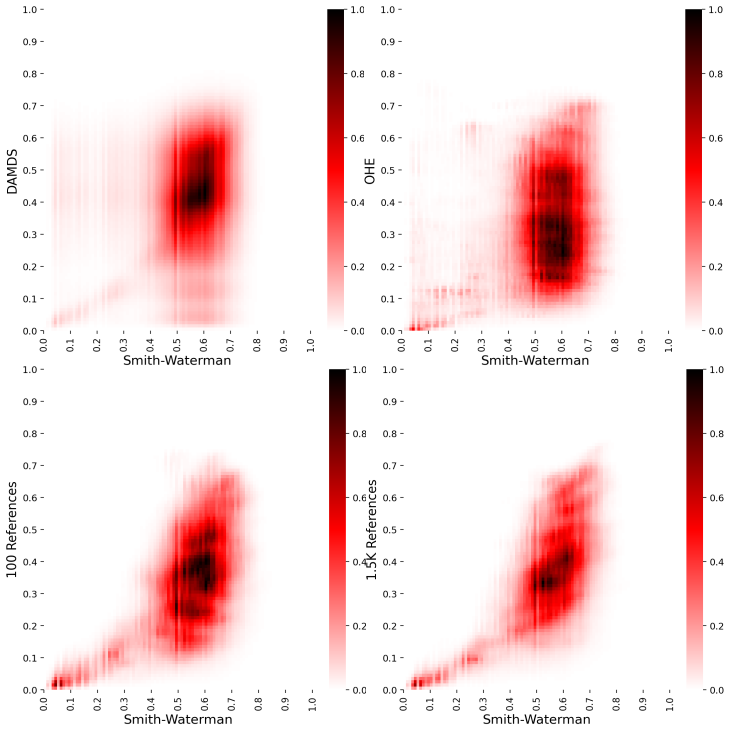


Figure 1.3: Heatmaps of Smith-Waterman distance vs projected distance in 3D

and distributed deep learning is clear. Without such an framework, the first stage needs to be executed on a big data framework like Twister2[KGW+17] and the deep learning stages need to be done in PyTorch or a similar deep learning framework. The intermediate data generated needs to be moved between frameworks that typically involve file storage. In the solution presented in [WF] experiments need to be performed for varying K values and multiple runs for each individual K (to run using different random samples of K reference sequences). This means for each run; two frameworks need to used and data needs to moved between the frameworks. Which can add up to a considerable amount of time spent on non-productive tasks. Figure 1.4 shows visualisations (in 3D space) of dimension reduction achieved using DAMDS and 1.5 shows the results from the proposed autoencoder based MDS solution on a 170K gene sequence dataset. While the positions of the clusters may look different, the clustering of data points is similar in both cases. This is discussed in

more detail in [WF].

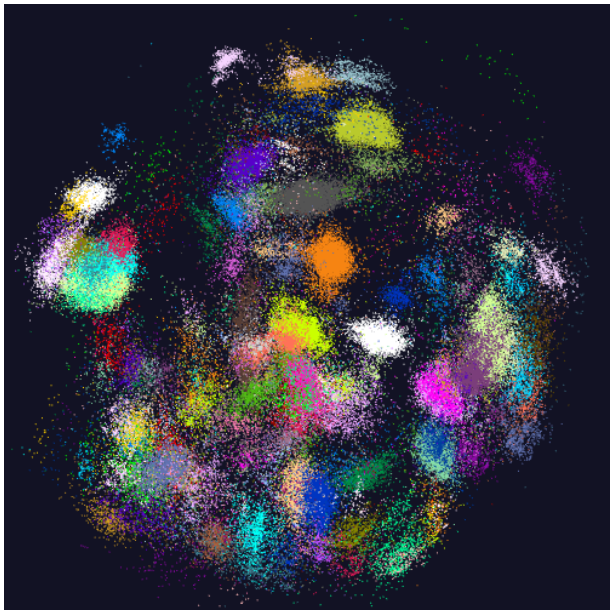


Figure 1.4: DAMDS 170K points projected to 3

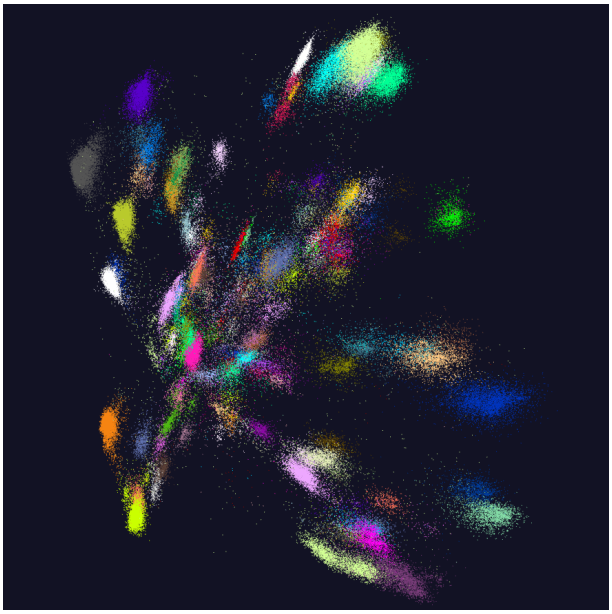


Figure 1.5: 1K reference sequences, 170K points projected to 3D

CHAPTER 2

LITERATURE REVIEW

There is a plethora of research and development being done around the big data/HPC domains and machine learning and deep learning domain. While some work is more on specific use cases, some are working on more general frameworks and optimizations that can be applied to a wide range of problems. Below we focus on related work in regard to two major areas that are developed and discussed. The first is work-related to distributed dataflow frameworks, which is the high level distributed data processing layer that is discussed in this paper. Then related work for distributed deep learning frameworks are discussed.

There is no specific definition for dataflow in the research literature; the term *dataflow* is used to denote various versions of the same underlying model. The dataflow process network presented by Lee et al. [LP95] can be to some extent seen as a formal definition of the dataflow model. A layered dataflow model was introduced by Misale et al. [MDAT17] which was built on top of [LP95] to represent big data processing frameworks that follow the dataflow paradigm. In their work, they break down big data frameworks into three layers which as a whole represent the entirety of the dataflow model. They then go onto show the mapping of each presented layer into components that are present in several popular frameworks. This mapping also holds true for Twister2, where those three layers can be loosely mapped to the communication, task and data layers. How the dataflow model is adopted by the popular Google cloud dataflow framework is described by Akidau et al. [ABC+15] including implementation details of the framework. Apache Spark[Spa] develops a dataflow graph with an additional linage graph that is logged and tracked to achieve fault tolerance.

Naiad[MMI+13] introduced the timely dataflow model system for iterative and incremental distributed computation processing; timely dataflow is a slightly different and more complex interpretation of the dataflow model. Naiad presents a stateful dataflow model, where the edges of the dataflow graph carry unbounded streams of data over dataflow nodes containing mutable state. Dryad[IBY+07] is a high-performance distributed execution engine that is suitable to run coarse-grained data-parallel programs as acyclic dataflow graphs for execution; the user has to program the dataflow graph in Dryad explicitly. In general, a Dryad application combines computational vertices with communication links to form a dataflow graph. In[Hie] the authors show how coarse-grain parallelism of an application can be leveraged and used through Hierarchical Task Graphs. The main objective of the technique is to provide a program dependency representation without a cyclic dependency. Cyclic dependencies are hidden from the user by encapsulating them within a supernode. Turbine[WAM+13] is a distributed many-task dataflow engine that evaluates program overhead and generation of tasks. During program execution, the system breaks parallel loops and invocation of concurrent functions into smaller fragments and execute them parallelly using Twister2; the explicit dataflow programming is hidden from the user, which is different from Dryad and similar to the Turbine model.

The relationship between dataflow and MPI at an operator level is not well-defined within the research literature to our knowledge. There has been work done in this area that discusses the role of dataflow for parallel programs such as MPI applications. OpenStream, which is a dataflow extension to OpenMP was introduced by Pop et al. [PC13], and they discuss the advantages of such a model. The importance of understanding dataflow within MPI programs is discussed by Strout et al. [SKH06] they also introduce a dataflow analysis framework for an MPI

implementation.

There has also been some effort in the big data domain to unify many frameworks through a single API. Apache Beam[apa] developed as the programming model for Google Cloud Dataflow[ABC+15] addresses this issue and strives to provide users with a single API that can be used to develop parallel data processing applications. Apache Beam has been widely adopted in the industry, with a large user base taking advantage of its uniform API which supports multiple back-end distributed processing frameworks. Apache Beam can be extended to support new Distributed data processing back-ends by implementing a component named beam runners. Li et al. [LGM+18] discuss the challenges faced when developing an Apache Beam runner for IBM streams[HAG+13].

Starting with the initial deep learning frameworks such as Torch[Tor], Theano [ARAA+16], Caffe[JSD+14], deep learning has seen the introduction of many frameworks, each developing on top of the previous frameworks. Many of the early frameworks did not support distributed execution, and in some, distributed execution support was added later. Caffe2[cafa], Pytorch[PGM+19], TensorFlow[AAB+15] and MXNet[CLL+15] are among the most widely adopted deep learning frameworks. Caffe was extended to Caffe2, which supported distributed training. Pytorch introduced distributed data parallel mode which allowed Pytorch to be run on multiple machines. TensorFlow allowed distributed training through distributed strategies or through Horovod[SDB18]. Among the many available deep learning frameworks, PyTorch and Tensorflow are the most used frameworks and account for most of the use in academia and industry. Deeplearning4j[dee] is a java based distributed deep learning framework that runs on the JVM. While many of the well known deep learning frameworks do support distributed training, it can sometimes be technically challenging to set up and may discourage researchers/developers who are not used

to orchestrating a cluster on their own from using the distributed features. Because these frameworks are targeted towards optimizing deep learning model training, they do not provide the capabilities provided by big data frameworks to pre-process data in a distributed fashion.

The BigDL[DWQ+19] framework developed by Intel to run on top of Apache Spark address this need. By providing a built-in deep learning framework, users can do the data pre-processing, and machine learning workloads using Apache Spark API's and seamlessly move data into deep learning models developed using the BigDL framework.

CHAPTER 3

INTRODUCTION

With the proliferation of data that is available to be processed and analyzed in recent years, the demand for frameworks and algorithms that can be used to infer useful information from raw data has also proliferated. Machine learning algorithms and, more notably, deep learning/neural networks started to play an ever-increasing role in the big data world to process and analysis the data and to extract valuable information from raw data in every thinkable field and industry, from agriculture to finance. More and more machine learning algorithms and optimized versions of algorithms were introduced to tackle the growing demand for data analysis. As the volume of data increases, requirements that data processing frameworks/algorithms need to meet in order to cater to these requirements also increase. As compute, and memory requirements that are needed to process the amounts of data available quickly exceeded the capabilities of a single machine or computing unit, parallel and distributed algorithms needed to be utilized to scale-out data processing and analysis to 100's if not 1000's of machines.

3.0.1 Big Data Frameworks

With these requirements and the introduction of the map-reduce framework, a large number of distributed data processing frameworks such as Apache Hadoop[Whi12], Apache Spark[ZCD+12], Twister2[KGW+17], Apache Fink[CKE+15], etc. gained popularity because of their ease of use and simple programming interfaces and most importantly their ability to scale into large computing clusters to run programs in parallel. In addition to providing easy to use high-level programming interfaces to develop distributed data processing applications, these frameworks also developed

machine learning libraries on top of the distributed platforms and provided the ability to use parallel implementations of popular machine learning algorithms such as KMeans, SVM, etc. out of the box. As the data processing requirements kept increasing, data processing frameworks needed to increase in performance to make sure they performed efficiently on the available compute resources. Improvements that were made in recent years have been two-fold; first, most frameworks and new frameworks that address shortcomings of existing frameworks have worked towards improving and optimizing the core distributed building blocks such as efficient communications and improved task scheduling and fault tolerance. Resulting in more efficient execution of machine learning algorithms on distributed environments. Secondly, vast amounts of research have been done on developing more efficient parallel implementations of machine learning algorithms to allow such algorithms to process and analyze more and more raw data. While these research efforts have improved the existing systems, it is clear that there is more work to be done and further optimizations that can be achieved.

The High-Performance Computing (HPC) domain has been developing, fine-tuning and perfecting the mechanics and strategies that need to be used to maximize the performance of parallel and distributed programs and algorithms for the past several decades. However, developing parallel programs and algorithms using the tools available in the HPC domain, such as MPI[SGO+98], is non-trivial. While machine learning algorithms developed using MPI implementations such as OpenMPI, vastly outperform implementations available in big data frameworks such as Apache Spark [KWEF18], they are much harder to develop and require expert knowledge in the domain. Due to the ease of using big data frameworks, they have gained wide adoption in both industry and academia, even though they provide sub-par performance in many cases. This led to the initial motivation to develop Twister2[KWG+18],

which is a high-performance data analytic framework built with core HPC principles and know-how, which enabled it to perform on par with frameworks such as OpenMPI while providing a high level, easy to use programming interface for the end users/developers. Another important aspect that needs to be noted is that most if not all distributed big data processing frameworks are built around the dataflow model as opposed to the control flow model that is widely adopted in applications and programs in the HPC domain. Dataflow is viewed as the model best suited for tackling large sums of data for processing and analysis both in stream processing and batch processing and has seen wide adaptation.

3.0.2 Machine Learning and Deep Learning

While distributed data processing frameworks provide the capability to utilize large compute clusters to process data and play an important part in the big data echo system, machine learning and deep learning are the mechanisms that allow useful information to be extracted from the raw data. Without efficient distributed implementations of both machine learning and deep learning algorithms, distributed data processing capabilities would be of little use. A vast amount of research has been done and is being done on how to efficiently run machine learning algorithms on data processing frameworks. From the data processing frameworks end, it is essential that easy to use high-level programming API's are provided so that engineers and scientists can write machine learning algorithms without the need to understand the underlying distributed computing concepts in great detail. Big data frameworks like Apache Spark[ZCD+12] have introduced rich machine learning libraries (MLlib[MBY+16]), which can be easily used out of the box to address such requirements. More recently, the Deep learning models have started to replace tra-

ditional machine learning algorithms for various use cases because of the superior performance they tend to exhibit with the ability to consume vast sums of data.

With the rapid adaptation of deep learning in many domains, deep learning has slowly become the go-to solution for many machine learning problems. Mainly because of its ability to provide solutions to a wide range of problems. Deep Learning is now used in a wide range of applications that can be span from small experiments that may run on a single machine or a single GPU device to a large scale where the training and inference of the deep learning model may span multiple machines or employ supercomputers. These large models may contain billions of parameters and require terabytes of data to train. Frameworks such as PyTorch[PGM+19], TensorFlow[ABC+16], Caffe[JSD+14], etc. have allowed deep learning models to be developed in a distributed manner. In addition to optimized software frameworks, the vast adaptation of deep learning has prompted the design and development of specialized hardware components which are able to run deep learning computations extremely efficiently. The two most well-known hardware components are GPU's and TPU's; while GPU's are more general purpose and have been around for some time, TPU's have been designed specifically to handle deep learning computations. It is important to note that while GPU's and TPU's are able to achieve several times the performance of traditional CPU's these hardware devices are more expensive and generally not as prevalent as CPUs. Therefore having the ability to run machine learning/deep learning applications on traditional CPU's is an essential requirement and may be financially favourable in some cases.

When considering real-world machine learning/deep learning applications, most are comprised of two major components,

- Data collection and pre-processing/Machine Learning
- Deep Learning/Model training and inference

Data collection and pre-processing can be handled using big data systems such as Spark[ZCD+12], Twister2[KWG+18], Flink[CKE+15], etc. And training and inference can be handled using deep learning frameworks such as PyTorch[PGM+19]. However, with the adaptation of deep learning to solve many problems, many use cases where deep learning needs to be applied to big data processing pipelines have emerged. For such use-cases, it is vital to have an end-to-end solution that can handle big data analysis as well as deep learning while keeping the performance of such a system in mind. In order to have an efficient deep learning application, both the data pre-processing and deep learning components need to be efficient. In some use cases, the bulk of the application run time would be taken up by data pre-processing since it can be a highly time-consuming task.

With many machine learning applications being converted or replaced by deep learning applications, it has become more and more important to facilitate easy to use deep learning frameworks which can integrate with the data processing pipeline. One such machine learning application area that motivated some of this proposed work is Multi-Dimensional Scaling(MDS). MDS is an important tool that allows researchers and scientists in many areas such as biology to understand higher dimensional data by projecting them into lower dimensions where they can be visualized. However, even with the existing state of the art distributed algorithms such as DAMDS[RF13] it has been difficult to scale beyond a couple hundred thousand data points because of the massive memory requirements of their algorithms. Preliminary results obtained by doing MDS using deep learning models have shown that similar results to DAMDS[RF13] can be obtained using deep learning models that incur a fraction of the memory requirement. However, the bulk of the runtime of this deep learning-based MDS application is spent on complex data pre-processing steps. In such use-cases having the ability to implement both the pre-processing and

deep learning in a single framework where the data flows seamlessly between the two stages while providing excellent performance would be highly desirable. Such applications provide both the motivation and the evaluation foundation for the ideas proposed in this document.

The first half of the puzzle to provide efficient big data analysis is addressed using Twister2 TSet's[WKG+19], which provide a high-performance iterative dataflow framework, TSet's employ knowledge from High-Performance Computing(HPC) to provide a more efficient alternative to frameworks such as Apache Spark[ZCD+12] and Apache Flink[CKE+15]. Additionally, the integration of Twister2:TSet's with Apache Beam[apa] as a distributed back-end processing engine allows users to run Apache Beam pipeline jobs on top of the high-performance dataflow back-end without any changes to the existing code.

The next challenge is to provide seamless integration between data pre-processing and the deep learning portion of the application. BigDL[DWQ+19] provides such a framework for Apache Spark, BigDL[DWQ+19] allows users to develop deep learning applications on top of Apache Spark. However, this inevitably faces performance and efficiency issues of the underlying Apache Spark framework, which have been addressed in Twister2:TSet's. Therefore it is important to extend Twister2:TSet's to support deep learning applications that would provide an efficient end-to-end solution for deep learning applications that need to be integrated into the big data analytic workflow. Having an end-to-end solution would allow developers to develop the needed deep learning applications quickly and run them on existing compute clusters such as Kubernetes[BGO+16], Nomad[nom], Mesos[HKZ+11], or use Slurm[YJG03] to run on supercomputers and leverage high speed interconnects to increase performance.

CHAPTER 4

DISTRIBUTED DATA PROCESSING/MACHINE LEARNING

In order to tackle the issue of processing large sums of data, it is important to have a high performance distributed data processing framework, which also provides easy to use high-level programming interface. The ecosystem that has been built around the big data ecosystem is vast. Frameworks such as Hadoop[Whi12], Spark[ZXW+16], and Flink[CKE+15] focus on batch processing; Storm[TTS+14], Heron[KBF+15], and Flink[CKE+15] target stream processing; while TensorFlow[ABC+16] and PyTorch[Ket17] are for machine learning and deep learning. Frameworks such as Apache Beam[apa] provide a unified API for both batch and stream processing and support many of the above-mentioned frameworks as data processing engines underneath. In addition to the frameworks mentioned above, there are a large number of frameworks developed by both academia and industry that provide optimizations and specializations while hiding the underlying complexities of parallel and distributed computing. One important takeaway from studying these frameworks is that most of them are designed around the core dataflow model and differ in how the implementation of the core dataflow model concepts are done at each level. Most successful frameworks in this domain have developed easy to use high-level programming interfaces so that even users with very little understanding of distributed computing can write efficient distributed applications. The runtime system takes the responsibility of dynamically mapping the dataflow graph into an execution graph and executing it on a cluster as efficiently as possible. Most of these dataflow system's have come from the database and big data research communities and have not employed the vast amount of knowledge in the HPC community on distributed computing at the core of their designs. In[KWEF18] the authors studied this topic in more detail, and their findings motivated the development

of Twister2[twi, KGW+17], which is a data analytics framework for both batch and stream processing. The goal of Twister2 is to provide users with performance comparable to HPC systems while exposing a user-friendly dataflow abstraction for application development. Twister2 was able to achieve significant performance improvements when compared to the state of the art big data frameworks such as Apache Spark and Apache Flink, using highly optimized distributed communication operators[KWG+18]. This highly-optimized communication layer paved the path to develop a high-performance dataflow abstraction that serves as a high-level programming language interface for the developers/scientists. This abstraction, named TSet, provides functionality similar to other popular frameworks such as RDD API in Apache Spark and DataSet API in Apache Flink. TSet's are the solution presented in this paper to address efficient distributed data processing, which is the first major component that needs to be addressed to understand the role of HPC in deep learning. To understand the influence of HPC in the design and development of TSet's and the challenges that need to be addressed when applying HPC domain knowledge, one needs to compare and contrast the dataflow model and MPI (MPI has been established as the defacto standard in HPC).

4.1 Comparison of Dataflow and MPI

MPI can be considered the defacto standard used in the HPC domain due to its wide adaptation in the domain. In most cases, applications and programs developed in the HPC domain are built around the Bulk Synchronous Parallel (BSP) model. BSP can be loosely defined as coarse-grained parallelism, which performs barrier synchronizations using inter-process communications. MPI specification is well suited to implement BSP-style programs. However, since the primitives provided by MPI implementations such as OpenMPI are very low-level communications

operators, it requires expert knowledge in parallel computing domain to develop applications/programs using them. Collective operations such as gather, reduce, allreduce, allgather, scatter, etc., are well known highly optimized communications primitives specified by MPI and implemented by frameworks such as OpenMPI. These collective operations have been researched and optimized throughout the past couple of decades in the HPC community to a great extent. However, it is not possible to directly use these primitives in a dataflow model because of several key differences. Dataflow collectives are driven by the following requirements that make them slightly different from MPI specification-based collectives

- A detailed view of the dataflow model for batch and stream processing in Twister2.
- A more efficient way of handling iterations for dataflow framework with Twister2 TSet API.
- An evaluation of the presented framework to showcase its expressiveness and performance.
- A connected dataflow model for big data frameworks
- Review of extensibility and compatibility of TSet's to other frameworks.

Twister:Net[KWG+18] introduced a set of collective communication operations that correspond to collective communications in MPI specifications while also supporting the requirements stated above. It was also able to achieve comparable performance to OpenMPI. MPI specification based frameworks such as OpenMPI provide the bare minimum requirements to implement an efficient parallel application but leave the heavy burden of thread and data management to the user. Developing and dataflow model on top which hides all that information while preserving performance which can compare with MPI is challenging since each layer of

abstraction inadvertently adds more overhead to the system. TSet's is an attempt to achieve this using Twister:Net as the foundation.

4.2 Twister2 Framework

Before looking at the Twister2 dataflow model, it is important to understand the Twister2 framework and how it is structured. The high-level overview of the components in Twister2 is depicted in Figure 4.1. In essence, twister2 is a flexible, high performance distributed data processing engine developed from scratch with HPC technologies and principles at its core. The framework is built with several well-separated layers, each layer abstracting out underlying complex information and from the layer above and providing a clean API interface so that underlying implementation can be switched out without having to change higher levels. The current Twister2 code provides several implementations for some layers, which allows the user to pick and choose layer implementation that best suit their needs.

4.2.1 Data Access Layer

The data access layer is the lowermost layer of the Twister2 framework. It provides users with access to data storage systems such as regular file systems or distributed data layers such as HDFS or NFS. In addition to the supported storage frameworks, users can develop custom data readers by implementing their own data sources. Other than the direct use by end-users, the upper layers of the framework also leverage the data access layer to store and retrieve temporary data that needs to be persisted in files,

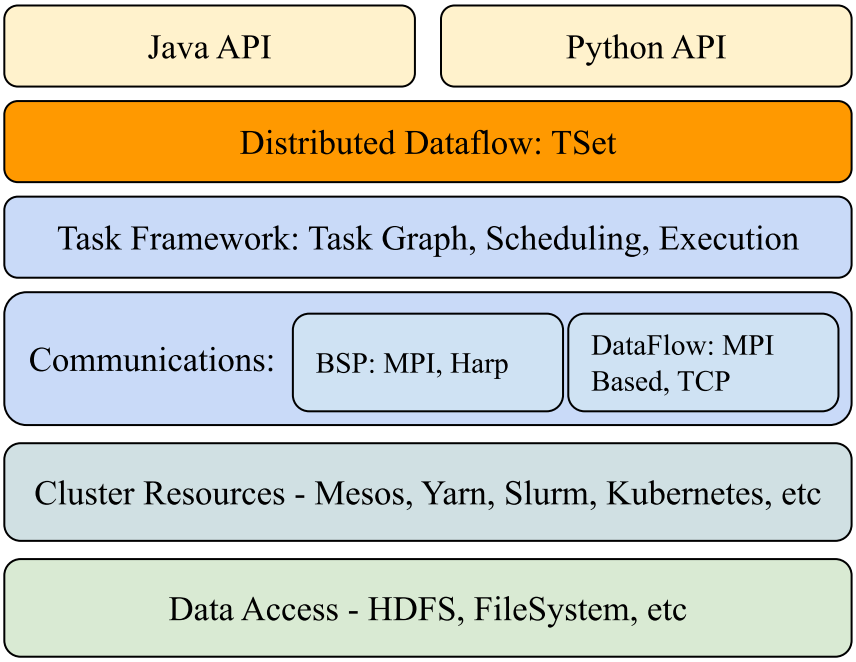


Figure 4.1: Twister2 architecture

4.2.2 Cluster Resource Layer

One of the main requirements when running a distributed frameworks is the ability to obtain and manage a large number of compute resources in the form of compute nodes. In Twister2 the resource management is handled by the cluster resource layer. To make sure Twister2 can be easily used in a multitude of environments, several popular resource management frameworks are supported. For HPC environments, Twister2 supports Slurm resource manager to allocate and manage compute nodes in HPC clusters. In addition, Mesos, Yarn and Kubernetes are supported. The end-user can choose any one of the frameworks which best suites them to deploy a Twister2 cluster. In addition to these, a local resource manager is also supported to allow users to run twister2 locally on a single machine; this is mainly to try out and explore the framework before deploying in a real environment.

4.2.3 Communications Layer

The distributed communication layer is one of the most important layers in the Twister2 framework. It provides highly optimized distributed communication primitives and collective communications such as reduce, gather, keyedreduce, allreduce, etc. Each communication operation is developed and implemented with highly optimized algorithms which mimic the algorithms used in HPC libraries, with the addition of support for dataflow style communications that allow dynamic data sizes. An in-depth study of the communications layer and how it performs compared to other frameworks such as OpenMPI, Spark and Flink is done in [KWG+18], the results show that Twister2 is able to perform on par with HPC frameworks like OpenMPI while vastly outperforming big data frameworks such as Apache Spark. As with all other layers, the communication layer supports several different implementations; for BSP style communications, the layer is implemented with MPI standards and using Harp[ZRQ15] library. For dataflow style communications, the layer is implemented using MPI standard and regular TCP. The communication details such as buffer management and communication links are hidden from the upper layers, and a clean API interface is presented. If needed, end-users can directly develop data processing applications using the communication layer API's. This allows the users to manage all the execution details, such as thread management. While this allows for fine-grained performance optimizations for the application, it is significantly harder to develop the application since most of the execution details need to be handled by the end-user.

4.2.4 Task Layer

The next layer of abstraction in the Twister2 framework is the task layer. Applications developed using the task layer do not need to handle communication details as they are abstracted out. Users specify the tasks and how the tasks are connected through the primitives provided by the API. This layer will manage how the tasks are distributed and executed on the cluster, the application developed will be converted into a task graph, and then the communication layer will be leveraged to perform the necessary distributed communications to achieve the task connections specified by the application. The task layer also manages thread pools to make sure the execution is done as efficiently as possible using the compute resources made available to the application.

4.2.5 Distributed Dataflow : TSet

The highest layer of abstraction in the Twister2 framework is the distributed dataflow abstraction which is also named the TSet API. This layer abstracts out all the lower-level details of both the task layer and the communication layer from the end-user and allows the end-user to develop applications using high-level dataflow abstractions named TSet's. TSet's are similar to Apache Spark RDD'a and Apache Flink DataSet's at the API level; however, optimizations done at this layer and at lower levels of the framework allows TSet's to perform much more efficiently. Implementation details and performance of the Twister TSet layer is discussed in more detail in the following sections and is also discussed in [WKG+19].

TSet User API's

Currently, the TSet API is exposed in both Java and Python programming languages to allow users to choose based on their programming language preference. The Java API is the native implementation since the framework is implemented in Java. The python API acts as a wrapper around the Java API and supports all the functionality provided by the Java API.

4.3 Twister2 Dataflow Model

Dataflow is the most prevalent and most widely adopted model for processing large quantities of data in a distributed environment. The main goal is to hide the implementation details from the end-users while preserving the performance of the framework as much as possible. Since the model follows the flow of data, it makes the process of parallelizing tasks and dynamically building task and process dependencies simpler. In the dataflow model, the application/program written by the end-user is converted into a dataflow graph; this graph can either be created statically or dynamically. The graph consists of task vertices and edges; a task vertex represents a computational unit, in which some specified computational logic is applied to the data that flows through the edges of the dataflow graph. When executing in a distributed manner, each vertex will be represented by a number of execution tasks that will apply the same computational logic on separate sets of input data. The edges which represent communications between tasks will be responsible for properly coordinating and gathering the results from multiple tasks that are being executed in parallel.

Figure 4.4 shows the dataflow graph that represents an implementation of KMeans algorithm done using the TSet API. Each box in the diagram represents a vertex,

and each arrow represents an edge. This diagram gives a clear picture of how data flows between task vertices. Based on the dataflow graph that is generated, the framework can generate a concrete execution plan to determine when each task needs to be executed and how the communication channels need to be set up. This execution plan, which is self a graph, will take into account the data dependencies between vertices in the dataflow graph and parallelisms defined for each vertex in the dataflow graph. The dataflow graph generation can either be done statically or dynamically. If the graph is statically generated, the complete structure of the dataflow graph will be known by compile time. With a dynamically generated dataflow graph, the framework will complete the latter parts of the graph as execution happens.

It is vital that both batch processing and stream processing modes of the dataflow model need to be considered when designing the framework; prioritizing one or the other would result in sub-optimal performance when using one of the two modes. For example, Apache Spark[ZXW+16] was designed as a batch processing engine and therefore handles streaming operations using a mini-batch approach, while Apache Flink[CKE+15] which was designed as a stream processing engine at its core. Twister2 framework has been developed with the goal of adding first-class support for both batch and streaming models. To achieve this, batch and streaming modes need to be supported throughout each layer of the system. A quick example would be how batch and streaming can handle task scheduling differently. In batch mode, a sequence of tasks that need to be executed in order can be executed at the same location to improve performance through data localization. Such a scheduling decision cannot be made in stream mode for a sequence of tasks because each task in the sequence needs to be executing at the same time.

4.3.1 Layered Model

The Twister2 framework has been designed with a layered architecture so that each layer is independent of the layer below it. This is achieved by using a set of well-defined API interfaces between each layer. Having a well defined, fully decoupled layered architecture has two major benefits. First is that each layer can be designed and optimized without any negative effect on the other layers, and each layer could potentially have multiple implementations that cater to different use cases. Secondly, advanced users could develop custom layers to replace one or more of the layers in the framework to suit their needs. At the lowest layer, which is the communication layer, the framework provides the user with the ability to set up parallel processes and exposes a set of communication primitives such as reduce, gather, allreduce, scatter, etc., to the users. At this layer, the system has semantics that are very similar to MPI specifications. And each communication operator has been developed using optimized algorithms derived HPC domain knowledge. The layer above the communication layer is the task layer. At the task layer, the main abstraction users interact with are tasks; the details of the communications layer are concealed from the user. The user models applications as a set of tasks and define dataflow connections between the tasks that were given. Internally the framework will replace the connections defined by the end-user with the most suitable optimized collective communication operation when the dataflow graph is generated.

Finally, the TSet layer, which can be viewed as the true dataflow model abstraction layer, is built on top of the task layer. TSet layer has similar characteristics to other big data framework API's such as Apache Spark RDD's and Apache Flink DataSet. Even though the high-level programming language interface exposed to the user is similar to other big data frameworks, underneath the optimized communication layer and the iteration model adopted by Twister2 allows it to outperform

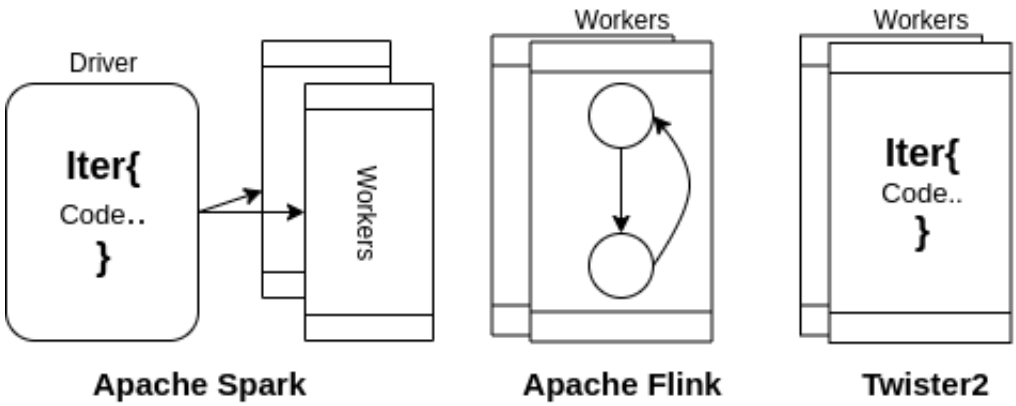


Figure 4.2: Different iteration models in Spark, Flink and Twister2

other major big data processing frameworks. In this paper, we will discuss the details of the TSet layer and its design, some aspects of the task layer will be touched upon to help with discussion, but in general, the implementations of the task layer and communication layer are not within the scope of this paper.

4.3.2 Iterations

Most, if not all, complex parallel/distributed applications have some section of the code that is executed in an iterative fashion. This is even more prevalent in machine learning and deep learning programs, where the bulk of the computation time is spent within iterative code segments. In most parallel programs other than pleasingly parallel programs, the end of an iteration is accompanied by some sort of synchronization operation. This is typically used to share the results calculated during the iteration that was completed. For example, in a deep learning applications, after each iteration the calculated local gradients need to be shared and averaged before the next iteration in order to calculate the weight parameters for the model. Therefore an efficient iteration model is vital for the performance of any distributed data processing framework. The approach taken by Twister2 TSet's to

handle iterations is an important point that sets TSet's apart from other big data frameworks. Before looking at the iteration model in Twister2 it is important to understand how existing frameworks have addressed iterations and how they have gradually moved towards BSP style iteration model adopted by Twister2 TSet's.

Apache Hadoop[Whi12] was one of the first open-source big data frameworks developed using the map-reduce model. Before being replaced with frameworks such as Apache Spark, Apache Hadoop was used heavily for distributed data processing applications. In Apache Hadoop, at the end of each iteration, the intermediate results were written to disk and read back from disk for the next iteration. This meant there were two file I/O operations for each iteration. This was very inefficient because I/O operations are extremely time-consuming operations even with a state of the art storage device. As a solution to this issue, iterative MapReduce frameworks such as Apache Spark[ZCD+12] and Twister[ELZ+10] introduced in-memory operations, which removed the need to write to ad read from disk after each iteration. Instead, the intermediate state would be kept in memory and used in the next iteration. This can also be viewed from the angle of how the semantics of the iteration is handled. How Apache Spark handled, iterations can be seen as moving the control of the iterations from the client (as in Apache Hadoop) to the driver/master. Performing iterations at the driver still retains several unwanted overheads. After each iteration, the results must be collected at the driver process and later broadcast to each worker for the next iteration. For example, if the program was training a neural network using data parallelism, at the end of each iteration the weight parameters would need to be collected and averaged at the driver node and broadcast back to the workers for the next iteration, in a BSP model this could be done more efficiently using a collective operation such as allreduce. In Apache Flink iterations are embedded into the generated dataflow graph itself. This results in the iteration control

being moved into the worker level rather than being controlled at the driver level as with Apache Spark. However, Apache Flink does not currently support nested iterations because of constrained introduced by the iteration model of embedding the iterations in the dataflow graph.

Twister2 adopts its iteration model from HPC BSP model, which is arguably the most used iteration model in HPC. With this model, Twister2 TSet's moves the iteration control to the worker level, the primary contrast with how Apache Flink, which also moves the iteration controls to the worker level, is that, unlike Flink, the iterations are not baked into the dataflow graph. This facilitates a cleaner iteration model with no constraints on nested iterations, so TSet's are able to support nested iterations without any additional modifications needed. Figure4.2 shows how Apache Spark, Apache Flink and Twister2 handles iterations. Moving the iteration logic into the workers removes the need to gather information at a central location, as seen in Apache Spark after each iteration, allowing the use of more optimized communication primitives to be used for synchronization at the end of each iteration. For example, end-users can use a collective communication operator like AllReduce to perform the synchronization.

Supporting the BSP style iteration model in the framework is straightforward in the communications layer of Twister2 since the framework closely resembles the BSP model at this layer. The challenge of implementing this iteration model comes when we move to the higher layers, which are the task layer and TSet layer. In these layers, the program written by the end-user needs to be converted into a dataflow graph before it can be executed, the iteration logic also needs to be encapsulated within this logic. Since Twister2 does not embed the iterations in the dataflow graph itself, the framework has to build the sub dataflow graph needed for the logic inside iterations for each iteration. This can be more clearly understood by looking

at an example. Algorithm 1 shows the pseudo-code for the KMeans algorithm using the TSet API. In this, each time line 7 is executed, the dataflow graph for the TSet "reducedMap" needs to be generated; this adds unwanted computation overheads to the execution of the program. To address this issue Twister2 TSet's introduce the iterative execution mode in which the framework caches the dataflow graph built during the first iteration and reuses it for subsequent iterations. The code change needed to change to the iterative mode is listed in Algorithm 2. Passing in "true" to the "cache" method informs the framework that it should consider that section of the logic as an iterative section. With this information, the framework will save the required dataflow graph segments and reuse them in subsequent iterations. The single additional change that is required is to call the "finishIter" function, which cleans up the data structures that were created in the environment.

Algorithm 1: KMeans using Twister2 TSet API

```
1 CachedTSet points = tc.createSource(...).cache();
2 CachedTSet centers = tc.createSource(...).cache();
3 ComputeTSet kMapTSet = points.direct().map();
4 ComputeTSet reducedMap = kMapTSet.allReduce(...).map(...);
5 for i=0 to maxIteration do
6 |   kMapTSet.addInput(centers);
7 |   centers = reducedMap.cache();
8 end
```

Algorithm 2: KMeans using Twister2 TSet API in Iterative mode

```
1 CachedTSet points = tc.createSource(...).cache();
2 CachedTSet centers = tc.createSource(...).cache();
3 ComputeTSet kMapTSet = points.direct().map();
4 ComputeTSet reducedMap = kMapTSet.allReduce(...).map(...);
5 for i=0 to maxIteration do
6 |   kMapTSet.addInput(centers);
7 |   centers = reducedMap.cache(true);
8 end
9 reduced.finishIter();
```

Migrating the iteration control to the worker level can be seen as the logical next step in improving the performance of iterations in distributed big data processing frameworks. From managing iterations in the client as in Apache Hadoop to managing iterations in the drive/master node as done in Apache Spark to finally moving the iteration control completely into the worker nodes them-self. This mode of handling iterations in distributed and parallel programs has been tested and proven in the HPC community over the past couple of decades, and the results obtained for TSet's that are presented in this paper will further solidify this claim.

4.4 TSet's

TSet API is the high-level programming interface exposed by Twister2; it is similar in function and capabilities to RDD API in Apache Spark and DataSet API in Apache Flink. It has both a Java API and a Python API; the Python API is built on top of the Java API. The TSet API defines a set of transformations and actions which the users can use to develop distributed data processing applications. The

details of how the program is parallelized and executed in a distributed manner is hidden from the user. TSet's support both batch processing and stream processing and expose both with similar interfaces. The discussion and explanations of the concepts and details on how TSet's are developed will apply to both modes; if some part does not apply to one of the modes, it will be stated explicitly.

When developing a data processing application, the user will define the program as a graph where vertices represent computations and edges represent dataflows that happen between computation units. Underneath the framework will convert this into a fully-fledged dataflow graph with parallelism information embedded and take care of data distribution. One of the main goals of the TSet API is to provide an easy to use and simple to understand interface for the end-users. To preserve the simplicity of the API, the TSet API does add some restrictions and is not as expressive as lower-level API's such as the Task API or Communication API, but it does preserve the versatility as much as possible.

The TSet API is comprised of two major entities which are namely "TSet" and "TLink". TSet's represent the vertices of the graph, and TLink's are the edges that connect one TSet to another. Table4.1 lists all the TSet's and TLink's which are currently supported in the Twister2 framework.

4.4.1 TSet

A single TSet represents a node in the dataflow graph. Each TSet is accompanied by some form of computation logic that it is responsible for executing. Each TSet typically has an input and an output associated with it; this is true for all TSet's other than the SourceTSet and the SinkTSet. SourceTSet only has an output link, and the SinkTSet only has an input link; this is because they represent the start and

TBase	sets	TSet	batch	ComputeTSet	
				SourceTSet	
				SinkTSet	
				CachedTSet	
			streaming	SComputeTSet	
				SSourceTSet	
				SSinkTSet	
		TupleTSet	batch	KeyedTSet	
			streaming	SKeyedTSet	
	links	TLink	batch	BSingleTLink	AllReduceTLink
					ReduceTLink
				BIteratorTLink	DirectTLink
					ReplicateTLink
					PartitionTLink
					KeyedGather
					KeyedReduce
					KeyedPartition
					JoinTLink
				BBaseGatherTLink	AllGatherTLink
					GatherTLink
			streaming	SSingleTLink	SAllReduceTLink
					SReduceTLink
					SDirectTLink
					SReplicateTLink
					SPartitionTLink
					SKeyedPartition
				SBaseGatherTLink	SAllGatherTLink
					SGatherTLink
				SIteratorTLink	-

Table 4.1: Twister2 TSet Organization

end of the dataflow graph. Each program written using the TSet API starts off with a SourceTSet; from this, the application graph can be gradually built by connecting TSet's using TLink's. The API restricts the user so that only valid transformation sequences can be applied when creating the dataflow graph. For example, applying a key-based collective operation such as KeyedReduce on a TSet such as a Map that does not output a key-value pair would result in an invalid graph. Therefore the framework enforces several restrictions through the API itself to make sure the user does not end up programming an invalid dataflow graph.

In order to provide first-class support for both batch and stream processing, all TSet's have corresponding stream and batch versions defined. On top of these two categories, TSet's can be divided into two broad areas based on the type of data that is being handled.

- TSet - Used for individually typed data
- TupleTSet - Used for keyed data arranged in Tuples

Table 4.1 shows the TSet's that are available in the API and how they are categorised into sub-groups. These well-defined categories of TSet's allow the framework to apply the most suitable lower-level semantics to obtain the best possible performance when it is converted into task level dataflow graphs and execution graphs.

4.4.2 TLink

A TLink represents an edge in the dataflow graph. TLinks are therefore responsible for managing the communications that need to happen between TSet's. Because TSet's are executed in a distributed manner, the TLinks perform some form of distributed communication based on its definition. Table4.1 lists all the TLink's

Name	Description	Is action
Compute	Performs basic compute transformation, more general version of map and flatmap	false
Map	Performs map transformation	false
Flatmap	Performs flatmap transformation	false
MapToTuple	Maps Values to a Key-Value pairs	false
Join	Perform join operations such as inner-joins, outer-joins on the TSet's	false
Union	Generates the union of 2 or more TSet's	false
Cache	Evaluates the current graph and caches the values in-memory	true
ForEach	Evaluates the current graph and performs an computation per item	true
Sink	Evaluates the current graph and stores the results as speficied through the sink function	true

Table 4.2: Twister2 TSet Transformations and Actions

that are available in the Twister2 TSet API. As with TSet's TLink's have separate implementations to support batch and streaming versions to make sure Twister2 performs optimally for either version. Underneath each TLink is implemented using highly optimized communication operators introduced in Twister:Net[KWG+18]. For example, the reduce operation is performed using an inverted binary tree to keep the number of calls that are needed to complete the communication to a minimum. Based on the content of the messages communicated through the TLink's, they can be categorized into three major categories.

- SingleTLink - For communications that produces a single output
- IteratorTLink - For communications that produces an iterator
- GatherTLink - Specialized TLink for Gather operations (gather, allgather)

These categories are present for both batch and streaming modes. Table 4.3 shows all the TLink's in the API in greater detail. The table shows how each communication primitive is represented as TLinks and the message content for each.

Mode	Communication	Message Content	Parallelism	TLink Representation
Batch	Reduce	T	m to 1	SingleTLink[T]
	Allreduce	T	m to 1	SingleTLink[T]
	Direct	Iterator[T]	m to m	IteratorTLink[T]
	Broadcast	Iterator[T]	1 to m	IteratorTLink[T]
	Gather	Iterator [Tuple[Integer, T]]	m to 1	GatherTLink[T]
	Allgather	Iterator[Tuple[Integer, T]]	m to 1	GatherTLink[T]
	KeyedGather	Iterator[Tuple[K, Iter[T]]]	m to n	IteratorTLink[Tuple[K, Iter[T]]]
	KeyedReduce	Iterator[Tuple[K, T]]	m to n	IteratorTLink[Tuple[K, T]]
	Partition	Iterator[T]	m to n	IteratorTLink[T]
	KeyedPartition	Iterator[Tuple[K, T]]	m to n	IteratorTLink[Tuple[K, T]]
	Join	Iterator[JT[K, U, V]]	m to n	IteratorTLink[JT[K, U, V]]
Streaming	Reduce	T	m to 1	SingleTLink[T]
	Allreduce	T	m to 1	SingleTLink[T]
	Direct	T	m to m	SingleTLink[T]
	Broadcast	T	1 to m	SingleTLink[T]
	Gather	Iterator[Tuple[Integer, T]]	m to 1	GatherTLink[T]
	Allgather	Iterator[Tuple[Integer, T]]	m to 1	GatherTLink[T]
	Partition	T	m to n	SingleTLink[T]
	KeyedPartition	Tuple[K, T]	m to n	SingleTLink[Tuple[K, T]]

Table 4.3: Twister2 Communication semantics

It also shows the parallelism mapping for each TLink, that is, whether it is an "*m to 1*", "*1 to m*", or "*n to m*" operation.

4.4.3 Transformations, Actions and Lambdas

One of the main goals of the TSet API is to make the programming interface which is simple yet powerful, allowing users to develop complex distributed data processing applications with ease. To this end, the API exposes a set of transformations and actions, which are listed in Table4.2. Each transformation/action is applied on a TLink which represents a stream or batch of data. Applying a transformation/action on the input data results in the creation of a new TSet. Figure 4.3 shows a simple example to visualize how transformations and communications are composed. In the figure, the source which generates some form of input data is fed into a "map" transformation and "foreach" action through a direct communication link. The "foreach" action simply iterates through the input data, which the user can use to simply print the data into the console. Data flowing into the map transformation will be modified using the user-defined computation logic for that map operation. The transformed data can then be sent through a communication operation, which is specified as "reduce" in this example. Users can expand the program and perform any number of transformations by stacking them one after the other to generate the desired final result.

As listed in Table 4.2 Twister2 API provides a range of transformations to be used. Most transformation takes in a function that will be used as the computation logic for that transformation; others, more specialized transformations/actions have the logic required to perform the computation built into them. The API also supports lambda functions when the Java API is used. Lambda functions are much easier to develop and use. The only slight drawback is using lambda functions is that the TSet Context, which includes runtime metadata such as parallelism, rank, etc., is not accessible from the lambda function due to limitations in the Java programming language.

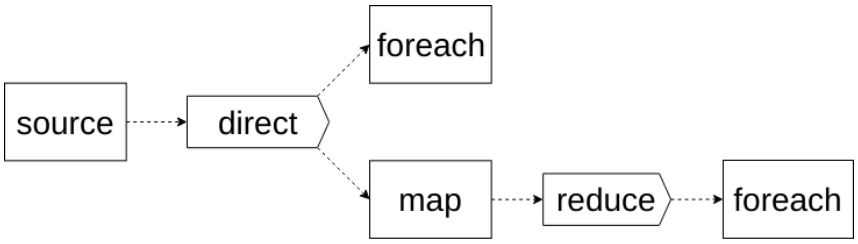


Figure 4.3: Example TSet execution

4.4.4 Lazy Evaluation

During runtime, all applications written using the TSet API are evaluated using a lazy evaluation strategy. This essentially means that until one of the "actions" is called, the dataflow graph will not be evaluated by the framework. Currently, there are three actions specified in the TSet API, which are "sink", "foreach" and "cache", as listed in Table 4.2. Only when one of these actions are attached to the dataflow graph will the graph be executed by the framework. For example, when looking at Figure 4.3 once the "foreach" action is called, the framework willtraverse *b*ack th*r*ough the *d*ataflow graph to determine the sequence of transformations and communications that need to be performed to complete that action. Only then will the framework execute the particular sub-graph and output the results. For Figure 4.3 the two sub-graphs that will be executed are as follows.

- source -> direct -> foreach
- source -> direct -> map -> reduce -> foreach

Lazy execution model allows the framework to perform optimizations on the graph execution efficiently; for example, if there are multiple consecutive map transformations connected through direct communications links in the sub-graph that needs to be evaluated, the framework can simply pipeline the map transformations into a single map operation to improve performance. If required, the user is free to

add dummy actions after each transformation to make sure each transformation is evaluated separately, but this would result in sub-par performance.

Alternatively, the users can force the entire graph to be executed using the "eval" method provided in the TSetEnvironment. In this mode, the framework will traverse the complete dataflow graph starting from the initial sourceTSet using breadth-first traversal and then execute the entire dataflow graph. When the entire graph is executed, the framework will add its own action to the leaves of the graph. Therefore the "foreach" actions in the example given in Figure 4.3 would need to be replaced with "map" transformations so that the framework can append actions to the end of the graph. This mode of execution is the default execution mode for streaming applications. For streaming applications, the complete dataflow graph needs to be executed concurrently, which necessitates the use of this mode. Executing sub-graphs of the dataflow graph as discussed above is undefined for streaming mode and only applies for batch mode.

4.4.5 Caching

Caching intermediate results during execution is an important strategy that helps improve performance, especially in iterative algorithms. The TSet API allows users to cache intermediate results of the dataflow graph by calling the "cache" action. When this is called, the framework will evaluate the sub-graph up to that call and save the results in-memory using a specialized TSet named the "CachedTSet". Using the cached result in an iterative computation will remove the need to evaluate the complete sub-graph for each iteration, improving the performance of the program. "CachedTSet"'s can be consumed in two distinct manners.

1. Treated as a source and build a new graph with the CachedTSet as the starting point
2. Set as an input to a separate TSet so the cached values can be accessed during computation steps

This gives the user the freedom to use cached results as needed. This can be more clearly understood in section 4.4.6 which walks through how the KMeans algorithm can be efficiently implemented in the TSet API.

4.4.6 KMeans Walk Through

Kmeans is a well-known machine learning algorithm that is used to cluster data points into a number of clusters. The parallel version of KMeans is implemented by dividing the data points in the dataset into several partitions and assigning the data partitions to distributed tasks. Each task calculates the distances to the centers for the data partition assigned to it and assigns each data point to the closest center point. At the end of each iteration, values are summed across all the tasks to generate the new center points before the next iteration. Listing 1 shows the pseudo-code for implementing the KMeans algorithm using the TSet API.

In line 1 and 2, two SourceTSet's are created for the data points and the initial centers. Both of them are cached since they will be in an iterative algorithm to make sure data loading is only done once. Twister2 will partition the data according to the user logic and load only the assigned partition in each parallel task. All the center points will be loaded at each task since they cannot be partitioned. Line 3 calls a map transformation on the cached data points TSet; this map function contains the logic to calculate the euclidean distances from each point to the centers and to make the correct center assignment for each data point. At line 4, an "AllReduce"

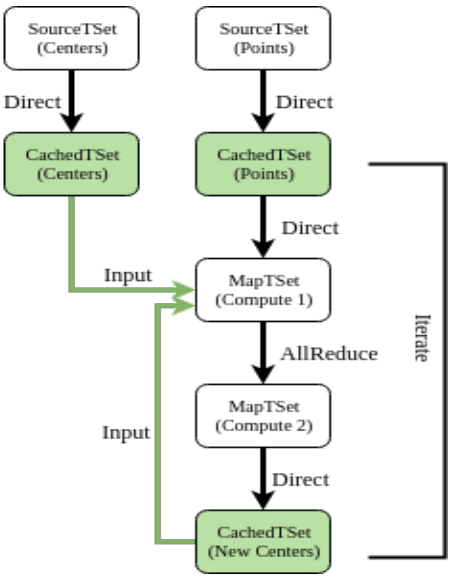


Figure 4.4: K-Means TSet API dataflow graph

collective communication operation is applied on the map TSet to collect the new center values from all the parallel map tasks. These values are then sent through another map transformation which does an average to calculate the center points for the next iteration. Since this was done using an "AllReduce" operation, each parallel worker will now have the same values as the new centers. Lines 5-8 show the iteration logic used in the KMeans algorithm. In line 6, the code adds the cached center values into the first map transformation which does all the calculations since the center values are needed for that calculation. In line 7 we update the centers with the new values so that the next iteration is performed with the new center values. The dataflow graph for the Kmeans algorithm is visualized in Figure 4.4. This implementation also employs both methods of using a cached TSet mentioned in section 4.4.5. In line 3, the cached data points TSet is used as a source, and in line 6 the cached centers TSet has been used as input for a map function. Another important note to make is that the transformations in line 3 and 4 are only evaluated when line 7 calls the "cache" action.

4.4.7 TSet Implementation and Challenges

The TSet implementation resides as a sub-component within the larger Twister2 framework. The complexity of the Twister2 implementation stack is managed through the clearly defined layers that were discussed. Because the communication layer handles all the parallel communication operations and because the task layer handles all the task scheduling and execution operations, the TSet layer does not need to manage the complexities of those operations. Instead, the TSet layer directly relies upon the Task and Communication layer to perform those operations. The complexity for TSet's mainly come from how the dataflow graph is managed and on how it abstracts out as much of the finer details as possible while providing a flexible interface to the end-user. The complete Twister2 code base is made up of around 170K lines of Java, Python and JavaScript code. The code added to implement TSet's on top of the Twister2 framework makes up around 15K lines of code of this codebase.

4.5 Apache Beam Twister2 Runner

Apache Beam[apa] is a popular open-source framework that was introduced and promoted by Google. The main goal of this project is to provide a unified model for distributed data processing in both batch and stream mode. It has attracted a large developer community and has seen wide adaptation in the industry and academia. The major selling point of Apache Beam is that once a program is developed using its unified API, it can be executed on a multitude of data processing frameworks without having to do any additional changes. To achieve this, Apache Beam supports several distributed data processing back-ends, which are known as Beam runners. Currently over 10 such back-ends including Apache Spark[ZCD+12],

Apache Flink[CKE+15], Google Cloud Dataflow[ABC+15], etc. Supporting many back-ends and being able to seamlessly switch between back-ends to run a given program makes Apache Beam an amicable choice for distributed data processing for many end-users. Figure 4.5 shows the high-level architecture of Apache Beam and how it provides a unified model to the end-user through various programming API's and various execution platforms.

In order to run programs written using the unified API provided by Apache Beam in other frameworks, each runner includes transformation logic that converts the program into a program that is understood by the runner's respective framework. For example, if the program is to be executed on an Apache Spark cluster, the Apache Beam Spark runner will convert the program into a program written using RDD's which is the programming interface of Apache Spark. This is done programmatically, so the end-user does not need to have any knowledge on how this is done. For any distributed data processing framework being supported as a distributed back-end in Apache Beam open ups the possibility to access the vast user base that it boasts easily. With the flexibility and versatility of the Twister2 TSet API we were able to contribute and be accepted as a fully-fledged distributed data processing back-end for Apache Beam in a relatively short time. Twister2 is now listed on the Apache Beam homepage[bea] among popular frameworks such as Apache Spark.

In order to develop a full-fledged Apache Beam runner, there is a set of transformations that need to be implemented so that Beam programs can be converted into Twister2 TSet programs. The five major operations that need to be supported are listed below. Other than these five major transformation primitives, many smaller requirements need to be met in order to become a fully-fledged Apache Beam runner. A PCollection is an unordered bag of elements in Apache Beam.

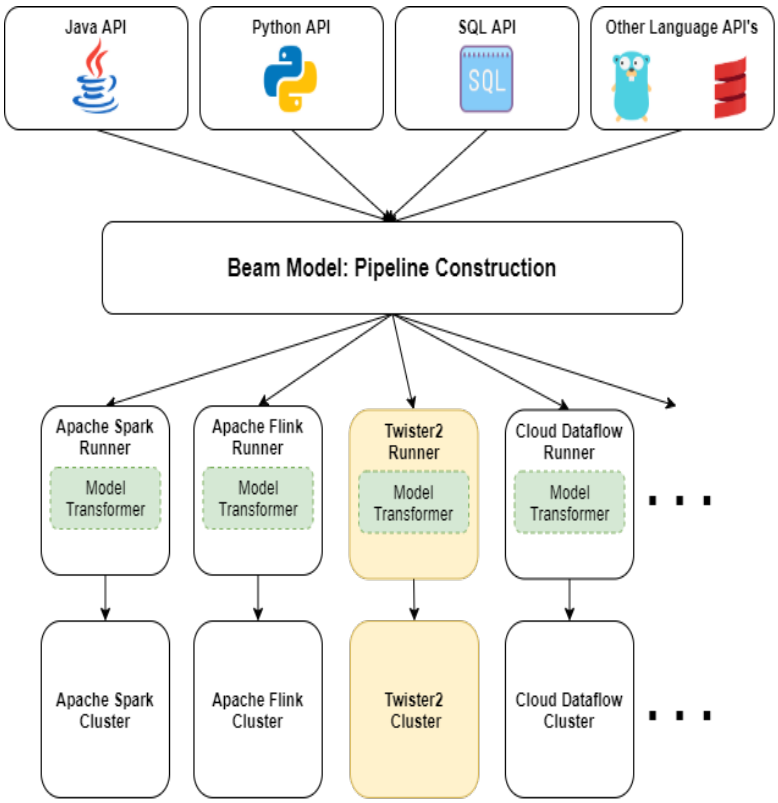


Figure 4.5: Apache Beam Architecture

1. Read - parallel connectors to external systems
2. ParDo - per element processing
3. GroupByKey - aggregating elements per key and window
4. Flatten - union of PCollections
5. Window - set the windowing strategy for a PCollection

4.5.1 Read

The read primitive represents data sources in the Apache Beam. It is used to collect raw data and connect to external data sources. Beam supports two versions of read primitives, one for bounded data sources and one for unbounded data sources (streams). Each need to be implemented by the runner to support batch and stream data. Within the runner, the read transformer is responsible for converting the beam data source into a data source supported by that particular runner. In the Twister2 runner this is achieved by transforming the beam data sources into a SourceTSet in the Twister2 TSet API. Since beam data sources are distributed objects, the transformed read primitives also need to adhere to the distributed nature of the data source and support parallelism through data source splits.

4.5.2 ParDO

The ParDO primitive can be thought of as the core data transformation primitive in Apache Beam; it defines the element-wise transformations that need to be performed on data points. ParDO is the most complex primitives of all the primitives that a runner needs to support since it acts as a catch-all operation that represents multiple standard operations. Not only does it represent operations such as map and

flatmap, it also supports multiple outputs, stateful processing, side inputs, teardown operations, etc. Therefore in the transformer that converts ParDo primitives into equivalent counterpart in Twister2 it also needs to support the whole range of functionality. ParDO primitive is implemented using a combination of ComputeTSet and CachedTSet in the Twister2 runner, while ComputeTSet is versatile enough to support most of the use cases of the ParDo operation, supporting side inputs require the use of CachedTSet's.

Within the ParDo primitive, users are allowed to define the custom processing logic using the DoFn abstraction; within the transformer logic, DoFn are converted into Twister2 functions using the function abstractions provided in the Twister2 TSet API. DoFn's support needs to cover the complete function lifecycle defined by the Beam. A bundle represents a small batch of data points

1. Setup - Called once per function instance for initialization work
2. StartBundle - Called once per bundle
3. ProcessElement - called for each element
4. FinishBundle - finalizing details per bundle
5. TearDown - Called once per function instance for cleanup

4.5.3 GroupByKey

The groupByKey primitive allows key-value pairs to be collected into groups based on the key value. This is an essential operation that is required in many machine learning and data analytic pipelines. Keys in the groupByKey primitive are seen as a sequence of bytes; therefore, the grouping logic needs to implemented based on the byte sequence even if knowledge of the key types are available. In the Twister2

runner groupByKey primitive is implemented using a combination of KeyedTSet's and ComputeSet's.

4.5.4 Flatten

The Flatten primitive is one of the more simple primitives in Beam; as the name suggests, the function of the primitive is to take a batch of PCollections and flatten them to create a single PCollection that represents all whole batch; however, windows defined within the data need to be preserved during the transformation. Within the Twister2 runner, this operation is implemented using union transformation listed in Table 4.2.

4.5.5 Window

The Window primitive applies a user-defined function (WindowFn) that applies to each element in a PCollection so elements can be assigned to specific windows in the outputs of that PCollection. The functionality is somewhat similar to a groupby operation that is based on timestamps. The Window primitive is implemented using ComputeTSet, and the WindowFn is defined using a ComputeCollector function in the TSet API.

Once all the primitives are implemented, the runner needs to pass a rigorous test suite to make sure that all the edge cases are supported properly. The acceptance of the Twister2 runner for Apache Beam opens up a wide user base for Twister2 since it can directly be used with Apache Beam programs. It also validates the versatility and the extensibility of the Twister2 TSet API since all the Beam primitives were implementable with no custom changes needed on the Twister2 TSet API side.

4.5.6 Implementation and Challenges

Apache Beam is a well established open source project that has a large developer and user community. While it is an Apache project, the core developers of the project are from Google since it was donated to the Apache foundation from the cloud dataflow team in Google. Apache Beam already has a considerable amount of users who use the project in production level environments; therefore, any contribution done to the project is code reviewed and validated by the current committers. The first challenge was to implement all the primitives that are described above; several segments of the implementation code is added in the Appendix. Most of the primitives were implemented using the "Compute" transformation that is provided by the TSet API. One thing that made the implementation more manageable was the fact that Beam converted all data objects into byte arrays before they are communicated since Twister2 TSet's support byte data loads handling types and data communication was made more straightforward. With regard to the implementation, one challenge was the use of a concept named Coders in Beam; Coders were responsible for serializing and de-serializing the content that was passed through the communication channels, in Twister2 such classes are serialized and instantiated in parallel workers when needed. However, the coder classes were not java serializable therefore, we had to create a workaround where we save the coder as a byte array and create the objects from this byte array when needed using a set of util methods. The most challenging part of the implementation was the test phase. In order to be accepted into the Beam project as a fully functional runner, the Twister2 runner needed to pass a rigorous set of unit and integration tests. In order to pass the integration tests, we had to upgrade the Twister2 local implementation to add more capabilities to it. This was because the Beam integration tests were executed in a single machine, and thus the local executor required to have all the capabilities of the

distributed executor, which we did not have at that time. Once the implementation passed all the required tests and the code standard was validated by the community, the runner was accepted into the Beam project as a fully-fledged runner. One aspect which made the process manageable was the continued support from several core members of the Beam project. They helped navigate the large Beam codebase so that we knew where to look for when implementing certain details.

The complete codebase needed to implement the Twister2 runner was close to 6000 lines of code. The Twister2 runner also had to pass around 300 integration tests before being accepted. While the research aspect of developing the runner was small, this validated that the Twister2 TSet API was powerful and extensible enough to cover all the requirements of large scale matured data processing project such as Apache Beam.

4.6 Evaluation

One of the main goals of developing the Twister2 TSet layer was to provide a high-level dataflow abstraction to the user while adding as little overhead to the performance of the program as possible. In Twister:Net [KWEF18] it was shown that the underlying communication layer performed exceptionally well compared to other big data processing frameworks and was on par with OpenMPI. However, the many additional layers that have been added can add significant overheads to the execution time. Therefore it is important to evaluate the performance of the framework at the TSet API level. This will also help solidify the effectiveness of the iteration model adopted in the Twister2 TSet framework.

To thoroughly evaluate the TSet API several applications are developed and executed in a distributed environment. To compare and contrast with other state

of the art frameworks, identical applications (algorithmically) were developed using OpenMPI(v4.0.1) and Apache Spark (v2.4). The applications implemented for the performance comparison are a distributed KMeans algorithm and DAMDS algorithm. DAMDS is a complex algorithm when compared to the more simple KMeans algorithm, and it shows how well Twister2 TSet API performs for complex machine learning algorithms. All the execution timing values presented are calculated by taking the average of 3 runs. Two compute clusters were used to perform the evaluation. The first cluster had 16 nodes of Intel Platinum processors with 48 cores in each node, 56Gbps InfiniBand and 10Gbps network connections. The second cluster had Intel Haswell processors with 24 cores in each node with 128GB memory, 56Gbps InfiniBand and 1Gpbs Ethernet connections. 16 nodes of this cluster were used for the experiments. Both clusters have network mounted file system (NFS) for storage requirements. For the experiment results, time taken to initially load data into the program is not counted. This is to make sure that the performance numbers are not affected by the IO performance differences in different frameworks. Intermediate data is always kept in-memory using a caching function provided by each function to remove unwanted file IO operations.

4.6.1 K-Means

KMeans is a well known iterative clustering machine learning algorithm. Given a set of data points, the algorithm finds "K" cluster centers and assigns the data points to these to perform the clustering operation. KMeans starts with some initialized values for the "K" cluster centers and gradually adjusts the centers to find the optimal center positions based on the initial center positions. Since this is a highly iterative but simple algorithm, it is ideal for testing the performance of the framework, and

the iteration model introduced. For evaluation, we use a distributed version of the KMeans algorithm. In each implementation of the algorithm, caching constructs provided by each framework was used to cache the input data partitions and the initial center points.

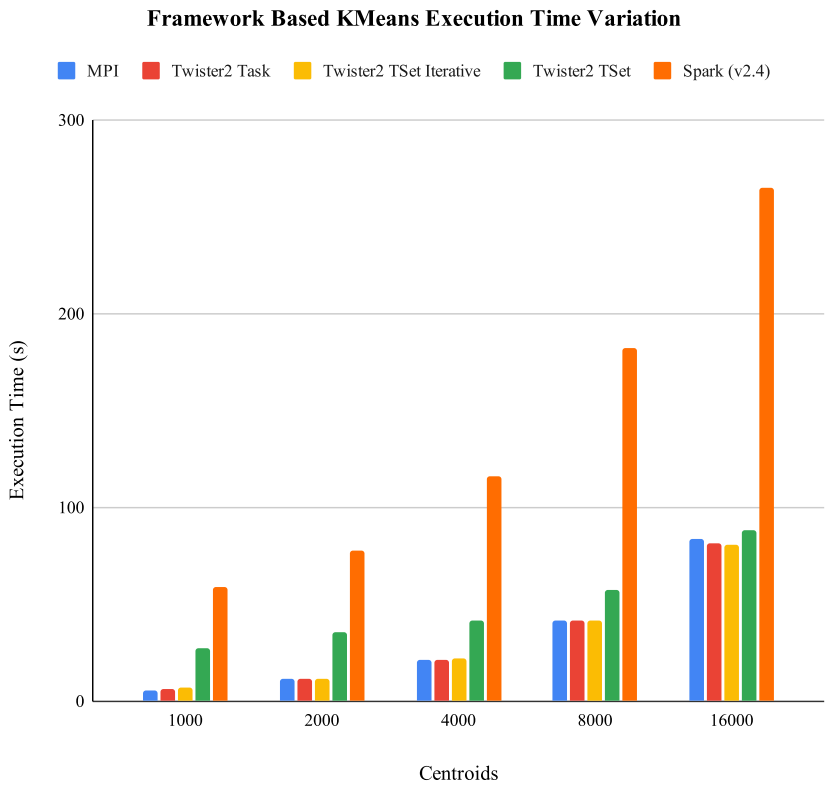


Figure 4.6: K-Means job execution time on 16 nodes with varying centers, 2 million data points with 128-way parallelism for 100 iterations.

The distributed KMeans algorithm was executed on a 16 node cluster with a parallelism of 128 for OpenMPI, Apache Spark, Twister2 Task and Twister2 TSet implementations; the results are shown in Figure 4.6. A Twister2 Task layer implementation was tested to observe the overhead added solely by the TSet layer. For TSet API, both methods listed in Listing 1 and Listing 2 are tested; the latter is labelled as Twister2 TSet Iterative. Each run was done with 2 million data points

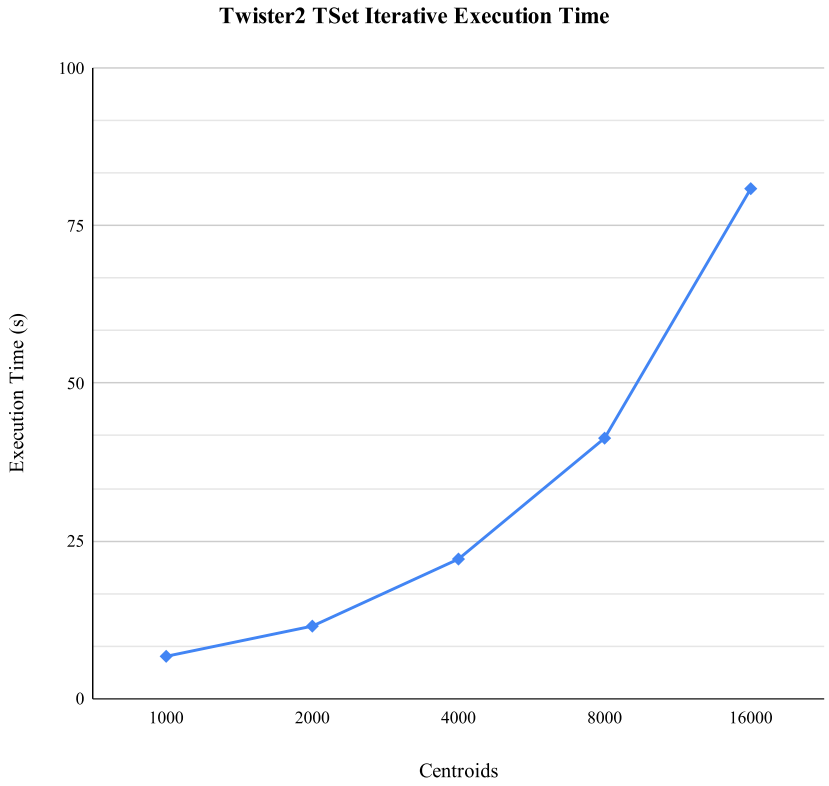


Figure 4.7: TSet Iterative K-Means job execution time on 16 nodes with varying centers, 2 million data points with 128-way parallelism for 100 iterations.

with two features and a varying number of centers starting from 1000 centers and increasing up to 16000 centers for 100 iterations. As seen in Figure 4.6 Twisters2 Task implementation and the Twister2 TSet Iterative implementation performs on the same level as OpenMPI. The base TSet implementation that does not use the optimizations done to iterations as mentioned in 4.3.2 performs slightly slower, which can be attributed to the overhead added because of the repeated creation of the dataflow graph. However, even in that mode TSet implementation performs significantly better than Apache Spark, which is 2-3x slower than the TSet implementation in each case. These results confirm that the TSet API provides a high level easy to use programming interface to users while adding very little overhead to the appli-

cation, even when compared to highly optimized HPC implementations of the same algorithm. It is important to note while TSet based implementation performs on par with OpenMPI, implementing the distributed KMeans algorithm in OpenMPI is significantly more complex and requires expert knowledge in parallel programming, while it is much easier to implement the same in Twister2 TSet API. Figure 4.7 shows how the Twister2 TSet implementation performs with increased centroids, as the number of centers double the total execution time of the algorithm also roughly doubles as expected because both computation load and communication load double.

4.6.2 Deterministic Annealing Multi-Dimension Scaling(DA-MDS)

Deterministic annealing multi-dimensional scaling (DAMDS) [RF13] is distributed multi-dimensional scaling (MDS) algorithm. MDS is a well-established machine learning algorithm that is used to project high dimensional data into lower dimensions. MDS is useful when high dimensional data needs to be visualized in three dimensions or two dimensions to analyze the data. DAMDS is a relatively complex algorithm and consumes a large amount of memory since it performs many matrix operations. DAMDS, as with most MDS implementations, has a computation and memory complexity of $O(N^2)$. Because of this, efficient parallel implementations are needed to execute DAMDS on even moderately large datasets. This complex algorithm will help evaluate the performance of the TSet API for complex machine learning algorithm with high computation and memory requirements. In order to compare against other frameworks, the DAMDS algorithm was implemented using Twister2 TSet API, Apache Spark and OpenMPI. In a previ-

ous peer-reviewed publication, it was shown that Apache Spark and Apache Flink based DAMDS algorithms perform up to an order of magnitude slower than Open-MPI based implementation[KWEF18]. Figure 4.8 shows the execution times for DAMDS implementation. The tests were conducted with varying input matrix sizes ranging from 5000x5000 to 25000x25000 with a parallelism of 128 for all the runs. Results show that the Twister2 implementation performs on-par with Open-MPI while outperforming the Apache Spark implementation by a significant margin. Figure 4.9 show how the Twister2 implementation compares with the OpenMPI implementation. While the Twister2 implementation runtime is slightly large, it is not a significant difference.

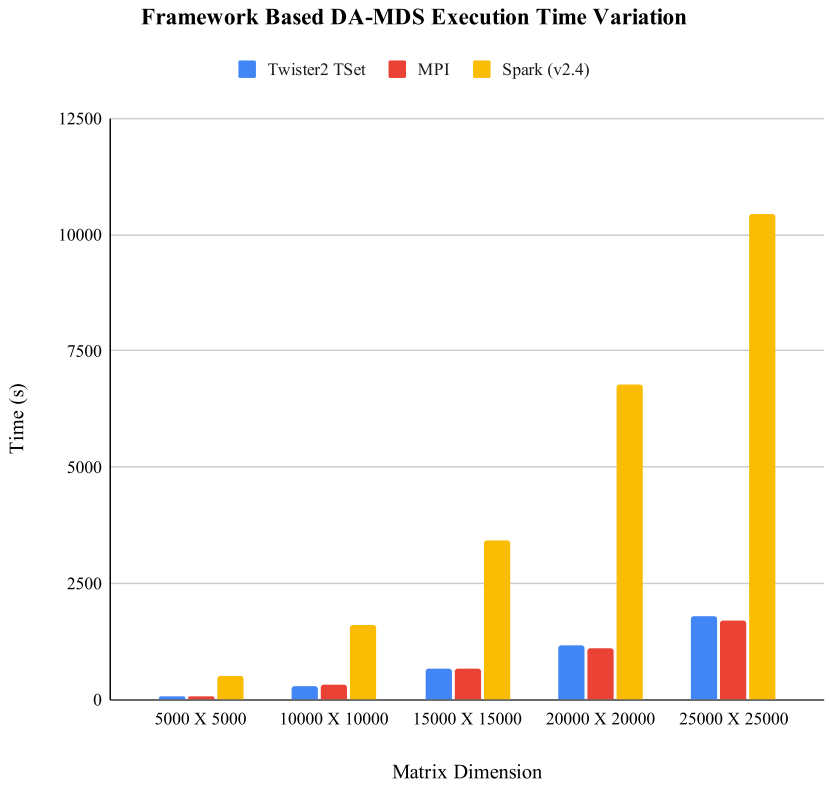


Figure 4.8: Execution time of DA-MDS with varying matrix sizes

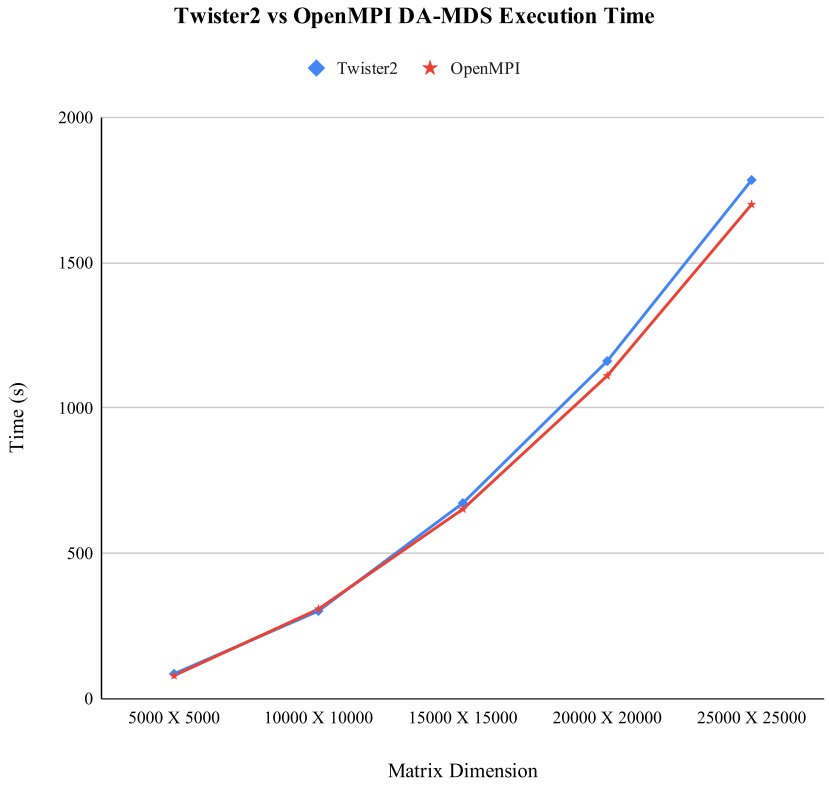


Figure 4.9: Execution time of DA-MDS with varying matrix sizes

4.6.3 Distributed SVM

Support Vector Machines (SVM) algorithm is a well known and widely used machine learning algorithm for classification. It is used by many researchers and scientists in various domain sciences. While SVM can be implemented using many different approaches, matrix decomposition methods, sequential minimal optimization-based methods, and stochastic gradient-based methods are the most popular approaches. Stochastic gradient-based methods have been proven to be efficient both in computation and communication complexity and are an ideal approach for distributed SVM implementations. For evaluations, the SVM algorithm was implemented using the stochastic gradient ensemble model. The ensemble model is not a highly

iterative algorithm at a distributed level, weight calculations are done each parallel process for a specified number of iterations, and as a final step, the results are communicated to synchronize the parallel work performed.

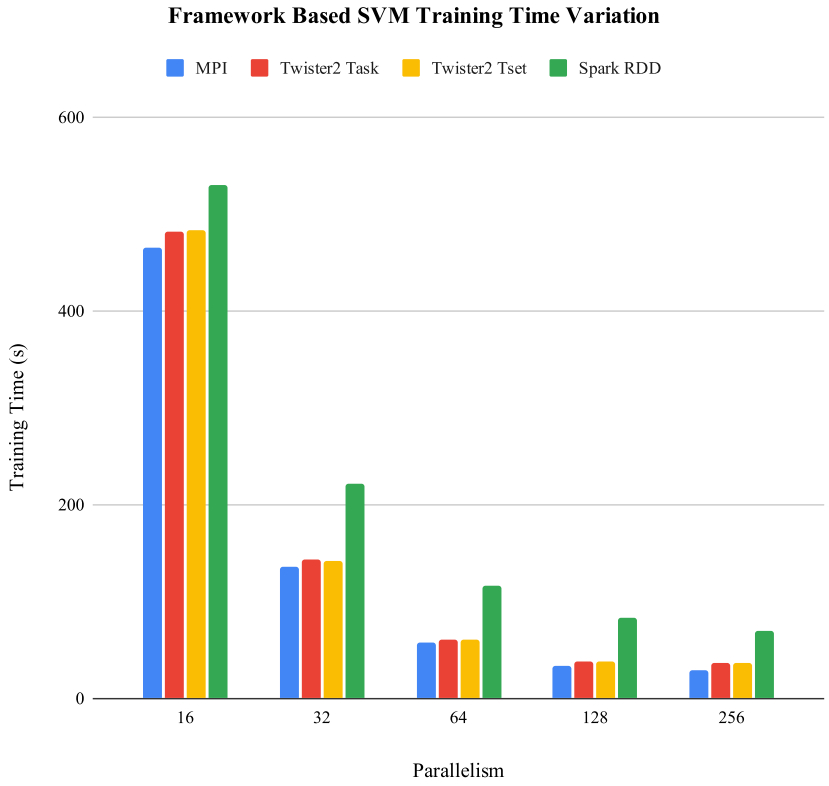


Figure 4.10: SVM job execution time for 320K data points with 2000 features and 500 iterations, on 16 nodes with varying parallelism

For evaluation, distributed SVM implemented using four frameworks. OpenMPI implementation of the Distributed SVM was done as the MPI standard-based implementation; MPI has become the defacto standard for parallel programming in the high-performance computing domain because of well-established efficiency. Model synchronization in the OpenMPI implementation is performed using highly efficient AllReduce communication primitive. The Apache Spark implementation of distributed SVM was done using the Spark RDD API; the final model synchroniza-

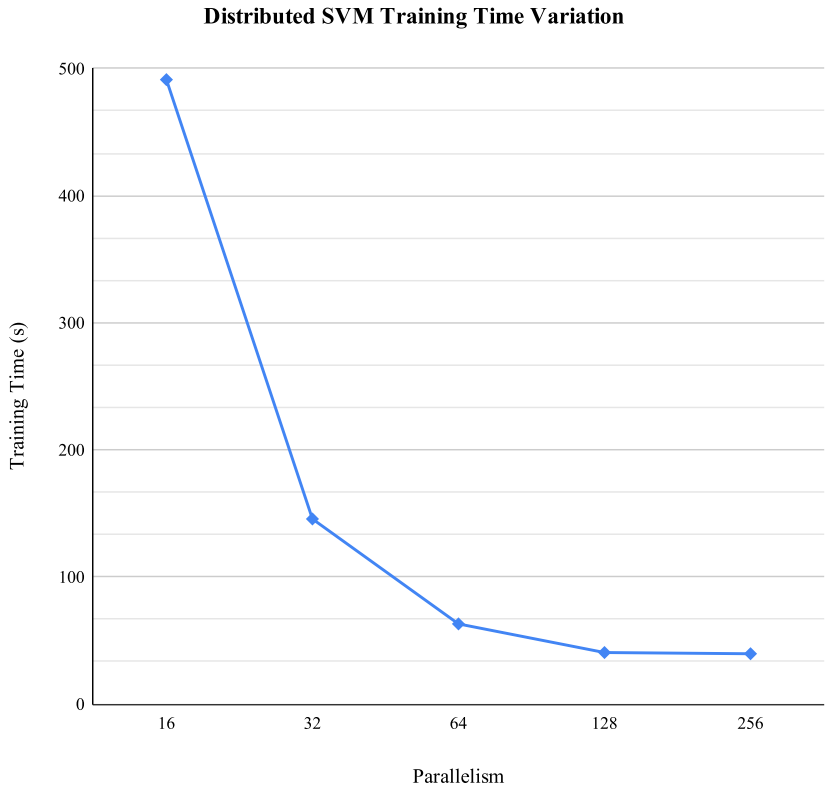


Figure 4.11: Twister2 TSet SVM job execution time for 320K data points with 2000 features and 500 iterations, on 16 nodes with varying parallelism

tion is done at the driver program by collecting the results from all the distributed executors at the end of training iterations. For Twister2, implementations were done using the Twister2 TSet API and the Twister2 Task API; The task API can be seen as a programming abstraction similar to Apache Storm. From the evaluation results in Figure 4.10 it is clear that Twister2 TSet implementation performs better than the Apache Spark implementation; both Twister2 TSet and Twister2 Task implementations are slightly slower than the OpenMPI implementation, which is to be expected. Since this algorithm only performs local iterations and only performs a single distributed collective communication, overheads due to distributed communications are minimal. Figure 4.11 show the execution time improvement

as parallelism increases. After parallelism of 128, the training time is dominated by communication times and other initialization times, which is why a significant improvement in training time is not observed, going from 128 to 256.

4.6.4 Dataflow Node

Evaluating the performance of various machine learning algorithms allows us to get a good understanding of how the framework performs in many different scenarios. However, it is also important to understand performance at a more fine-grained level. One such important test for a dataflow framework is to understand the overheads added to the runtime when a single dataflow node is added to the dataflow graph. To evaluate this, we created two dataflow graphs with the difference between the two graphs being a single map transformation that is essentially a dummy map transformation that simply forwards the input to the next node without any changes (a no-op transformation). The first graph is "source-map-allreduce" and the second is "source-map-map-allreduce". Similar graphs were implemented using Twister2 TSet API and Apache Spark RDD API to compare the results. Execution time was evaluated on 200K data points running for 100 iterations, for parallelism of 128 and 256. The results are shown in Figure 4.12 we can observe a slight overhead in the execution time in Twister2 and a much smaller overhead in Apache Spark. This is because Spark optimizes its map operations by pipe-lining consecutive map operations into a single operation. Even with the overhead, it is clear that Twister2 implementation performs almost an order of magnitude better than Spark. Operation pipe-lining optimizations and other related execution optimizations will be added to the Twister2 execution framework in the future to address these slight overheads.

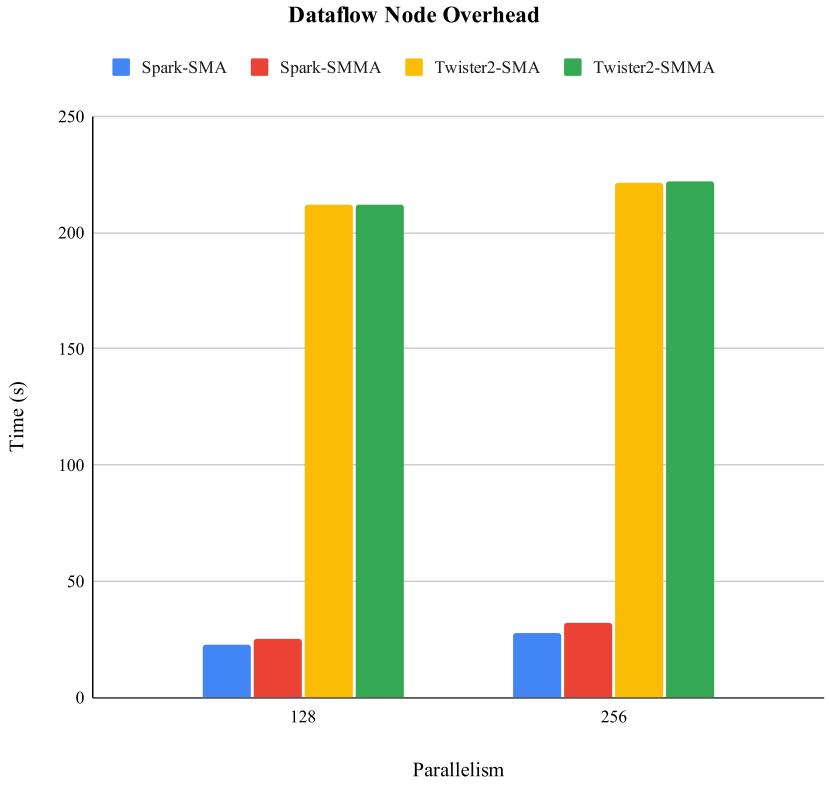


Figure 4.12: Execution time for Source-Map-AllReduce (SMA) and Source- Map-Map-AllReduce(SMMA) graph configurations. With 200K data points and 100 iterations

CHAPTER 5

DISTRIBUTED DEEP LEARNING - TWISTER2DL

With the immense growth of popularity and adaptation, deep learning has become one of the most researched areas in academia and one of the most sought after set of skills in the industry. Many companies and institutions have heavily invested research in this domain resulting in new deep learning models, new frameworks and improvements being done to existing frameworks and models. More and more data analytic tasks that were typically done using machine learning algorithms are now implemented using deep learning/neural network models because they are capable of producing better results than machine learning algorithms. Up until recently, deep learning was mostly utilized by larger technology companies running complex deep learning models, but with the rapid adoption of deep learning for many applications, more and more organizations have started using deep learning models for even relatively small data analytic tasks. This has led to increasing demand for using deep learning models in conjunction with distributed data analysis pipelines. Most of the popular deep learning frameworks such as TensorFlow and PyTroch mainly focus on the deep learning aspects and do not provide support for distributed data pre-processing; because of this, while they are well adapted for developing complex deep learning models, the data pre-processing would need to be done using a separate framework. Resulting in the need to move data between frameworks, which can be costly. The main goal of Twister2DL is to provide seamless integration between the high performance distributed data processing capabilities of Twister2 and deep learning. To this end, Twister2DL provides a deep learning framework that is built directly on top of the Twister2 TSet API and using Twister2 framework to achieve

distributed deep learning capabilities.

5.1 Motivations for Twister2DL

There are several motivating reasons for developing both deep learning capabilities and distributed data processing capabilities in a single framework, which was the main motivation for Twister2DL; they are as follows.

1. Input data processing
2. Data locality
3. Ease of distribution

5.1.1 Input data processing

The most popular method of testing the performance and accuracy of deep learning frameworks is to use some benchmark and the data set that accompanies that data set such as ImageNet[RDS+15], SQuAD[RZLL16], MNIST[LC10], etc. These datasets have already been curated and explicitly labelled as needed. Hence the users do not have to process the data before using them as inputs into the deep learning models to train/evaluate them. Using such curated datasets is justified when the goal of the research is to improve some deep learning model, to test the performance of a new framework or evaluate improvements made to an existing framework. In real-world applications, in many cases, the datasets are in raw form and need to be curated and labelled as needed. In batch data, the process might only need to be done a single time, but for streaming data the pre-processing steps would need to be done continuously. When a considerable amount of pre-processing needs to be done on the dataset before using it with the deep learning program, it makes sense to adopt a framework that provides both distributed data processing

capabilities and deep learning capabilities. Datasets used for deep learning tend to be large and raw data is typically even larger because they need to be cleaned and curated, therefore in most cases, the data pre-processing cannot be done on a single machine, which makes the desire for a distributed data processing framework even stronger.

5.1.2 Data locality

When working with large amounts of data, as done in most deep learning applications, moving the data around can be a costly operation, both in terms of time and money. The financial cost is especially important if cloud services are used since they typically charge based on the amount of data that is transferred. Since real-world data typically needs to pre-processed, if the deep learning applications are executed on a separate set of resources than where the data processing was done, the curated data would need to be moved between the two locations. In some cases, the pre-processing step may need to be performed multiple times while the application is going through development, costing both time and money. In such cases having the capability to perform the deep learning applications on the same framework will allow processed data to be used without the need to transfer data unnecessarily. Resulting in saved time and money.

5.1.3 Ease of use

Having deep learning capabilities in the distributed data processing framework makes it much easier to develop an end to end data analytic pipeline complete with deep learning models. The user does not have to worry about maintaining two clusters, one for data processing and another for deep learning. With Twister2DL,

all the cluster setup steps would be made easier since Twister2 supports several resource management frameworks such as Slurm, Kubernetes, Nomad, etc. Another important note to make is that while popular deep learning frameworks such as PyTorch do have distributed modes, they are notoriously hard to set up on a large cluster without expert knowledge.

5.1.4 Performance

There are currently several solutions that can be used to integrate data processing and deep learning. The first option would be to use the "connector approach". There are several connectors that have been written to enable deep learning on CaffeOnSpark[Cafb], TensorFlowOnSpark[Ten], TFX[BBC+17] and DeepSpark[KPJY16] are some such connector based solutions. The main drawback of such connectors is the boundary that exists between the two frameworks, even though they run on the same set of resources. Moving data between different framework results in inter-process communications, additional serialization/deserialization operations and even file operations for persistent. This causes significant overheads when executing an end to end application. The issues faced by connectors was mostly addressed for Apache Spark with the introduction of BigDL[DWQ+19] by developing a deep learning framework directly on top of the Apache Spark framework which is similar to the approach taken by Twister2DL with the Twister2 framework. However, since BigDL runs on Apache Spark, its performance is subject to the performance of the Spark framework, especially for communication operations. As shown in Twister:Net[KWG+18] and by the results shown in section 4.6 it is clear that much Twister2 boasts significantly higher performance numbers, hence developing a deep learning framework on top of Twister2 would allow Twister2DL

to leverage the superior performance to deliver a faster deep learning framework.

5.2 Twister2DL execution model

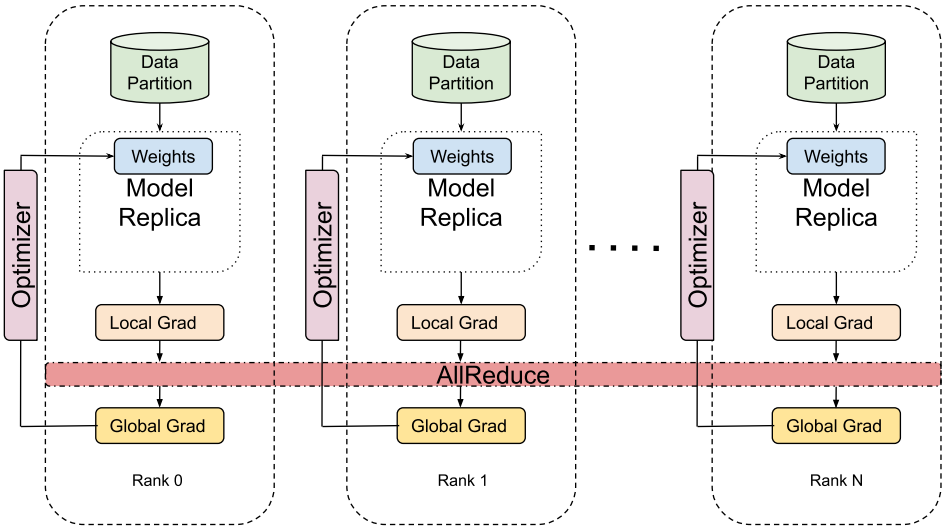


Figure 5.1: Twister2 execution model

Twister2DL implementation follows the basic strategies and principles of the BSP model to keep in line with the Twister2 framework. Figure 5.1 shows how distributed training is performed in Twister2DL. Currently, Twister2DL only supports data parallel mode, and hence the discussion will be based on the assumption that training is done using the data parallel mode. In Twister2DL each parallel training task is spawned as a Twister2 process, each will be assigned an global rank ranging from 0 to N-1 (for a parallelism of N). Each process will have its own replica of the model that needs to be trained. Unlike framework such as BigDL[DWQ+19] the model does not need to be broadcast from a central driver/master program; since Twister2 follows the BSP model, each parallel worker will build the model replica locally. Once the data partition is loaded, each parallel process will calculate the local gradient values for the model based on the input data partition. Once all the

parallel processes complete the local gradient calculation, an AllReduce operation will be used to aggregate all the local gradients and calculate the global gradient. Since this is an AllReduce operation, once it completes, each process will have identical copies of the global gradient. Next, the weights of the model will be updated using the optimization method specified by the user and the global gradient. This process iterated until the conditions set by the user is met. The stopping conditions may be some error threshold or a specific number of iterations/epochs. Once the training is complete, the model will be saved/persisted at the process with rank 0. The pseudo-code for how data parallel training works in Twister2DL is listed in Algorithm 3

Algorithm 3: Twister2DL data parallel training

```
1  // N parallel workers;
2  for rank=0 to N - 1 do
3      build model;
4      while stopping condition is incomplete do
5          read latest weight;
6          read data from data partition;
7          perform forward-backward to calculate local gradient;
8          call AllReduce to calculate global gradient;
9          update weights according to optimizer method;
10     end
11     if rank = 0 then
12         output/save trained model;
13     end
14 end
```

5.3 Twister2DL implementation

The implementation of Twister2DL is done entirely using the Twister2 TSet framework. Figure 5.2 shows the dataflow graph that is associated with the training phase of a deep learning model. The input data is structured as mini-batches or samples and taken in as the starting point of the dataflow graph in the form of a SourceTSet. The framework automatically distributes the data to all the parallel instances that are being executed. Each data partition is cached locally to improve the performance of the training operation since the data will be reused when running several epochs. The modal that needs to be trained is also handled as a TSet; however, in this case, each distributed instance of the TSet will have identical values. The model is also cached to make sure the model is not built for each iteration. To perform the actual training of the model, a special map operation is called on the input data. This map operation contains the logic to perform the forward and backward passes on the model; the cached model TSet is added into this map transformation as an input. Once the iteration is completed, and the local gradient values are calculated, the framework calls an AllReduce operation to aggregate all the local gradient values. The results of the AllReduce operation is fed into a final map transformation which calculates the final global gradient required for the next training iteration. Finally, each parallel worker updates the weight values of the local model based on the optimizer that is specified by the user. If the stopping conditions are met, the model can be saved for inference. If not, the next iteration of the training is performed. For the next iteration, the map transformation will have access to the updated weight parameters. In order to optimize the performance of the training phase, the framework relies upon the highly optimized Intel MKL and Intel MKL-DNN libraries to perform core computations.

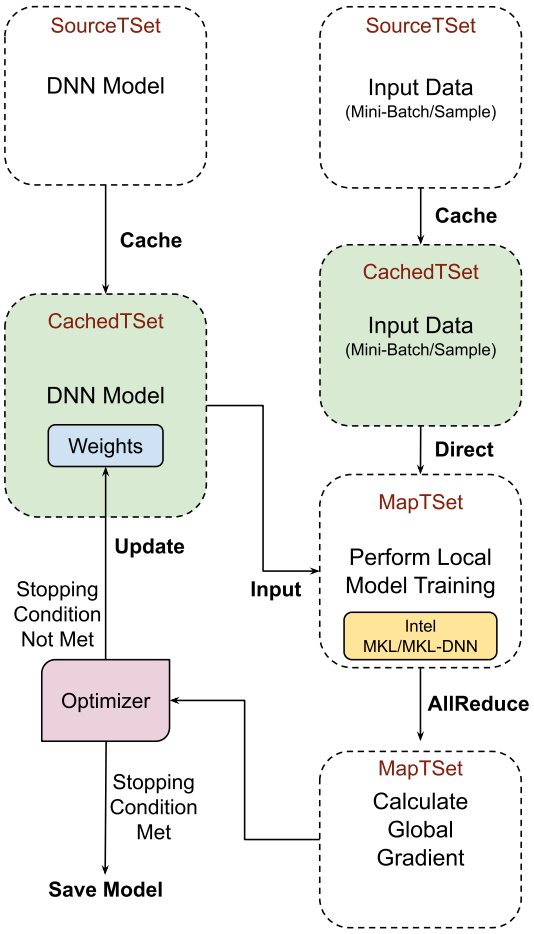


Figure 5.2: Twister2 Dataflow graph

5.3.1 Tensors

Tensors are a generalization of vectors and matrices, which are widely used in the machine learning and deep learning domains. Most deep learning frameworks support tensors as a core part of the framework. In many frameworks, tensors are the main building block and the main medium used to store input data and parameters within the framework. In order to have an efficient framework, the tensor operations should also be highly optimized since the bulk of the calculations are done through tensor arithmetics. Twister2DL supports tensors natively to make sure the tensor operations do not carry any unwanted overheads. The tensor is modelled after

the tensor package in Torch[Tor]. In order to obtain the best performance for the tensor operations, Twister2DL leverages Intel MKL[inta] library to perform highly optimized arithmetic operations.

Tensors are essentially n-dimensional data structures that contain data from a single scalar type such as Int, floats, double, etc. Following the models defined in the Torch[Tor] tensor package, tensors in Twister2 consist of two main parts, data and metadata that describe the structure of the data. Because of how they are constructed, tensors can represent standard types such as vectors and matrices straightforwardly, as shown in Figure 5.3.

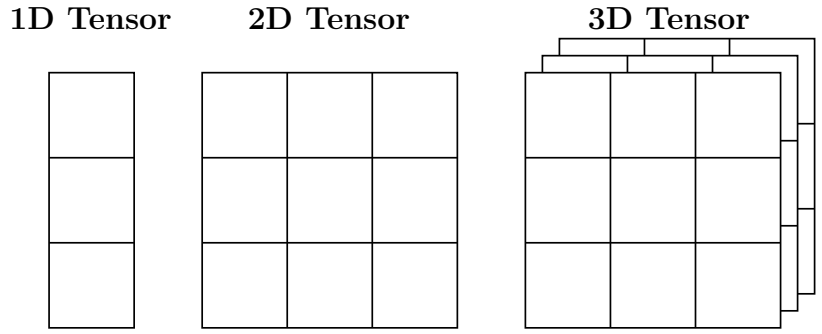


Figure 5.3: Data structures represented using Tensors

Tensors in the twister2 tensor package are built up of several key components; the separation of data and meta-data of the tensors allow the framework to easily recompose tensors into different layouts and views. In order to reshape a tensor, the framework only needs to alter the size and stride metadata of the tensor which means no data copying needs to be performed, saving valuable compute and memory resources.

1. data - A 1D array of either floats or doubles, this contains the actual data of the tensor.
2. size - A array that contains the sizes of each dimension of the tensor, For example a 3D tensor will have an size array such as *[Depth, Height, Width]*

3. stride - A array t that defines how the data layout is defined in physical memory and helps calculate the location of tensor values when needed. For example if the data is stored contiguously in memory the stride array would be *[Height*Width, Width, 1]*. The "1" at the final index means that each element is placed contiguously in memory.

Tensor Operations

Tensor operations are the core computational kernels that perform all the required calculations for model training and inference. Therefore it is vital that tensor operations are highly optimized. To this end, Twister2 supports optimized implementations of tensor operations for both floating-point precision operations and double-point precision operations. It is important to have separate implementations because the number of floating-point operations done in floating-point precision calculations are half of what is done for double point precision calculations. Internally to boost performance, the framework leverages highly optimized math kernel operations provided by Intel MKL [inta] which will be discussed in more detail in section 5.3.2.

5.3.2 Optimized Kernel Operations

For deep learning workloads, the bulk of the computation time is spent on vector operations. Therefore improving performance on vector operations can significantly improve the performance of deep learning frameworks. This is why libraries such as BLAS play a significant role in deep learning frameworks. In order to improve math kernel operations in Twister2DL internal math operations are implemented using Intel MKL and Intel MKL-DNN libraries. Both these libraries are geared

towards optimizing CPU operations which aligns with the computation requirements of Twister2DL

Intel MKL

Intel MKL, recently named as oneAPI Math Kernel Library[inta] is a highly optimized math library. It provides enhanced math routines to improve the performance of applications and frameworks; this is especially important for machine learning and deep learning frameworks. BLAS, LAPACK, vector math and random number generation functions are among the highly optimized core functions that are provided by the Intel MKL library. The library is specially optimized for Intel hardware to obtain the maximum possible performance from Intel hardware. The tensor operations implemented in Twister2DL use Intel MKL extensively, especially for vector and matrix operations. Intel MKL library is written in C++; therefore, Twister2DL connects to the library through a Java binding that is provided by Intel. The kernels calls do not require any additional processing; therefore is easy to use and integrate with the framework.

Intel MKL-DNN

Intel MKL-DNN[intb] which has been recently renamed to Intel oneAPI Deep Neural Network Library (oneDNN) [intb] is a library specifically developed to enhance the performance of deep learning frameworks. This allows the same API's to be used regardless of the hardware that it runs on. The hardware that it runs on can be CPU's or GPU's or even other hardware devices once support is added in the library. It provides a set of primitives which can be used in deep learning frameworks. And within the library, these primitives have been highly optimized to run on Intel hardware. Functions such as activation functions, batch normalization,

convolutions, etc. are provided by the framework. This allows developers to focus on the other aspects of the deep learning framework.

One drawback when using the Intel MKL-DNN library is that it requires custom memory layouts to be implemented on the framework side in order to be used. Tensor operations that used Intel MKL internally cannot be simply re-routed to use the math kernels provided by Intel MKL-DNN. Unlike in MKL, MKL-DNN implements higher-level kernel operations specifically built for deep learning. Element-wise operations such as RELU, Normalization layers and Convolution layers are some of the kernels supported by MKL-DNN.

In order to support MKL-DNN, a conversion layer was added in Twister2DL; once the user defines the model the framework builds an intermediate representation to convert the model into a layout that can be used with MKL-DNN. The execution graph is built using the intermediate representation. The intermediate representation is also responsible for building the memory layouts that are needed by MKL-DNN; once the memory layouts are built, relevant kernels can be invoked.

5.3.3 Forward and Backward Propagation

When looking at how neural networks are designed and trained the flow can be broken down into the following steps.

1. Define Model network
2. Forward propagate through network with input
3. Calculate loss based on loss criterion
4. Backpropagate starting from the loss to calculate the gradients for each layer
5. Update weights based on learning rate and gradient

6. Iterate from the start till end condition

In twister2DL forward propagation and backward propagation is handled by generating a forward and backward execution graph based on the model that is defined by the user. Each layer internally has the computation logic defined for the forward calculation and the backward gradient calculation. The automatically generated execution graph for gradient calculation can be thought of as a layer level "autograd" similar to the full-blown autograd systems available in frameworks such as PyTorch, but less versatile.

5.3.4 Implementation Challenges

Twister2DL being built directly on top of the Twister2 TSet framework removes many of the complexities that generally come with developing a distributed framework. This is further helped along by the fact that Twister2DL is developed following the BSP model. Because of these two factors, Twister2DL needs very little implementation detail to handle distributed workloads and the distributed communication operations (AllReduce). However, developing an extensible framework for deep learning comes with its own set of unique challenges which need to be addressed. The first, and arguably the most important in respect to usability, is how the code is structured and how the extension points are defined. Deep Learning is an ever-evolving field, and new network models are being introduced frequently. New network models mean new network layers that need to be supported by the framework. To adapt with such changes, Twister2DL needed to provide a simple but effective set of extension points that users/developers can leverage to add new layers, which in turn adds support for new network models. Appendix ii shows some of the extension points provided by the framework. Performance optimizations and

all the detailed work that went into achieving them is another area that proved to be very challenging during implementation. To improve the performance of tensor operations, Twister2DL relies on the Intel MKL libraries. The MKL library is rather straightforward to use once the tensor package is in place. All that is left is to invoke the required math routine based on the tensor operation. However, the challenging part comes when further optimizations are performed using the MKL-DNN library. MKL-DNN required inputs to follow certain memory formats. To allow this without having two separate implementations for MKL and MKL-DNN for each layer type, the framework converts the layers through an intermediate state so they can be run with MKL-DNN, this conversion and making sure the input data is re-ordered according to the required memory formats proved to be very challenging. Finally, each layer requires all the tensor operations required for the mathematical calculations to be accurate. Implementing each layer and making sure the calculations were correct was also a challenging task to do with limited resources.

Appendix ii contains some code examples that showcase code segments to illustrate the implementation complexities to some extent. The codebase that was written for Twister2DL contains approximately 32,000 lines of code.

5.4 Twister2DL Programming Interface

In order to make the process of building a deep learning model as smooth as possible, the programming interfaces provides easy to understand constructs that follow the existing popular frameworks as closely to make the transition easy. In order to perform training using the Twister2DL framework, users need to define the following main components. We will look at how each of these are defined using a simple autoencoder network as an example. The code for constructing the autoencoder

code and training it is defined in Algorithm 4, The code segment does not list all the details of the actual code, only the lines needed for the explanation.

1. Optimizer Type - Distributed or Local
2. Input Data
3. Network model
4. error criterion
5. optimizer method - Adam, Adagrad, etc.
6. stopping condition

5.4.1 Optimizer

The optimizer represents the execution unit that takes in the model, data and other relevant information to perform the training and inferencing operations. Internally the optimizer handles the details of how the framework organizes and executes the deep learning application. Currently, Twister2DL supports two optimizer implementations. They are namely local optimizer and distributed optimizer.

Local Optimizer

The local optimizer is made available to the end-user mainly for testing and debugging purposes. This optimizer is lightweight and can be used to run test models or small models on a single machine or laptop device. The local optimizer does allow users to run parallel training, which is handled internally using the Twister2 standalone cluster mode. It is not advised to run large models using the local optimizer since it is not optimized or designed to handle such workloads.

Distributed Optimizer

The distributed optimizer has been designed and developed to run large deep learning applications on compute clusters. The optimizer is designed to run on the Twister2 cluster to which the deep learning job was submitted. Internally the distributed optimizer handles all the required model and local data caching to improve runtime performance of the application. In addition to that, several runtime optimizations are programmed into the distributed optimizer. One such optimization is the usage of a custom data packer. Data packers are extensions supported by Twister2 which allows users to define how data needs to be packed when serialization and de-serialization are performed on data before it is sent over the communication channel. This helps reduce the size of the data being transmitted through the communication channel and/or reduce the time that the frame takes to perform the serialization and de-serialization operation; both are typically expensive operations. The distributed optimizer leverages this feature to define how the local gradients are to be packed to reduce communication overheads.

5.4.2 Input Data

In order to perform distributed data-parallel training efficiently, the framework needs to handle input data efficiently. The framework is responsible for assigning data partitions to each parallel process and making sure each process only loads the allocated data partition. Local data also needs to be cached in-memory whenever possible to avoid the need to read data from stable storage multiple times. File operations are generally extremely expensive operations and can slow down the application.

In Twister2DL input data is handled through extensions that are built on top of the Twister2 TSet API. All input data is handled as special SourceTSet's. The framework provides several convenience methods through a data set factory API which natively supports data types such as CSV files and other delimited data files. The API also provides the ability for the user to define custom implementations for data loading by extending extension interfaces that are supported by Twister2 TSet API. The Twister2 framework internals will automatically handle details on how data is partitioned and distributed among all the parallel workers. Once the data is read from the raw data source, they are converted into either "Sample" or "Minibatch" sourceTSet's. This required since the optimizer expects data to provided as one of these two options. When using the provided data API methods, these steps are also handled transparent to the user.

Sample

A 'Sample' represents a single data point in a deep learning input dataset. A sample is made up of a feature tensor and a label tensor. For example, a single image and the associated category is a sample. In most cases, it would make sense to use the 'Minibatch' data type.

Minibatch

A 'Minibatch' represents a batch of input data, similar to the sample, the minibatch is also constructed with a feature tensor and a label tensor. The difference between sample and minibatch is that the minibatch contains an extra dimension in the tensor that corresponds to the minibatch size. For example, if the minibatch is set to 32 in an image data set, the minibatch data object would have the first dimension size of 32 and the other dimensions to represent the image as required.

5.4.3 Network Model

The most important component of the deep learning application is the network model. Twister2DL API allows users to define the network model by defining a set of layers and stacking them up to create the network model. The list of layers that are currently supported in Twister2DL is listed in Table 5.1. In addition to the currently supported layers, the framework provides well-defined interfaces that can be extended to add new layers into if needed. There are many more layers that need to be implemented in order to make the framework more complete. Implementing layers require more engineering work and will be added in the future as the framework is adopted and developed. Table5.1 also indicates if the layers are supported with MKL and/or MKL-DNN libraries. Currently, only a handful of layers are supported for MKL-DNN because of the implementation complexities involved in developing MKL-DNN supported layers.

5.4.4 Error Criterion

The error criterion a.k.a loss criterion is used evaluate the candidate solution at the end of the forward propagation. The loss calculated using the loss criterion is then used to perform backward propagation on the network. TwisterDL provides several error criterion's which the user can use out of the box. And also provides extension interfaces to implement custom loss functions when existing criterion are not sufficent.

5.4.5 Optimizer Method

The optimizer method refers to the method that is used to update parameters as the model training is performed. Gradient descent has become the defacto stan-

Layer	Description	MKL	MKL-DNN
Sequential	A special layer that is used to construct the network by concatenating layers sequentially	true	true
Reshape	Used to reshape input tensors as needed, such as reshaping an image into a vector	true	true
Linear	Applies a linear transformation on the input data	true	true
Relu	Applies rectified linear unit function element-wise on the input	true	true
LeakyRelu	Similar to Relu function with the addition of a small positive gradient when the unit is not active	true	true
Logsoftmax	Applies logsoftmax function to the input data	true	false
Sigmoid	Applies the Sigmoid function element-wise to the input data	true	false
Dropout	Sets sections of the input data to zero using a Bernoulli distribution. Each input element is has a probability P of being set to zero	true	false
SpatialConvolution	Applies 2D convolutions on the input data. Input data is asumed to be 4D or 3D and the convolution is applied to the last two tensors. Padding can be defined to handle the edges	true	false
SpatialMaxPooling	Applies 2D max pooling operation on the input data	true	false

Table 5.1: Twister2DL supported layers and available implementations

dard for neural network optimization as well as for many other machine learning algorithms. Various frameworks implement different optimized versions of gradient descent, some well known implementations of gradient descent for deep learning are Adam, Adagrad, Adadelta, etc. While no one implementation is better than other for all use cases, each have strengths and weakness of their own. In general gradient descent has 3 variants. Which are described briefly below.

Batch gradient descent

In batch gradient descent each step takes all the data points in the dataset into consideration before the parameters of the model are updated. This is a very slow optimization method especially for large datasets since each iteration needs to process all the data in the dataset. However batch gradient descent results in a very smooth gradually decreasing graph for the cost/loss when mapped against epochs. Batch gradient descent is guaranteed to reach a global minimum for convex error surfaces and will reach a local minimum for non-convex error surfaces. However in practice batch gradient descent is rarely used for large datasets because of it takes a relatively long time to converge. The other two gradient descent are preferred over batch gradient descent most of the time.

Stochastic gradient descent

Stochastic gradient descent (SGD) can be thought of as the polar opposite of batch gradient descent. In stochastic gradient descent for each data point we pass it through the network and calculates the gradients to update the parameters before the next data point is processed. With stochastic gradient descent the cost/loss does not gradually decrease, this is because parameters are update after processing each data point. As a result the cost/loss keeps fluctuating as the parameters are updated with each iteration and step. Therefore it is not guaranteed that this would method would reach a local minimum. In the long run it will generally decrease the cost/loss and converges faster especially for larger datasets. stochastic gradient descent is also applicable to online learning applications. It has been shown that the fluctuating nature of the SGD can be mitigated to some extent by decreasing the learning rate as training progress.

Mini-batch gradient descent

Mini-batch gradient descent can be placed in between batch gradient descent and SGD. With mini-batch gradient descent n data points are chosen at a time and at each step the parameters are updated based on those n data points. This removes results in smoother cost/loss decrease compared to SGD. Another major advantage of mini-batch gradient descent over SGD is that in the implementation, frameworks can utilize vectorized implementations for forward and backward calculations. This can boost the performance of the training significantly. Because of mini-batch gradient descent combines the best of both worlds it is the most widely used gradient descent variant of the 3.

However plain mini batch gradient descent does not guarantee good convergence due to several key challenges.

1. Identifying the correct learning rate can be difficult, A small learning rate can result in a slow convergence and a large learning rate can cause the cost/loss function to fluctuate and even diverge. Learning rate schedules [DCM+92] are able to mitigate this issue to some extent by changing the learning rate as training progresses, however these generally need to be defined in advance and are not learnt based on the dataset.
2. The learning rate applies to all parameters equally. This might not be ideal for some scenarios, For example if the training is done with sparse data, it would be desirable to apply higher learning rates to features that appear less frequently.
3. As with SGD, mini-batch gradient descent can get stuck in local minima, for non-convex error surfaces.

In order to tackle the challenges mentioned above and to improve various other factors of mini-batch gradient descent many optimized mini-batch gradient descent algorithms have been in introduced by various frameworks. Twister2DL implements several such popular algorithms.

Adam

Adaptive Moment Estimation (Adam) [KB14] is a well-known gradient descent optimization algorithm; it is simple to implement and had very little memory requirements. Implementations of Adam is available in most deep learning frameworks due to its popularity. Adam also calculates adaptive learning rates for each parameter which helps address issues with the plain mini-batch gradient descent model. It also uses and keeps track of an exponentially decaying average of prior gradients. In [KB14] the authors show that the algorithm works well in practice compared to other gradient optimization models.

Adagrad

Adagrad[DHS11] is another well-known gradient descent optimization algorithm that is implemented in many frameworks. Adagrad is especially suited for sparse data because it adapts the learning rate for parameters based on the occurrence frequency of features related to a given parameter. For example, adagrad will assign higher learning rates for parameters associated with features that infrequently occur in data and assign lower learning rates to parameters associated with features that frequently occur in the data.

Adadelta

Adadelta[Zei12] is an extension of adagrad algorithm, which aims to build upon adagrad by fixing a few inefficiencies of adagrad. Adadelta aims to reduce the fast, monotonically decreasing nature of the learning rate in adagrad. Adadelta restricts the number of past gradients that are accumulated as opposed to adagrad, which has no such limit.

RMSprop

RMSprop[HSS12] was introduced by Geoffrey Hinton; it is not a published optimizer. However, RMSprop has gained popularity and is implemented in many deep learning frameworks. Similar to Adadelta, RMSprop aims to solve the diminishing learning rate observed in Adagrad.

5.4.6 Stopping condition

The stopping condition defines when the training iterations should stop. Currently Twister2DL supports 3 stopping conditions.

1. Max Iteration - stop training once a maximum number of iterations is reached
2. Min Loss - stop training when the loss value reaches the defined min value
3. Max Epoch - stop training if the maximum number of epochs is reached

Algorithm 4: Twister2DL data parallel training for Autoencoder

```
1  // Create input data
2  SourceTSet<MiniBatch> source =
     DataSetFactory.createMiniBatchDataSet(env, dataFile, miniBatchSize,
     dataSize, parallelism);
3  // Define network model
4  Sequential model = new Sequential();
5  model.add(new Reshape(new int[]{features}));
6  model.add(new Linear(features, l1));
7  model.add(new ReLU(false));
8  model.add(new Linear(l1, l2));
9  model.add(new ReLU(false));
10 model.add(new Linear(l2, l1));
11 model.add(new ReLU(false));
12 model.add(new Linear(l1, features));
13 model.add(new Sigmoid());
14 // Define loss/error criterion
15 AbstractCriterion criterion = new MSECriterion();
16 // Define the Optim method
17 OptimMethod optimMethod = new Adam();
18 // Define optimizer with the details
19 Optimizer<MiniBatch> optimizer = new DistributedOptimizer(env, model,
     source, criterion);
20 optimizer.setOptimMethod(optimMethod);
21 optimizer.setEndWhen(Triggers.maxEpoch(epoch));
22 optimizer.optimize();
```

Line 2 creates the input dataset using the "DataSetFactory" class. To create an input dataset you need to pass in several parameters to the function; the exact parameters may differ based on the API call. "env" is the Twister2 environment that is used during the execution, this is an object that is created for any Twister2 application. Next, "dataFile", "miniBatchSize" and "dataSize" are values needed to create the soureTSet, as the names suggest these specify the data file (CSV file in this case) the size of the mini-batch which is to be used during training and the total

size of the input dataset. The final parameter which is passed in is the parallelism, this dictates the number of parallel tasks that the execution happens with. Lines 4-13 defines the network model, which is an autoencoder in this case. Twister2DL has a set of defined layers that the user can use to develop the network. If a layer that is needed by the user is not defined in the current API, Twister2DL has been developed in an extensible manner that the end-user can add custom layers to the framework by extending the proper API interface classes. If the features, l1 and l2 are set to 12, 8 and 3, respectively, the network created in Lines 4-13 is illustrated in Figure 5.4. Line 15 defines the error/loss criterion to be used to calculate the error of the network, for this mean squared error is used. In line 17, the optimizer method that is used to update the weight parameters based on the calculated gradient is defined. Finally, all the details are passed to the DistributedOptimizer at lines 19-21. Twister2 supports two optimizers which are the LocalOptimizer and the DistributedOptimizer. The local optimizer is suitable for debugging and initial tests on a single machine. The DistributedOptimizer runs on the Twister2 cluster to which the job is submitted. At line 21 the code sets the stopping condition which will be used to determine when the training phase will be completed. In this example, it is set to the given number of epochs. Once all the required objects have been defined, the training process will start once line 22 is executed.

5.5 Evaluations

In order to test the efficiency of Twister2DL, several tests where done and compared against Intel BigDL[DWQ+19] and the popular PyTorch framework. The comparisons are made against Intel BigDL since it is a popular framework that provides similar functionality to Twister2DL, that supports an end to end distributed data

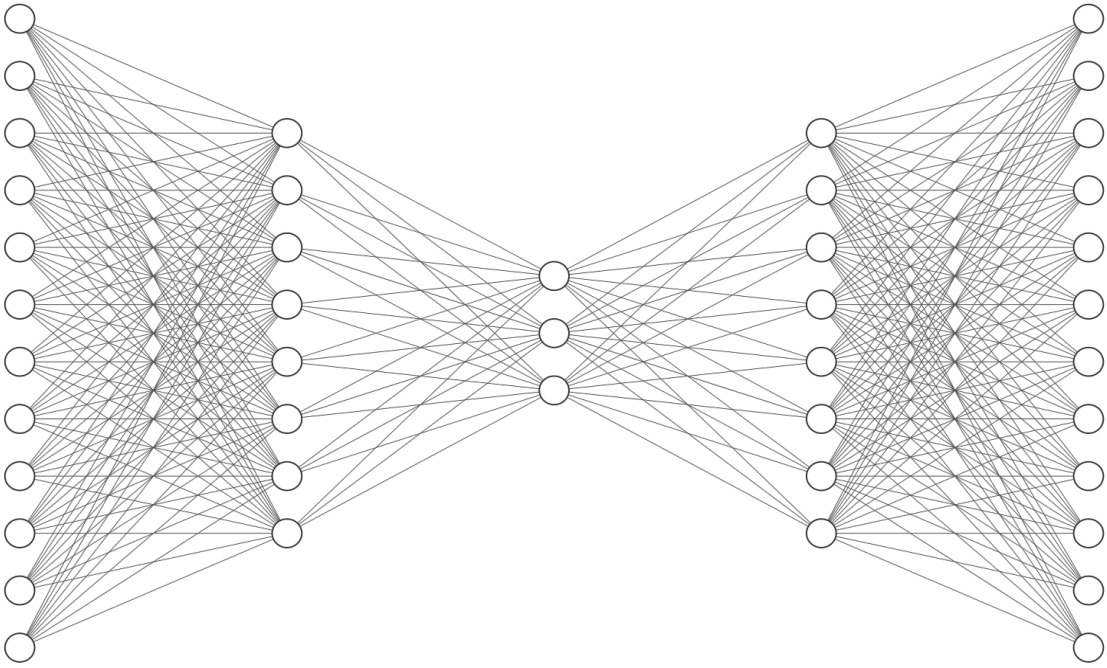


Figure 5.4: Simple Autoncoder network layers

processing pipeline including deep learning. Twister2DL does not currently support GPU's which is also the case for Intel BigDL. For more transparency and insight, we also evaluated against PyTorch since it is the most prevalent deep learning framework in the domain. However, it is important to keep in mind that PyTorch only provides deep learning capabilities and does not provide distributed data processing capabilities; therefore, it does not have any of the dataflow related overheads. Additionally, running distributed data parallel workloads with PyTorch require manually created and executed scripts on each node, in both Twister2DL and BigDL, a single command will transparently handle the parallel execution of workers on each node. Evaluation runs were done on a cluster with 12 nodes of Intel Platinum processors with 48 cores in each node, 56Gbps InfiniBand and 10Gbps network connections. Each value presented is obtained through an average of 3 independent runs.

5.5.1 The effect of programming languages on runtime performance

Before we look into the performance number between different frameworks, it would be worthwhile to take a quick look at the effect different programming languages have on runtime performance of a framework. In the context of this research, the main languages that are involved are Java, Scala, Python and C++. Java is the main language used to develop Twister2 and Twister2DL. Scala is the core language used in Intel BigDL. Python and C++ are the core languages used on PyTorch. Twister2 and Spark both do have Python API's, but the core runtime is not implemented using Python. First, looking at Java and Scala, they are quite similar in construct; both run on the JVM hence have comparable performance in most cases; therefore, we will mainly compare Java, Python and C++.

While each language has its pros and cons, in term of runtime performance Java and C++ are considered to be faster languages than Python. One of the main aspects working against Python is that it is an interpreted language, therefore unlike Java and C++, which compile the code into an intermediate state one time, Python has to perform this for each run which adds a large overhead. Between Java and C++ the comparison is much more nuanced; while one may be faster in some aspects, the other can be faster in others. For example, in [EKWF16] the authors show that Java can obtain C level performance with proper optimizations like the use of JIT. However, memory management of Java using the garbage collector is considered to be less efficient than what can be achieved in C++. Since C++ allows for more fine-grained control in areas such as memory management in general, highly optimized code can be written in C++ that can outperform Java. This is especially important in deep learning frameworks to develop highly optimized kernel codes. In

order to achieve comparable performance, most if not all Python-based frameworks rely on native code libraries and kernels written in C++ to do most of the heavy lifting. This is the strategy used by PyTorch, while the framework is written in Python, it mostly acts as a glue code, and all the computationaly heavy tasks are done using C++ code; therefore performance-wise, PyTorch does not have any drawbacks caused by Python.

Twister2DL also leverages kernel code libraries written in native C++. This is done through the use of Intel MKL libraries which are written in C++. This allows Twister2DL to alleviate some of the overheads caused by the JVM during runtime. However, all the other dataflow features are implemented on pure Java therefore, the framework does suffer from some overheads caused by the JVM. This is made evident to some extent with Cylon [WPA+20], which is a framework developed by the authors as a follow-up project to Twister2 with a C++ core; Cylon provides a subset of Twister2 capabilities. For the supported operations, Cylon is able to outperform Twister2 due to the excellent memory management and highly optimized code.

5.5.2 Autoencoder

Autoencoders are a well-known network model used in deep learning for various tasks. To evaluate the performance of Twister2DL, we implement a couple of simple autoencoders in both Twister2DL, BigDL and PyTorch to observe the execution times.

The first experiment was done to evaluate how each framework performs with increased workloads in a distributed setting. The training was performed on 10 compute nodes, with each node running four processes, amounting to a parallelism of

40. The autoencoder had 9 layers having $1024, 768, 576, 432, 324, 432, 576, 768, 1024$ units in each layer respectively. Relu was used as the activation function for the autoencoder. Training was done with Adam as the optimizer and mean square error as the loss criterion. The data size increased from 320K data points to 1280K data points. The mini-batch size is set to 8000 data points in all the runs. The training is done for 100 epochs. Each runtime depicted in the graph is taken from an average of 3 identical runs. The results of this evaluation are summarized in 5.5. From the results it is clear that Twister2DL performs better than Intel BigDL framework; for 320K data points BigDL takes roughly 40% more time. PyTorch performs better than both Twister2DL and BigDL; this is expected since PyTorch is a framework that is dedicated for deep learning and implemented using optimized c++ kernels. However, being able to perform with only a 20-25% execution time difference while providing all the dataflow capabilities mentioned in earlier sections is encouraging.

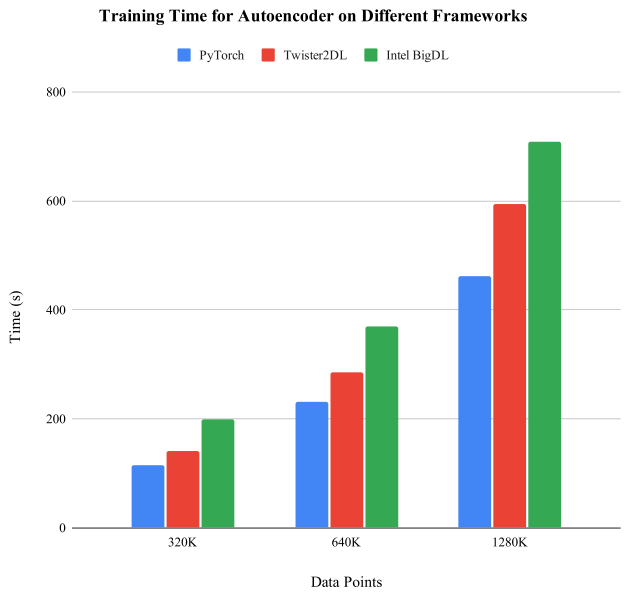


Figure 5.5: Training time for varying data sizes with different frameworks for 9 layer autoencoder. 100 epochs, mini-batch size 8000, parallelism 40

Twister2DL is able to further improve training performance by leveraging its MKL-DNN support. Figure 5.6 show the evaluation results, which also include executions done with MKL-DNN enabled in Twister2DL. While Pytorch does support MKL-DNN to optimize computations, currently, that features does not work for distributed data parallel (DDP) jobs. BigDL also supports MKL-DNN; however when executing the experiments with the setting enabled, execution time increased. We suspect this might be caused by some bug in the BigDL framework since, theoretically, it should improve performance; therefore, the results of BigDL with MKL-DNN enabled are not presented in the evaluations. As seen in the results, MKL-DNN is able to provide an additional 10-15% performance improvement for Twister2DL. For 1280K data points Twister2DL only takes a 12.6% performance hit compared to PyTorch, while BigDL takes a 53.0% performance hit.

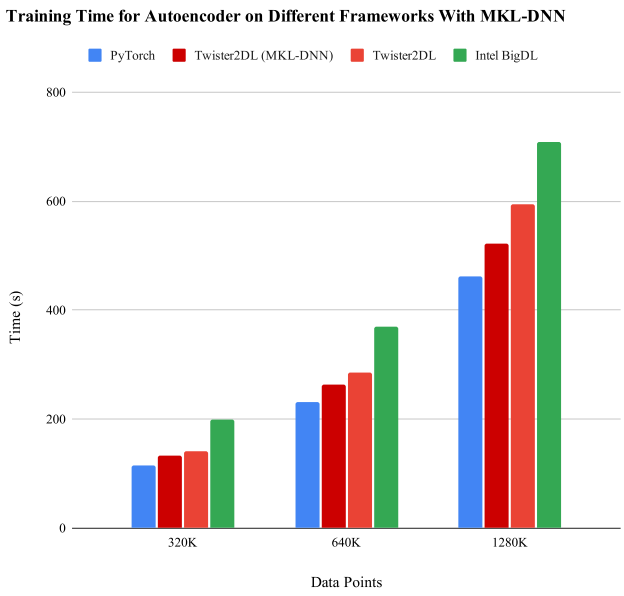


Figure 5.6: Training time for varying data sizes with different frameworks for 9 layer autoencoder. 100 epochs, mini-batch size 8000, parallelism 40

The two main components that contribute to the training time is the computa-

tion time and the framework overheads. Compute time refers to all the calculations that need to be done for forward propagation, backward propagation and other gradient calculations. The framework overheads include the time taken to communicate between parallel processes to synchronize and share intermediate data. In deep learning, local gradients which are shared among parallel workers at each iteration. Figure 5.7 shows a breakdown of the two components for each framework. As expected PyTorch has minimal overheads since the only overhead it incurs is the gradient communication. For Twister2 and BigDL it is slightly more complex. Since both frameworks support dataflow model, there are several layers that are in place between the application logic and the execution to support the dataflow model. These layers add more overhead during execution time. First, looking at the compute times, all frameworks have roughly equal compute times, Twister2DL is written in Java which is generally slower than c++ which is used in PyTorch for the computation kernels; however, the use of MKL libraries have enabled the framework to match the computation performance of PyTorch, with MKL-DNN enabled Twister2DL is able to outperform PyTorch compute time by roughly 12-14% . Looking at the overhead times, Twister2DL has significantly lower overheads when compared with Intel BigDL. The lower overheads observed in Twister2DL can be attributed to the BSP style approach taken by Twister2DL and the highly optimized communication operations in the Twister2 framework.

Scalability is one of the most important aspects of any distributed framework. In an ideal framework, the time taken to process a job would half when the number of parallel workers is doubled for the same dataset. However, in practice, it is not possible to achieve a perfect speedup due to various overheads and inefficiencies. In order to evaluate the scalability of Twister2DL another set of experiments were performed. In this experiment, an autoencoder with an input size of 2048 and 11

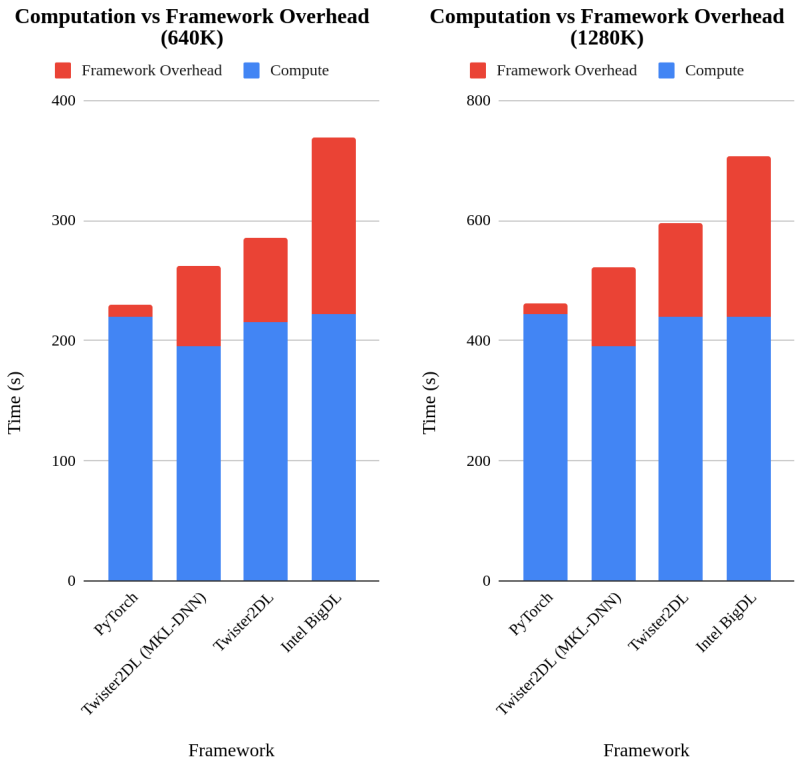


Figure 5.7: Execution time breakdown for 9 layer AE training with different frameworks with 640K,1280K data points on 40 parallel workers

layers (including the input and output layers) was trained with a dataset of 2.3 million synthetic data points. Parallelism was increased from 36 workers to 288 workers, doubling each time. For each run the total batch size was kept at 288K, which meant the mini-batch size was 64K when running with 36 workers and 8K when running with 288 workers. Keeping the total batch size constant allows the number of iterations to be constant throughout all the runs. The training was done for 10 epochs. The results of the evaluation are shown in figure 5.8. Each point in the graph was calculated from an average of 3 identical runs. The results show that the training time decreases with the number of workers. From 36 to 72 workers, the training time decreases by 1.75x, which is good. However, the decrease from 144 to 288 workers is only around 1.2x; this is because the amount of work allocated for

each worker per iteration is small; hence the reduction in computation time is offset by the increase in other overheads in the system to some extent.

Figure 5.8: Training time for 11 layer AE with increasing parallelism, 2.3 million data points and 10 epochs

5.5.3 Convolutional Neural Network

Convolutional Neural Networks or commonly known as CNN's is a widely used neural network model in the deep learning domain. CNN's are most well known for image related deep learning applications such as image recognition applications. They are good at learning patterns in grid-like data like images or time series. At their most basic form, CNN's are constructed using convolution layers and max-pooling layers. Similar to weights in the MLP models CNN's learn a set of filters and weights that embed patterns present in the training image data.

MNIST [LC10] is a well-known image data set that contains a set of handwritten digit images. The dataset has 70K images where each image is a $28x28$ pixel image.

This dataset is widely used to evaluate image classification networks by training the network with 60K images and evaluating the accuracy based on the predictions for the other 10K images. Figure 5.9 shows a network model that is used to perform digit recognition on the MNIST dataset. Since the aim of the evaluations for Twister2DL is to access its performance, the model has been chosen just for that and not based on the result accuracy of the model, as there could be better models that produce higher accuracy numbers. The code that needs to be written to implement this network in Twister2DL is listed in Algorithm 5. The users simply need to define the network, and the framework will internally do all the additional work transparent to the user.

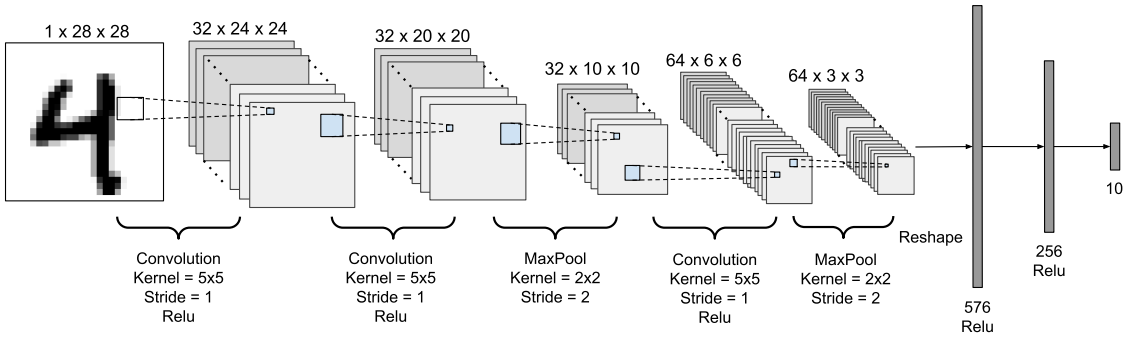


Figure 5.9: Convolutional Neural Network (CNN) for MNIST handwritten digit image data classification

In order to evaluate the performance of Twister2DL the same model was implemented in both Intel BigDL and PyTorch. Again PyTorch is only used as a reference since it is not functionally the same as Twiter2DL and BigDL. The performance evaluation is mainly focused on how well Twister2DL performs against BigDL. In order to be able to change the number of data points during evaluation,

synthetic data is used instead of the actual MNIST data set. Since the accuracy of the trained model is not important in this context the use of synthetic data is justified. For the first set of experiments, the model is trained on 12 nodes with each node running 4 workers using the distributed data parallel approach. Training performed for 100 epochs, the number of images in the dataset is increased step-wise from 60K to 240K to observe how the training time increases with increased data load. The mini-batch size is set to 256 for each run. The results of the evaluation are shown in Figure 5.10. Each data point represented in the graph is obtained from an average of 3 identical runs.

From the results, it can be observed that Twister2DL performs significantly better than Intel BigDL. For 60K images, BigDL took roughly 56% more time for training. As expected PyTorch performed better than both dataflow frameworks. However, Twister2DL is able to perform competitively. This, considering the additional overheads in the framework that are needed to provide the ease of use and other distributed data processing capabilities it provides, is more than satisfactory. For example, for 120K images, Twister2DL only takes 25.8% more training time while BigDL takes 84.5% more time to process the images.

In order to further understand the training time and which operations make up the total training time figure 5.11 shows the breakdown of the training time separating the computation time from the framework overheads. From the breakdown, it is clear that Twister2DL adds minimal overhead to the training time when compared to Intel BigDL. And as expected PyTorch has almost no overheads since the only operation that happens in PyTorch other than the computation is the communication operation. Twister2DL also shows a slightly better (roughly 15-20%) computation time when compared to BigDL. This can be attributed to better memory and data structure re-use in the Twister2DL framework. PyTorch edges slightly better com-

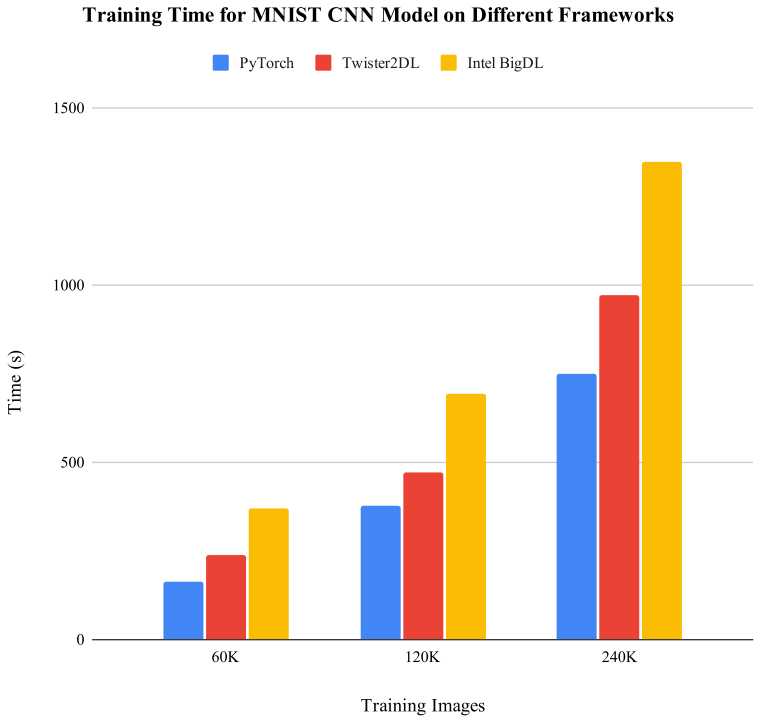


Figure 5.10: Training time for varying data sizes with different frameworks for MNIST CNN model. 100 epochs, mini-batch size 256, parallelism 48

pared to both, which is expected since the computations kernels in PyTorch are implemented in optimized c++ code.

Similar to the previous application, training of MNIST CNN model was evaluated under scaling. The MNIST CNN model shown in 5.9 is trained for 50 epochs with 250K (245760) synthetic images. The parallelism is increased from 12 parallel workers to 192 parallel workers doubling the number of workers each time. The mini-batch size is decreased from 4096 for parallelism of 12 to 256 for the parallelism of 192; this is to keep the effective batch size equal for each run. with 50 epochs, this results in 250 iterations in each run, parameters are synchronized at the end of each iteration. Figure 5.12 shows the training times for Twister2DL, Intel BigDL and PyTorch for this evaluation. The graph also shows the corresponding

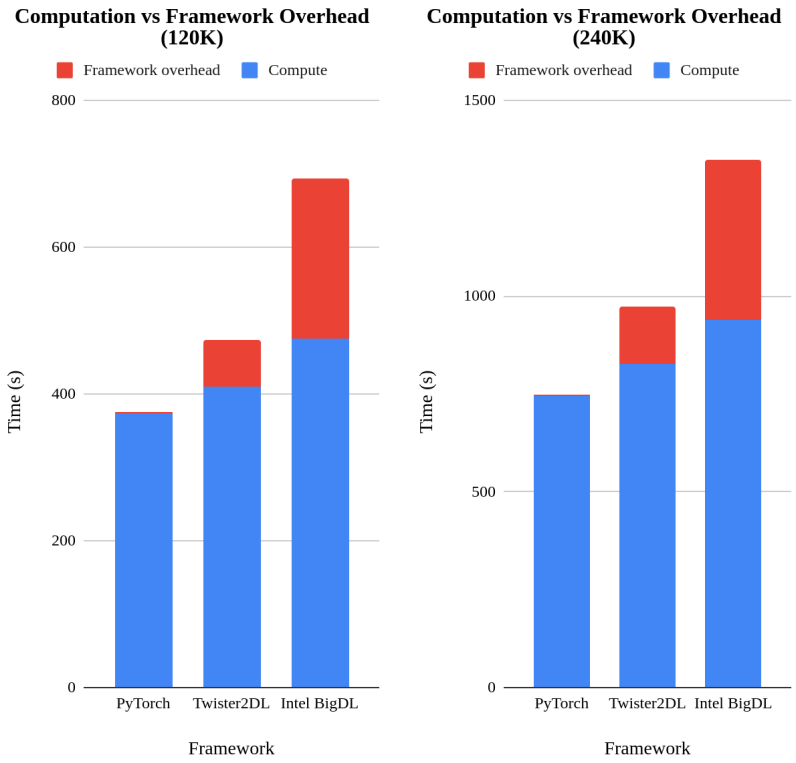


Figure 5.11: Execution time breakdown for MNIST CNN model training with different frameworks with 120K,240K image data points on 48 parallel workers

compute times as a dotted line. The results show that throughout all parallelisms, Twister2DL outperforms Intel BigDL. PyTorch performs significantly better than Twister2DL for lower parallelisms; for higher parallelisms, the performance difference becomes negligible. Another important observation that can be made is the gap between computation time and total training time in each framework. For both Twister2DL and PyTorch the difference is small; this is because the overheads added by the system are minimized. Achieving small overheads in PyTorch is straightforward since it does not provide any dataflow abstraction; however achieving minimal overhead in Twister2DL is much more challenging because of the dataflow model it supports. The design that is rooted in HPC principles has allowed Twister2DL to achieve these minimal overhead numbers. This is made more evident when looking

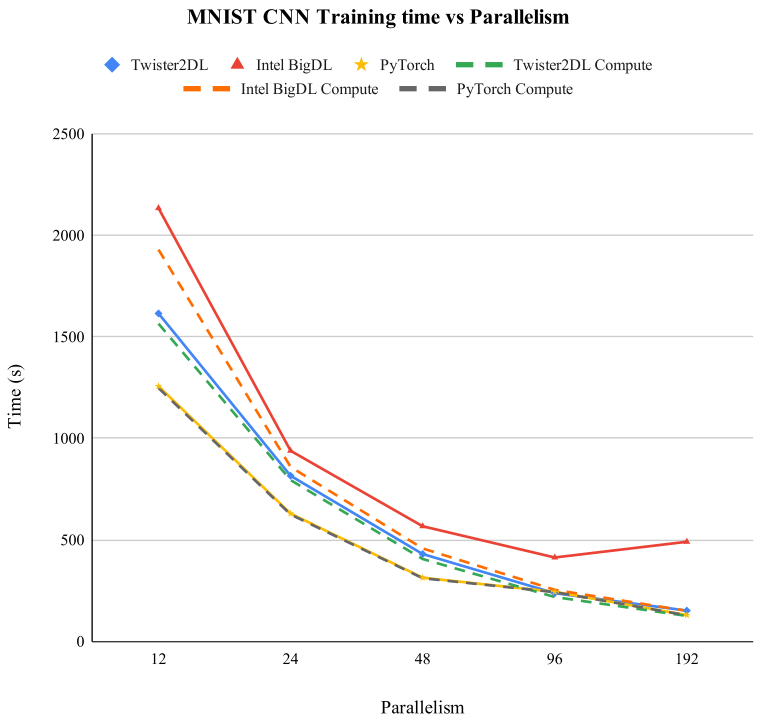


Figure 5.12: Training time for MNIST CNN with increasing parallelism, 250K images and 50 epochs

at the difference in compute and total training time in BigDL framework which also has the burden of supporting a dataflow model. The overhead added by the framework is evident in the time difference between compute and total training time. As parallelism is increased, training time lost to overheads also increase. Going from 96 workers to 192 workers, the compute time decreases as expected, but the total running time increases for BigDL; this is because the framework overhead increase is larger than the advantage gained by decreasing compute time. The scale experiments solidify the ability of Twister2DL to scale into a large number of compute nodes with minimum synchronization overheads.

Another way to look at how well an application scales is to look at the speedup it gains with the increase of parallelism. In an ideal application, the execution time

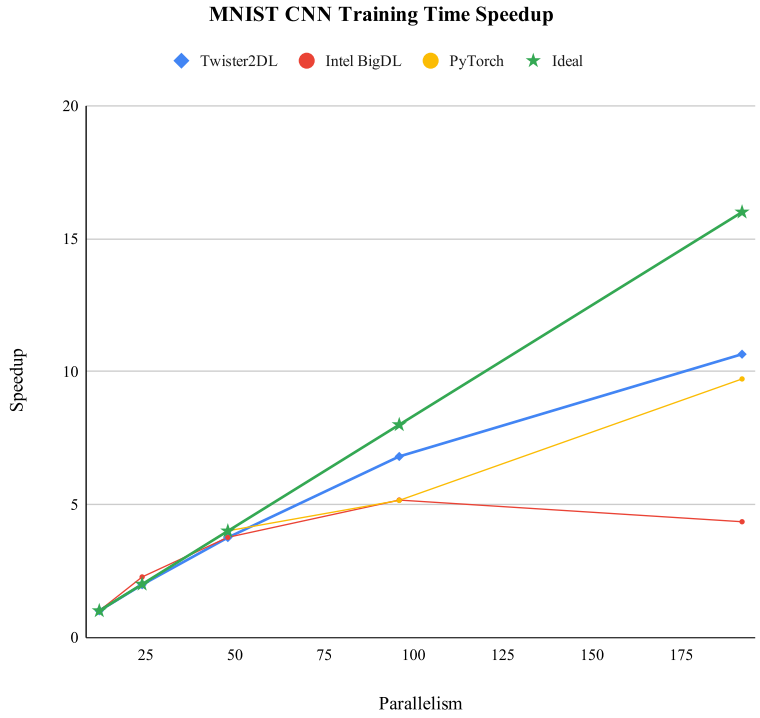


Figure 5.13: Speedup achieved by frameworks compared to Ideal speedup for MNIST CNN training

would half each time the parallelism is doubled. Therefore the speedup going from 12 workers to 24 workers would be 2. Figure 5.13 shows the speedup of all three frameworks obtain compared to the ideal speedup. While both Twister2DL and PyTorch perform well, reaching good speedup (10.6x and 9.7x respectively at 16x) BigDL is not able to achieve a similar speedup (only 4.3x at 16x). This further shows the ability of TwisterDL to scale well when compared to BigDL.

Framework	**Smith-Waterman Calculations**	**Autoencoder Training**
Twister2DL	773	36
PyTorch	8476	26

Table 5.2: Execution times for MDS with autoencoder in seconds

5.5.4 MDS with Autoencoder

Finally, we loosely evaluate the autoencoder based MDS application that was introduced in section1.2 as one of the motivational applications for Twister2DL. While this is not the optimal example to illustrate the end to end use of Twister2 TSet's and Twister2DL, it does work as a good reference application that shows how data pre-processing can a major portion of the end to end data processing pipeline. As described in section1.2 the autoencoder based MDS application has a heavy pre-processing step that is needed to calculate Smith-Waterman distances.

The evaluation was done on 8 compute nodes, with each running 2 workers. For evaluations we slightly change the number of gene sequences that are used for training from 170K to 144K, so they can be evenly distributed among parallel workers. Training was done for 100 epochs with a mini-batch size of 1000 data points. The autoencoder is kept similar to the one presented in [WF] which is $InputSize x 128 x 3 x 128 x InputSize$. Leveraging the distributed data processing capabilities in Twister2DL, the Smith-Waterman calculations and the autoencoder training are performed in a distributed manner utilizing all 16 workers. For the PyTorch implementation, the Smith-Waterman calculations were performed as a python script running on a single node. Autoencoder training was performed using PyTorch DDP using all 16 workers. It is important to note that one could perform the Smith-Waterman calculations in a parallel manner by using Twister2 TSet or some other big data framework by generating an intermediate data file and then loading that intermediate data file in PyTorch. However, this use of two frameworks is the main issue that Twister2DL aim's to address; therefore use a python script for data pre-processing which would be the approach many data scientists with little experience in distributed applications take. Table 5.2 show times taken by each framework to execute the two main components of the application. This is further

illustrated in Figure 5.14. From the results, it is clear that even though PyTorch trains the autoencoder faster than Twister2DL, it takes almost 11X more time for the Smith-Waterman calculation. This is because Twiter2DL is able to perform the pre-processing in the same distributed application. The Smith-Waterman calculation could be easily parallelized because it is a pleasingly parallel application, but the power Twister2DL is that you can write much more complex data pre-processing steps that might require distributed machine learning capabilities easily with Twister2 TSet's.

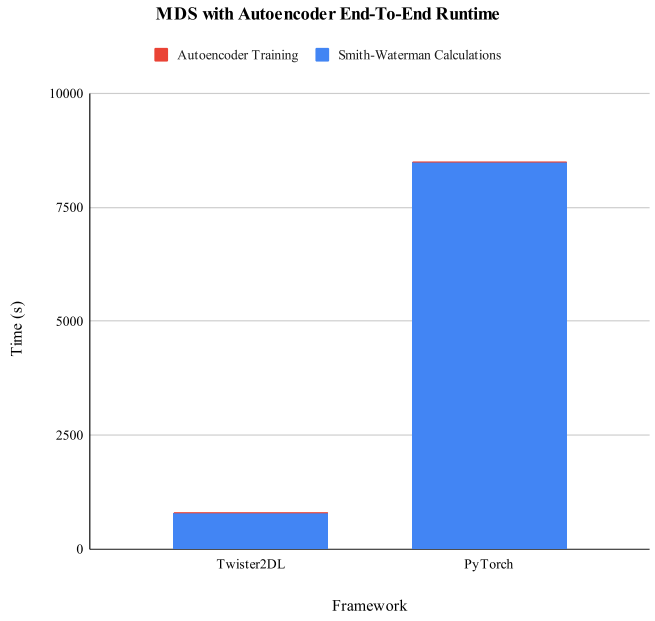


Figure 5.14: Total execution time for MDS with autoencoder, the time includes pre-processing time for Smith-Waterman calculations and the training time for the autoencoder, 100 epchocs

Algorithm 5: Twister2DL data parallel training for CNN model

```
1  SourceTSet<MiniBatch> trainSource =
     DataSetFactory.createImageMiniBatchDataSet(env, trainData, 1, 28, 28,
     miniBatchSize, dataSize, parallelism, false);
2  SourceTSet<MiniBatch> testSource =
     DataSetFactory.createImageMiniBatchDataSet(env, testData, 1, 28, 28,
     miniBatchSize, testDataSize, parallelism, false);
3  int featureSize = 3 * 3 * 64;

4  Sequential model = new Sequential();
5  model.add(convolutionMN(1, 32, 5, 5));
6  model.add(new ReLU());
7  model.add(convolutionMN(32, 32, 5, 5));
8  model.add(new SpatialMaxPooling(2, 2, 2, 2));
9  model.add(new ReLU());
10 model.add(new Dropout(0.5, false));
11 model.add(convolutionMN(32, 64, 5, 5));
12 model.add(new SpatialMaxPooling(2, 2, 2, 2));
13 model.add(new ReLU());
14 model.add(new Dropout(0.5, false));
15 model.add(new Reshape(new int[]featureSize));
16 model.add(new Linear(featureSize, 256));
17 model.add(new ReLU());
18 model.add(new Dropout(0.5, false));
19 model.add(new Linear(256, 10));
20 model.add(new LogSoftMax());

21 AbstractCriterion criterion = new CrossEntropyCriterion();

22 //Define Oprimizer;
23 Optimizer<MiniBatch> optimizer = new DistributedOptimizer(env, model,
     trainSource, criterion);
24 optimizer.setOptimMethod(new Adam());
25 optimizer.setEndWhen(Triggers.maxEpoch(epoch));
26 optimizer.optimize();
27 double accuracy = model.predictAccuracy(testSource, batchSize,
     testDataSize);
```

CHAPTER 6

CONCLUSION

We see the development of an end-to-end framework for both distributed data processing and distributed deep learning as a vital tool that is needed with the increased adaptation of deep learning. This will allow deep learning technologies to be leveraged in both existing data processing pipelines and new solutions that are built for various problems. Dataflow allows for an easy to use interface so that domain experts can easily use for machine learning and deep learning techniques in a distributed setting without having to learn extensively about parallel and distributed computing. The results obtained thus far show that the solution presented is viable and the HPC principles infused into the framework allows it to out perform rival and comparable frameworks significantly. The framework presented was able to achieve the goal of building a framework that seamlessly integrates distributed data processing and deep learning around the dataflow model. The authors are actively working on further improvements to the Twister2DL framework in terms of performance and support for a wider array of deep learning models. The potential for improvement on the TwisterDL framework is vast and can span in many different directions. With the ever changing technology landscape in the deep learning domain the authors see many aspects that can be developed to improve the framework. Quantization, Specialized hardware support (such as TPU's) and inbuilt popular deep learning models are a few such areas of promising future work.

CHAPTER 7

FUTURE WORK

Both distributed data processing and deep learning are ever evolving domains, therefore frameworks need to keep innovating and upgrading in order to keep their relevance. In our research we introduced frameworks backed by HPC principles to improve performance, however there are many areas that can be improved further. For Twister2 TSet's there is wide variety improvements that can be done to improve ease of use such as in built data connectors that can gather data from various data sources or in built machine learning algorithms similar to MLlib that is provided by Apache Spark, such work is left as future work. For Twister2DL the areas that can be further extended are countless. The most straightforward future work would be to support many more layer types so that Twister2DL can support network models such as LSTM, RNN, GAN, etc. Similarly more support for optimizer's, loss functions, etc. is also left as future work. Another major area left as future work is quantization, which would allow for a smaller memory foot print. Finally Twister2DL can be extended to support GPU computations. Intel MKL-DNN which is more recently known as OneDNN[one] recently added limited support for GPU's. OneDNN is a direct successor of Intel MKL-DNN and supports direct migration. This would be the most straigt forward path to add GPU support, however implementing based on cuda directly should provide better performance.

CHAPTER 8

RESEARCH GOALS IN ACTION

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

1. *Research on the limitations and shortcomings in current distributed dataflow frameworks* Our initial research into big data frameworks was motivated by [EKWF16] where we were looking into the performance of java thread and process based parallel machine learning algorithms. This work lead us to explore the inner working of distributed big data processing frameworks and to compare and contrast machine learning algorithms that are implemented using HPC frameworks such as OpenMPI versus big data frameworks such as Apache Spark and Flink [KWEF18]. The findings that showed the significant lack of performance in existing big data frameworks when compared to OpenMPI paved the path to further research in this area. This research work culminated into the development of Twister2 [KGW+20] which was a distributed data processing framework that was built from scratch with HPC best practices and principles in mind. Twister2 showed excellent performance that was comparable to OpenMPI and far out performed Spark and Flink in many cases [KGW+20]. With the foundation in place with a dataflow model based highly optimized communication library, we were motivated to further explore how we could extend the framework so higher level dataflow abstractions similar to Apache Spark RDD's could be implemented while providing performance comparable to HPC frameworks.

2. *Research into the applicability of a dataflow abstraction on top of HPC principles based on the Twister2 framework* Dataflow model based abstractions are more user friendly and was widely adopted with the popularity of big data

frameworks such as Apache Hadoop, Apache Spark and Apache Flink. The data based abstraction is more intuitive and hence easier to program with when compared to HPC frameworks which are mostly based on the MPI standards. However in our initial research [KWEF18, KGW+20] showed the inefficiencies of those frameworks and how HPC principles can improve performance. We further researched into supporting fully fledged dataflow abstractions ontop of the Twister2 framework to provide users performance of HPC frameworks and the ease of use of dataflow model. The research led to the design and development of Twister2 TSet's [WKG+19]. TSet's where able to provide a powerful dataflow abstraction while significantly out performing state of the art distributed data processing frameworks such as Apache Spark in many areas.

3. *Research into HPC strategies and concepts which would help boost performance of the high level dataflow abstraction within Twister2* While developing the Twister2 TSet [WKG+19] we researched into various HPC strategies and principles that can be leveraged to improve performance of the TSet abstraction layer. This research was especially focused on how iterations are handled in traditional big data frameworks vs how they are handled in HPC systems. The findings of the research led to a worker level iteration model being implemented in Twister2 TSet's. This iteration model contributed towards improving performance of the TSet abstraction enabling it to provide performance comparable to MPI implementations such a OpenMPI.

4. *Research into integrating distributed dataflow and distributed deep learning* With the capabilities and popularity of deep learning, it has become an essential component in many data processing and analysis workflows. However in the current big data and deep learning framework echo systems as shown in

figure 1.1, there is a disconnect which forces users to perform pre-processing and deep learning in separate frameworks. In order to fully understand this disconnect and propose a viable solution we researched deeper into this area. While there were several attempts made to bridge this gap by supporting deep learning frameworks such as Tensorflow on big data frameworks like Spark, each had drawbacks which prevented them from being widely adopted. Our research into integrating distributed dataflow and distributed deep learning motivated and set the foundation to extend the Twister2 framework to nativley support distributed deep learning. The developed framework extension named Twister2DL was able to provide seamless integration between data processing and deep learning in a distributed setting.

5. *Research into the applicability of HPC based dataflow model for distributed deep learning* As with Twister2 TSet's the aim of the research was to employ best practices used in the HPC domain to build Twister2DL. In order to achieve this we looked into how current state of the art deep learning frameworks are developed and how the computations are performed internally. The aim was to extract the core components that are needed and identify how a HPC based dataflow model would be able to implement them. With the knowledge gathered through the initial research we were able to design the necessary components while adding as little overhead as possible. BSP model fits well with data parallel training that is commonly used in deep learning for distributed training, therefore we opted to leverage the BSP model support in Twister2 to build up Twister2DL.

6. *Research into performance optimizations and optimized kernel libraries applicable to deep learning frameworks on CPU's* Execution time of deep learning workloads generally are dominated by the computation operations which are

used calculate gradients through forward and backward propagation. Even in a distributed training environment the computation workload is responsible for the majority of the training time. Therefore it is essential to have highly optimized kernel operations. State of the art deep learning frameworks such as PyTorch implement these kernels in optimized c++ libraries. To support optimized kernels in the JVM for Twister2DL we researched kernel libraries which are optimized for CPU operations. While GPU's are seen as the go to solution for deep learning workloads, we focused on CPU taking several conditions into account. First, the aim of Twister2DL is to provide both data pre-processing and deep learning in a single framework. Data pre-processing workloads are generally executed on CPU clusters so initially supporting CPU's for deep learning make the transition more streamlined. Second, GPU's are extremely expensive when compared to CPU's so for smaller data processing/deep learning applications CPU's would be more financially feasible as well. We do understand the merits on supporting GPU for computations and leave that work as future work. Our research into optimized CPU kernels lead adopt Intel MKL and Intel MKL-DNN libraries which allowed Twister2DL to perform on par with PyTorch with regard to CPU computations.

7. *Implement and evaluate an end-to-end solution for dataflow based distributed data processing and distributed deep learning, based on findings of aforementioned research* All the research and literature review done culminated with the development of Twister2 TSet's and Twister2DL. There two layers provide the end-to-end solution for distributed data processing and distributed deep learning on top of Twister2. Design and implementation decisions that were grounded on solid research made it possible to achieve the high performance number that are presented for both components. Twister2 TSet's was

able produce performances number comparable to HPC frameworks such as OpenMPI and vastly out perform state of the art big data frameworks such as Apache Spark and Apache Flink. All while providing an easy to use high level dataflow programming abstraction. Similarly Twister2DL was able to support an easy to use deep learning programming API and out perform Intel BigDL for deep learning applications in a distributed setting. While it was not able to match the performance of PyTorch which is a framework built specifically for deep learning, it was only a modest 15-20% slower. This is expected since Twister2DL provides more ease of use features through its dataflow model and seamless integration with distributed data pre-processing. The combined performance and ease of use for end to end data processing pipelines make the combination of Twister2DL and Twister2 TSet's a highly desirable framework for data engineers and scientists.

BIBLIOGRAPHY

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[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. 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 265–283, 2016.

[apa] Apache beam: An advanced unified programming model. `https://beam.apache.org/`. (Accessed: Aug 28 2019).

[ARAA+16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[BBC+17] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[bea] Apache beam: An advanced unified programming model. https://beam.apache.org//. (Accessed: Dec 2 2020).

[BG05] Ingwer Borg and Patrick JF Groenen. *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

[BGO+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, January 2016.

[cafa] Caffe2. https://caffe2.ai/. (Accessed: Jan 16 2021).

[Cafb] Caffeonspark. https://github.com/yahoo/CaffeOnSpark. (Accessed: Jan 12 2021).

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[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.

[DCM+92] Christian Darken, Joseph Chang, John Moody, et al. Learning rate schedules for faster stochastic gradient search. In *Neural networks for signal processing*, volume 2. Citeseer, 1992.

[dee] Deep learning for java. https://deeplearning4j.org/. (Accessed: Jan 16 2021).

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[DWQ+19] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. Bigdl: A distributed deep learning framework for big data. In

Proceedings of the ACM Symposium on Cloud Computing, pages 50–60, 2019.

[EKWF16] Saliya Ekanayake, Supun Kamburugamuve, Pulasthi Wickramasinghe, and Geoffrey C Fox. Java thread and process performance for parallel machine learning on multicore hpc clusters. In *2016 IEEE international conference on big data (Big Data)*, pages 347–354. IEEE, 2016.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. Proceedings of the 19th ACM international symposium on high performance distributed computing, pages 810–818. ACM, 2010.

[GP17] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[HAG+13] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. Ibm streams processing language: analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[Hie] Htg. `http://gecos.gforge.inria.fr/doku/doku.php?id=tutos:advanced:htg/`. (Accessed: Feb 24 2019).

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. volume 41 of *ACM SIGOPS operating systems review*, pages 59–72. ACM, 2007.

[inta] Intel mkl. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html`. (Accessed: Oct 14 2021).

[intb] Intel mkl dnn. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html. (Accessed: Oct 14 2021).

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 239–250. ACM, 2015.

[Ket17] Nikhil Ketkar. Introduction to pytorch. Deep learning with python, pages 195–208. Springer, 2017.

[KGW+17] Supun Kamburugamuve, Kannan Govindarajan, Pulasthi Wickramasinghe, Vibhatha Abeykoon, and Geoffrey Fox. Twister2: Design of a big data toolkit. *Concurrency and Computation: Practice and Experience*, page e5189, 2017.

[KGW+20] Supun Kamburugamuve, Kannan Govindarajan, Pulasthi Wickramasinghe, Vibhatha Abeykoon, and Geoffrey Fox. Twister2: Design of a big data toolkit. *Concurrency and Computation: Practice and Experience*, 32(3):e5189, 2020.

[KPJY16] Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon. Deepspark: A spark-based distributed deep learning framework for commodity clusters. *arXiv preprint arXiv:1602.08191*, 2016.

[Kru78] Joseph B Kruskal. *Multidimensional scaling*. Number 11. Sage, 1978.

[KWEF18] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, and Geoffrey C Fox. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. *The International Journal of High Performance Computing Applications*, 32(1):61–73, 2018.

[KWG+18] Supun Kamburugamuve, Pulasthi Wickramasinghe, Kannan Govindarajan, Ahmet Uyar, Gurhan Gunduz, Vibhatha Abeykoon, and Geoffrey Fox. Twister: Net-communication library for big data processing in hpc and cloud environments. 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 383–391. IEEE, 2018.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[LGM+18] Shen Li, Paul Gerver, John MacMillan, Daniel Debrunner, William Marshall, and Kun-Lung Wu. Challenges and experiences in building an efficient apache beam runner for ibm streams. *Proceedings of the VLDB Endowment*, 11(12):1742–1754, 2018.

[LP95] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[MDAT17] Claudia Misale, Maurizio Drocco, Marco Aldinucci, and Guy Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017.

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 439–455. ACM, 2013.

[nom] Nomad cluster. Accessed: Oct 18 2018.

[one] Intel® oneapi deep neural network library. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html#gs.04hn56`. (Accessed: April 22 2021).

[PC13] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and dataflow compilation of openmp streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.

[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. *arXiv preprint arXiv:1912.01703*, 2019.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[RF13] Yang Ruan and Geoffrey Fox. A robust and scalable solution for interpolative multidimensional scaling with weighting. In *2013 IEEE 9th International Conference on e-Science*, pages 61–69. IEEE, 2013.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[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.

[SKH06] Michelle Mills Strout, Barbara Kreaseck, and Paul D Hovland. Data-flow analysis for mpi programs. 2006 International Conference on Parallel Processing (ICPP'06), pages 175–184. IEEE, 2006.

[Spa] Spark coarse grained. `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-CoarseGrainedSchedulerBackend.html`. (Accessed: Feb 24 2019).

[SW+81] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[Ten] Tensorflowonspark. `https://github.com/yahoo/TensorFlowOnSpark`. (Accessed: Jan 2 2021).

[Tor] Torch. http://torch.ch/. (Accessed: Nov 21 2020).

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pages 147–156. ACM, 2014.

[twi] Twister2: Flexible, high performance data processing. https://twister2.org/. (Accessed: December 28 2019).

[WAM+13] Justin M Wozniak, Timothy G Armstrong, Ketan Maheshwari, Ewing L Lusk, Daniel S Katz, Michael Wilde, and Ian T Foster. Turbine: A distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae*, 128(3):337–366, 2013.

[WF] Pulasthi Wickramasinghe and Geoffrey Fox. Multidimensional scaling for gene sequence data with autoencoders.

[Whi12] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[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. 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.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pages 2–2. USENIX Association, 2012.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[ZRQ15] Bingjing Zhang, Yang Ruan, and Judy Qiu. Harp: Collective communication on hadoop. In *2015 IEEE International Conference on Cloud Engineering*, pages 228–233. IEEE, 2015.

[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.

Pulasthi Supun Wickramasinghe

EDUCATION

2013	B.Sc., First Class Hons, Computer Science and Engineering University of Moratuwa Sri Lanka
2021	M.Sc., Computer Science Indiana University Bloomington, IN, USA

INTERNSHIPS

2020	Research Intern Microsoft Redmond, Washington, United States

PUBLICATIONS

(Best Paper Award) 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 (HPBDIS), pp. 55-60. IEEE, 2019.

Pulasthi Wickramasinghe, Niranda Perera, Supun Kamburugamuve, Kannan Govindarajan, Vibhatha Abeykoon, Chathura Widanage, Ahmet Uyar, Gurhan Gunduz, Selahattin Akkas, and Geoffrey Fox. "High-performance iterative dataflow abstractions in Twister2: TSet." Concurrency and Computation: Practice and Experience (2020): e5998.

(Best Paper Award) Pulasthi Wickramasinghe, and Geoffrey Fox. "Multidimensional Scaling for Gene Sequence Data with Autoencoders." In 2nd International Conference on Computing and Data Science (CONF-CDS), 2021.

Pulasthi Wickramasinghe., L. D. A. Madusanka, H. P. M. Tissera, D. C. S. Weerasinghe, Shahani Markus Weerawarana, and Afkham Azeez. "An Event Driven Model for Highly Scalable Clustering for Both on Premise and Cloud Based Systems." In International Conference on Internet of Vehicles, pp. 325-336. Springer, Cham, 2014.

Supun Kamburugamuve, **Pulasthi Wickramasinghe**, Kannan Govindarajan, Ahmet Uyar, Gurhan Gunduz, Vibhatha Abeykoon, and Geoffrey Fox. "Twister: Net-communication library for big data processing in hpc and cloud environments." In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 383-391. IEEE, 2018.

Kamburugamuve, Supun, **Pulasthi Wickramasinghe**, Saliya Ekanayake, and Geoffrey C. Fox. "Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink." The International Journal of High Performance Computing Applications 32, no. 1 (2018): 61-73.

Kamburugamuve, Supun, **Pulasthi Wickramasinghe**, Saliya Ekanayake, Chathuri Wimalasena, Milinda Pathirage, and Geoffrey Fox. "TSmap3D: Browser visualization of high dimensional time series data." In 2016 IEEE International Conference on Big Data (Big Data), pp. 3583-3592. IEEE, 2016.

Kamburugamuve, Supun, Kannan Govindarajan, **Pulasthi Wickramasinghe**, Vibhatha Abeykoon, and Geoffrey Fox. "Twister2: Design of a big data toolkit." Concurrency and Computation: Practice and Experience 32, no. 3 (2020): e5189.

Ekanayake, Saliya, Supun Kamburugamuve, **Pulasthi Wickramasinghe**, and Geoffrey C. Fox. "Java thread and process performance for parallel machine learning on multicore hpc clusters." In 2016 IEEE international conference on big data (Big Data), pp. 347-354. IEEE, 2016.

Govindarajan, Kannan, Supun Kamburugamuve, **Pulasthi Wickramasinghe**, Vibhatha Abeykoon, and Geoffrey Fox. "Task scheduling in big data-review, research challenges, and prospects." In 2017 Ninth International Conference on Advanced Computing (ICoAC), pp. 165-173. IEEE, 2017.

Abeykoon, Vibhatha, Supun Kamburugamuve, Kannan Govindrarajan, **Pulasthi Wickramasinghe**, Chathura Widanage, Niranda Perera, Ahmet Uyar, Gurhan Gunduz, Selahattin Akkas, and Gregor Von Laszewski. "Streaming Machine Learning Algorithms with Big Data Systems." In 2019 IEEE International Conference on Big Data (Big Data), pp. 5661-5666. IEEE, 2019.

Widanage, Chathura, 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." In 2020 IEEE International Conference on Smart Data Services (SMDS), pp. 122-132. IEEE, 2020.

Perera, Niranda, 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).

Abeykoon, Vibhatha, 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), pp. 13-21. IEEE, 2020.

Kamburugamuve, Supun, Saliya Ekanayake, **Pulasthi Wickramasinghe**, Milinda Pathirage, and Geoffrey Fox. "Dimension Reduction and Visualization of the Structure of Financial Systems." (2015).

Kamburugamuve, Supun, **Pulasthi Wickramasinghe**, Saliya Ekanayake, Chathuri Wimalasena, and Geoffrey Fox. "WebPlotViz: Browser Visualization of High Dimensional Streaming Data with HTML5."

Uyar, Ahmet, Gurhan Gunduz, Supun Kamburugamuve, **Pulasthi Wickramasinghe**, Chathura Widanage, Kannan Govindarajan, Niranda Perera, Vibhatha Abeykoon, Selahattin Akkas, and Geoffrey Fox. "Twister2 Cross-Platform Resource Scheduler for Big Data."

Abeykoon, Vibhatha, Geoffrey Fox, Minje Kim, Saliya Ekanayake, Supun Kamburugamuve, Kannan Govindarajan, **Pulasthi Wickramasinghe** et al. "Stochastic Gradient Descent Based Support Vector Machines Training Optimization on Big Data and HPC Frameworks."

TECHNICAL REPORTS

Pulasthi Wickramasinghe, and Geoffrey Fox. "Similarities and Differences Between Parallel Systems and Distributed Systems."

Pulasthi Wickramasinghe, Judy Qiu, Minje Kim and Geoffrey Fox "Survey on Task Execution Frameworks and Designing Task Execution Model for Twister2."

Pulasthi Wickramasinghe, Judy Qiu, Minje Kim and Geoffrey Fox "Applicability of serverless computing for Machine Learning algorithms."

Pulasthi Wickramasinghe, Judy Qiu, Minje Kim and Geoffrey Fox "Survey on Serverless Computing Frameworks and Internals."

OPEN SOURCE SOFTWARE DEVELOPMENT

1. WebPlotViz: Lead developer. [*https://github.com/DSC-SPIDAL/WebPViz*]
2. Twister2: A Lead developer and researcher. [*https://twister2.org/*]
3. Apache Beam: Contributor. [*https://beam.apache.org/*]

CHAPTER 9

APPENDIX I : TWISTER2 BEAM RUNNER

This appendix show some of the implementation details of the Twister2 Beam runner. Below is a list of all the major classes that have been implemented in order to construct the Twister2 Beam runner.

- BeamBatchTSetEnvironment.java
- BeamBatchWorker.java
- Twister2BatchTranslationContext.java
- Twister2PipelineExecutionEnvironment.java
- Twister2PipelineOptions.java
- Twister2PipelineResult.java
- Twister2Runner.java
- Twister2RunnerRegistrar.java
- Twister2StreamTranslationContext.java
- Twister2TestRunner.java
- Twister2TranslationContext.java
- Twister2BoundedSource.java
- Twister2EmptySource.java
- BatchTransformTranslator.java
- StreamTransformTranslator.java
- Twister2BatchPipelineTranslator.java
- Twister2PipelineTranslator.java

- Twister2StreamPipelineTranslator.java
- AssignWindowTranslatorBatch.java
- FlattenTranslatorBatch.java
- GroupByKeyTranslatorBatch.java
- PCollectionViewTranslatorBatch.java
- ParDoMultiOutputTranslatorBatch.java
- ReadSourceTranslatorBatch.java
- AssignWindowsFunction.java
- ByteToWindowFunction.java
- ByteToWindowFunctionPrimitive.java
- DoFnFunction.java
- GroupByWindowFunction.java
- MapToTupleFunction.java
- OutputTagFilter.java
- Twister2SinkFunction.java
- SystemReduceFnBuffering.java
- ReadSourceTranslatorStream.java
- NoOpStepContext.java
- TranslationUtils.java
- Twister2AssignContext.java
- Twister2SideInputReader.java

The main control logic is defined in the "Twister2Runner" class and what each parallel worker performs is encapsulated in the "BeamBatchWorker" class. Figure 9.1 shows the code for the "BeamBatchWorker" class and how it controls the dataflow.

```
public void execute(BatchTSetEnvironment env) {
  Config config = env.getConfig();
  Map<String, String> sideInputIds = (LinkedHashMap<String, String>) config.get(SIDEINPUTS);
  Set<String> leaveIds = (Set<String>) config.get(LEAVES);
  TBaseGraph graph = (TBaseGraph) config.get(GRAPH);
  env.settBaseGraph(graph);
  setupTSets(env, sideInputIds, leaveIds);
  resetEnv(env, graph);
  executePipeline(env);
}

/** resets the proper TSetEnvironment in all the TSets in the graph that was sent over the wire. ...*/
private void resetEnv(BatchTSetEnvironment env, TBaseGraph graph) {
  Set<TBase> nodes = graph.getNodes();
  for (TBase node : nodes) {
    if (node instanceof BaseTSet) {
      ((BaseTSet) node).setTSetEnv(env);
    } else if (node instanceof BaseTLink) {
      ((BaseTLink) node).setTSetEnv(env);
    } else {
      throw new IllegalStateException("node must be either of type BaseTSet or BaseTLink");
    }
  }
}

/** Extract the sideInput TSets and the Leaves from the graph. ...*/
private void setupTSets(
    BatchTSetEnvironment env, Map<String, String> sideInputIds, Set<String> leaveIds) {
  sideInputDataSets = new LinkedHashMap<>();
  leaves = new HashSet<>();

  // reset sources, so that the graph does not have two source objects
  // created during deserialization
  Set<BuildableTSet> newSources = new HashSet<>();
  for (BuildableTSet source : env.getGraph().getSources()) {
    newSources.add((BuildableTSet) env.getGraph().getNodeById(source.getId()));
  }
  env.getGraph().setSources(newSources);

  for (Map.Entry<String, String> entry : sideInputIds.entrySet()) {
    BatchTSet curr = (BatchTSet) env.getGraph().getNodeById(entry.getValue());
    sideInputDataSets.put(entry.getKey(), curr);
  }
  for (String leaveId : leaveIds) {
    leaves.add((TSet) env.getGraph().getNodeById(leaveId));
  }
}

public void executePipeline(BatchTSetEnvironment env) {
  Map<String, CachedTSet> sideInputTSets = new HashMap<>();
  for (Map.Entry<String, BatchTSet<?>> sides : sideInputDataSets.entrySet()) {
    BatchTSet<?> sideTSet = sides.getValue();
    addInputs((BaseTSet) sideTSet, sideInputTSets);
    CachedTSet tempCache = (CachedTSet) sideTSet.cache();
    sideInputTSets.put(sides.getKey(), tempCache);
  }
  for (TSet leaf : leaves) {
    SinkTSet sinkTSet = (SinkTSet) leaf.direct().sink(new Twister2SinkFunction());
    addInputs(sinkTSet, sideInputTSets);
    eval(env, sinkTSet);
  }
}

/** Adds all the side inputs into the sink test so it is available from the DoFn's. */
private void addInputs(BaseTSet sinkTSet, Map<String, CachedTSet> sideInputTSets) {
  if (sideInputTSets.isEmpty()) {
    return;
  }

  TBaseGraph graph = sinkTSet.getTBaseGraph();
  TBase currNode = null;
  Deque<TBase> deque = new ArrayDeque<>();
  deque.add(sinkTSet);
  while (!deque.isEmpty()) {
    currNode = deque.remove();
    deque.addAll(graph.getPredecessors(currNode));
    if (currNode instanceof ComputeTSet) {
      if (((ComputeTSet) currNode).getComputeFunc() instanceof DoFnFunction) {
        Set<String> sideInputKeys =
            ((DoFnFunction) ((ComputeTSet) currNode).getComputeFunc()).getSideInputKeys();
        for (String sideInputKey : sideInputKeys) {
          if (!sideInputTSets.containsKey(sideInputKey)) {
            throw new IllegalStateException("Side input not found for key " + sideInputKey);
          }
          ((ComputeTSet) currNode).addInput(sideInputKey, sideInputTSets.get(sideInputKey));
        }
      }
    }
  }
}
```

Figure 9.1: BeamBatchWorker code

The bulk of the work in the Twister2 runner are performed by "Translators" the runner implements many translators as seen in the list above. The translator is responsible of converting segments of the beam pipeline code into corresponding Twister2 TSet code. Figure 9.2 shows the code for the "ParDoMultiOutputTranslatorBatch" class. This class converts the "ParDo" primitive in Beam into Twister2 TSet's.

```
@Override
public void translateNode(
    ParDo.MultiOutput<InputT, OutputT> transform, Twister2BatchTranslationContext context) {
  DoFn<InputT, OutputT> doFn;
  doFn = transform.getFn();
  if (DoFnSignatures.signatureForDoFn(doFn).processElement().isSplittable()) {
    throw new UnsupportedOperationException(
        String.format(
            "Not expected to directly translate splittable DoFn, should have been overridden: %s",
            doFn));
  }
  BatchTSetImpl<WindowedValue<InputT>> inputTTSet =
      context.getInputDataSet(context.getInput(transform));

  WindowingStrategy<?, ?> windowingStrategy = context.getInput(transform).getWindowingStrategy();
  Coder<InputT> inputCoder = (Coder<InputT>) context.getInput(transform).getCoder();
  Map<String, PCollectionView<?>> sideInputMapping;

  Map<TupleTag<?>, PValue> outputs = context.getOutputs();
  Map<TupleTag<?>, Coder<?>> outputCoders = context.getOutputCoders();

  // DoFnSignature signature = DoFnSignatures.getSignature(transform.getFn().getClass());
  DoFnSchemaInformation doFnSchemaInformation;
  doFnSchemaInformation = ParDoTranslation.getSchemaInformation(context.getCurrentTransform());
  sideInputMapping = ParDoTranslation.getSideInputMapping(context.getCurrentTransform());

  TupleTag<OutputT> mainOutput = transform.getMainOutputTag();
  List<TupleTag<?>> additionalOutputTags =
      new ArrayList<>(transform.getAdditionalOutputTags().getAll());
  Map<String, PCollectionView<?>> sideInputs = transform.getSideInputs();
  // TODO : note change from List to map in sideinputs

  // construct a map from side input to WindowingStrategy so that
  // the DoFn runner can map main-input windows to side input windows
  Map<PCollectionView<?>, WindowingStrategy<?, ?>> sideInputStrategies = new HashMap<>();
  for (PCollectionView<?> sideInput : sideInputs.values()) {
    sideInputStrategies.put(sideInput, sideInput.getWindowingStrategyInternal());
  }

  TupleTag<?> mainOutputTag;
  try {
    mainOutputTag = ParDoTranslation.getMainOutputTag(context.getCurrentTransform());
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
  Map<TupleTag<?>, Integer> outputMap = Maps.newHashMap();
  outputMap.put(mainOutputTag, 0);
  int count = 1;
  for (TupleTag<?> tag : outputs.keySet()) {
    if (!outputMap.containsKey(tag)) {
      outputMap.put(tag, count++);
    }
  }

  ComputeTSet<RawUnionValue, Iterator<WindowedValue<InputT>>> outputTSet =
      inputTTSet
          .direct()
          .<RawUnionValue>compute(
              new DoFnFunction<OutputT, InputT>(
                  context,
                  doFn,
                  inputCoder,
                  outputCoders,
                  additionalOutputTags,
                  windowingStrategy,
                  sideInputStrategies,
                  mainOutput,
                  doFnSchemaInformation,
                  outputMap,
                  sideInputMapping));

  for (Map.Entry<TupleTag<?>, PValue> output : outputs.entrySet()) {
    ComputeTSet<WindowedValue<OutputT>, Iterator<RawUnionValue>> tempTSet =
        outputTSet.direct().compute(new OutputTagFilter(outputMap.get(output.getKey())));
    context.setOutputDataSet((PCollection) output.getValue(), tempTSet);
  }
}
```

Figure 9.2: ParDoMultiOutputTranslatorBatch code

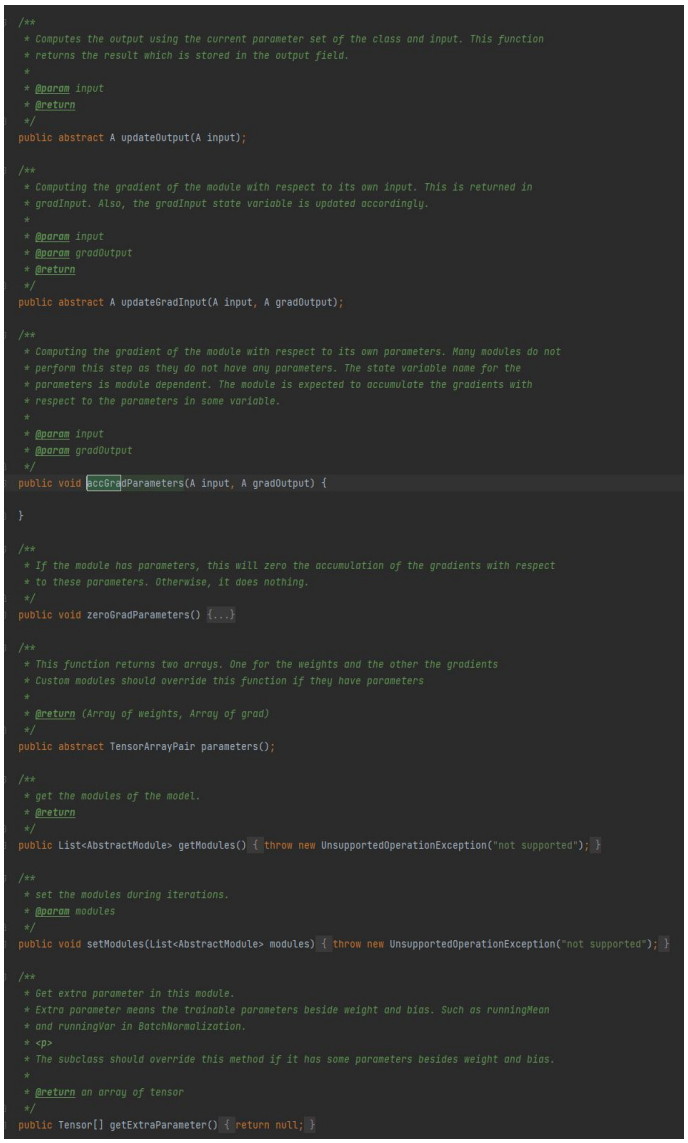
CHAPTER 10

APPENDIX II : TWISTER2DL IMPLEMENTATION

This appendix documents some implementation details of the Twister2DL implementation. The complete Twister2DL codebase is about 32,000 lines of code, therefore the codebase cannot be shown in its entirety. Below we list several key segments of the code that showcases more complex segments of the implementation to highlight the implementation complexity of Twister2DL.

First to look at how extensibility is supported the code segment shown in Figure 10.1 show the main methods that each deep learning layer needs to implement. And Figure 10.2 show some segments of the code for "LogSoftMax" layer which implement those functions.

Next, In order to provide glance at the complexities that are involved performing computation optimizations Figure 10.3 shows segment of the code from the "ReOrderMemory" class which helps with the memory mappings needed for MKL-DNN. And to further illustrage the point Figure 10.4 shows code segments from the additional computation codes needed for the "Linear" layer to support MKL-DNN.



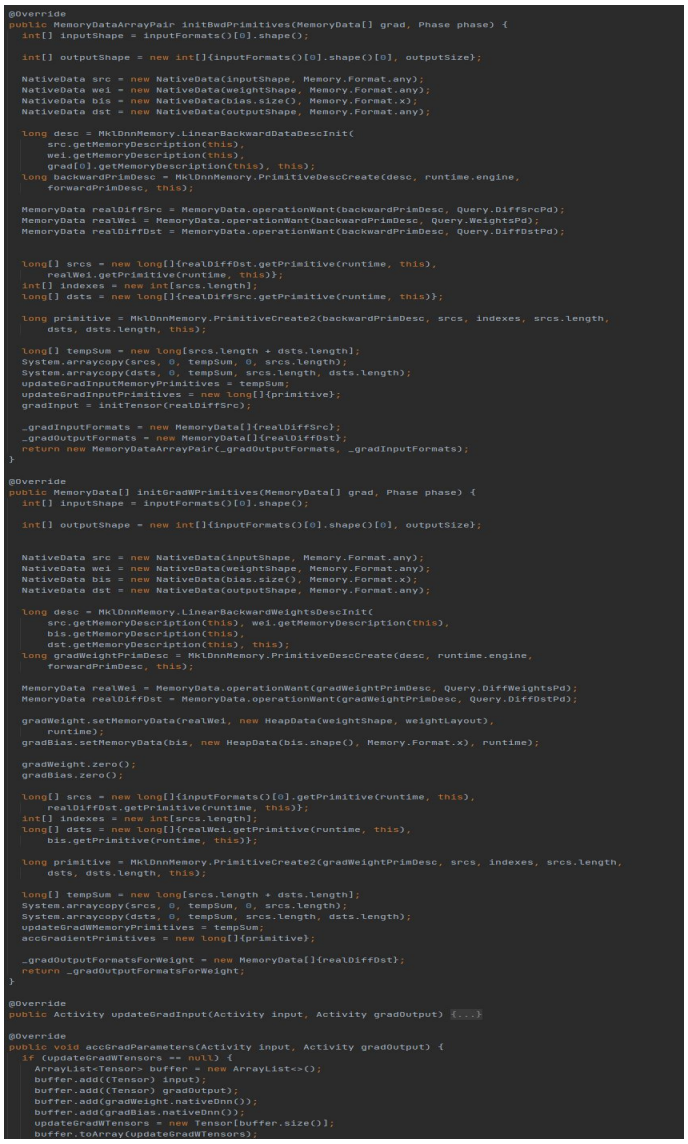
```java
/**
 * Computes the output using the current parameter set of the class and input. This function
 * returns the result which is stored in the output field.
 *
 * @param input
 * @return
 */
public abstract A updateOutput(A input);

/**
 * Computing the gradient of the module with respect to its own input. This is returned in
 * gradInput. Also, the gradInput state variable is updated accordingly.
 *
 * @param input
 * @param gradOutput
 * @return
 */
public abstract A updateGradInput(A input, A gradOutput);

/**
 * Computing the gradient of the module with respect to its own parameters. Many modules do not
 * perform this step as they do not have any parameters. The state variable name for the
 * parameters is module dependent. The module is expected to accumulate the gradients with
 * respect to the parameters in some variable.
 *
 * @param input
 * @param gradOutput
 */
public void accGradParameters(A input, A gradOutput) {

}

/**
 * If the module has parameters, this will zero the accumulation of the gradients with respect
 * to these parameters. Otherwise, it does nothing.
 */
public void zeroGradParameters() {...}

/**
 * This function returns two arrays. One for the weights and the other the gradients
 * Custom modules should override this function if they have parameters
 *
 * @return (Array of weights, Array of grad)
 */
public abstract TensorArrayPair parameters();

/**
 * get the modules of the model.
 * @return
 */
public List<AbstractModule> getModules() { throw new UnsupportedOperationException("not supported"); }

/**
 * set the modules during iterations.
 * @param modules
 */
public void setModules(List<AbstractModule> modules) { throw new UnsupportedOperationException("not supported"); }

/**
 * Get extra parameter in this module.
 * Extra parameter means the trainable parameters beside weight and bias. Such as runningMean
 * and runningVar in BatchNormalization.
 * <p>
 * The subclass should override this method if it has some parameters besides weight and bias.
 *
 * @return an array of tensor
 */
public Tensor[] getExtraParameter() { return null; }
```

Figure 10.1: Layer extension points code

Figure 10.2: LogSoftMax code

```
@Override
public MemoryDataArrayPair initFwdPrimitives(MemoryData[] inputs, Phase phase) {
  if (memoryOwner == null) {
    memoryOwner = this;
  }
  if (inputFormat == null) {
    _inputFormats = inputs;
  } else {
    _inputFormats = new MemoryData[]{inputFormat};
  }
  Util.require(_inputFormats.length == 1, "Only accept one tensor as input");

  if (outputFormat == null) {
    _outputFormats = _inputFormats;
  }

  shapeToString(_inputFormats[0].shape());

  Util.require(TensorNumeric.product(_inputFormats[0].shape())
          == TensorNumeric.product(_outputFormats[0].shape()),
      "input output memory not match, input shape " + shapeToString(_inputFormats[0].shape())
          + "output shape " + shapeToString(_outputFormats[0].shape()));

  int[] inputShape = _inputFormats[0].shape();
  int[] outputShape = _outputFormats[0].shape();
  int inputLayout = _inputFormats[0].layout();
  int outputLayout = _outputFormats[0].layout();
  realInput = _inputFormats;
  realOutput = _outputFormats;

  if (inputLayout != outputLayout) {
    if (inputLayout == Memory.Format.nhwc || inputLayout == Memory.Format.ntc) {
      // remind: if format of input MemoryData is nhwc or ntc,
      // its shape should be output shape
      realInput = initMemory(_inputFormats[0], outputShape, inputLayout);
    } else if (outputLayout == Memory.Format.nhwc || outputLayout == Memory.Format.ntc) {
      // remind: if format of output MemoryData is nhwc or ntc,
      // its shape should be input shape
      realOutput = initMemory(_outputFormats[0], inputShape, outputLayout);
    }
  }

  boolean noInt8Formats = inputFormats()[0].dataType() == DataType.F32
      && outputFormats()[0].dataType() == DataType.F32;

  long fwdReorderPrimDesc;
  if (noInt8Formats) {
    fwdReorderPrimDesc = MklDnnMemory.ReorderPrimitiveDescCreate(
        realInput[0].getPrimitiveDescription(runtime, memoryOwner),
        realOutput[0].getPrimitiveDescription(runtime, memoryOwner), memoryOwner);
  } else {
    throw new UnsupportedOperationException("Int8 not supported");
  }

  long fwdReorderPrim = MklDnnMemory.PrimitiveCreate2(fwdReorderPrimDesc,
      new long[]{realInput[0].getPrimitive(runtime, memoryOwner)}, new int[]{0}, 1,
      new long[]{realOutput[0].getPrimitive(runtime, memoryOwner)}, 1, memoryOwner);

  updateOutputPrimitives = new long[]{fwdReorderPrim};

  // recover to original data
  output = initTensor(realOutput[0]);

  reshapeOutputIfNeeded(_outputFormats[0], output.toTensor());

  return new MemoryDataArrayPair(_inputFormats, _outputFormats);
}


@Override
public MemoryDataArrayPair initBwdPrimitives(MemoryData[] grad, Phase phase) {
  if (memoryOwner == null) {
    memoryOwner = this;
  }
  if (gradInputFormat == null && inputFormat == null) {
    _gradInputFormats = inputFormats();
  } else if (gradInputFormat == null && inputFormat != null) {
    _gradInputFormats = new MemoryData[]{inputFormat};
  } else if (gradInputFormat != null) {
    _gradInputFormats = new MemoryData[]{gradInputFormat};
  }

  if (gradOutputFormat == null) {
    _gradOutputFormats = grad;
  } else {
    _gradOutputFormats = new MemoryData[]{gradOutputFormat};
  }
  Util.require(_gradOutputFormats.length == 1, "Only accept one tensor as input");
  Util.require(TensorNumeric.product(_gradOutputFormats[0].shape())
          == TensorNumeric.product(_gradInputFormats[0].shape()),
      "gradInput and gradOutput memory not match,"
          + "gradInput shape " + shapeToString(_gradInputFormats[0].shape())
          + "gradOutput shape " + shapeToString(_gradOutputFormats[0].shape()));

  int[] gradInputShape = _gradInputFormats[0].shape();
  int[] gradOutputShape = _gradOutputFormats[0].shape();
  int gradInputLayout = _gradInputFormats[0].layout();
  int gradOutputLayout = _gradOutputFormats[0].layout();
  realgradInput = _gradInputFormats;
  realgradOutput = _gradOutputFormats;

  if (gradInputLayout != gradOutputLayout) {
    if (gradOutputLayout == Memory.Format.nhwc || gradOutputLayout == Memory.Format.ntc) {
      // remind: if format of gradOutput MemoryData is nhwc or ntc,
      // its shape should be gradInput shape
      realgradOutput = initMemory(_gradOutputFormats[0], gradInputShape, gradOutputLayout);
    } else if (gradInputLayout == Memory.Format.nhwc || gradInputLayout == Memory.Format.ntc) {
```

Figure 10.3: MKL-DNN ReOrderMemory code



```
@Override
public MemoryDataArrayPair initBwdPrimitives(MemoryData[] grad, Phase phase) {
  int[] inputShape = inputFormats()[0].shape();

  int[] outputShape = new int[]{inputFormats()[0].shape()[0], outputSize};

  NativeData src = new NativeData(inputShape, Memory.Format.any);
  NativeData wei = new NativeData(weightShape, Memory.Format.any);
  NativeData bis = new NativeData(bias.size(), Memory.Format.x);
  NativeData dst = new NativeData(outputShape, Memory.Format.any);

  long desc = MklDnnMemory.LinearBackwardDataDescInit(
      src.getMemoryDescription(this),
      wei.getMemoryDescription(this),
      grad[0].getMemoryDescription(this), this);
  long backwardPrimDesc = MklDnnMemory.PrimitiveDescCreate(desc, runtime.engine,
      forwardPrimDesc, this);

  MemoryData realDiffSrc = MemoryData.operationWant(backwardPrimDesc, Query.DiffSrcPd);
  MemoryData realWei = MemoryData.operationWant(backwardPrimDesc, Query.WeightsPd);
  MemoryData realDiffDst = MemoryData.operationWant(backwardPrimDesc, Query.DiffDstPd);


  long[] srcs = new long[]{realDiffDst.getPrimitive(runtime, this),
      realWei.getPrimitive(runtime, this)};
  int[] indexes = new int[srcs.length];
  long[] dsts = new long[]{realDiffSrc.getPrimitive(runtime, this)};

  long primitive = MklDnnMemory.PrimitiveCreate2(backwardPrimDesc, srcs, indexes, srcs.length,
      dsts, dsts.length, this);

  long[] tempSum = new long[srcs.length + dsts.length];
  System.arraycopy(srcs, 0, tempSum, 0, srcs.length);
  System.arraycopy(dsts, 0, tempSum, srcs.length, dsts.length);
  updateGradInputMemoryPrimitives = tempSum;
  updateGradInputPrimitives = new long[]{primitive};
  gradInput = initTensor(realDiffSrc);

  _gradInputFormats = new MemoryData[]{realDiffSrc};
  _gradOutputFormats = new MemoryData[]{realDiffDst};
  return new MemoryDataArrayPair(_gradOutputFormats, _gradInputFormats);
}

@Override
public MemoryData[] initGradWPrimitives(MemoryData[] grad, Phase phase) {
  int[] inputShape = inputFormats()[0].shape();

  int[] outputShape = new int[]{inputFormats()[0].shape()[0], outputSize};


  NativeData src = new NativeData(inputShape, Memory.Format.any);
  NativeData wei = new NativeData(weightShape, Memory.Format.any);
  NativeData bis = new NativeData(bias.size(), Memory.Format.x);
  NativeData dst = new NativeData(outputShape, Memory.Format.any);

  long desc = MklDnnMemory.LinearBackwardWeightsDescInit(
      src.getMemoryDescription(this), wei.getMemoryDescription(this),
      bis.getMemoryDescription(this),
      dst.getMemoryDescription(this), this);
  long gradWeightPrimDesc = MklDnnMemory.PrimitiveDescCreate(desc, runtime.engine,
      forwardPrimDesc, this);

  MemoryData realWei = MemoryData.operationWant(gradWeightPrimDesc, Query.DiffWeightsPd);
  MemoryData realDiffDst = MemoryData.operationWant(gradWeightPrimDesc, Query.DiffDstPd);

  gradWeight.setMemoryData(realWei, new HeapData(weightShape, weightLayout),
      runtime);
  gradBias.setMemoryData(bis, new HeapData(bis.shape(), Memory.Format.x), runtime);

  gradWeight.zero();
  gradBias.zero();

  long[] srcs = new long[]{inputFormats()[0].getPrimitive(runtime, this),
      realDiffDst.getPrimitive(runtime, this)};
  int[] indexes = new int[srcs.length];
  long[] dsts = new long[]{realWei.getPrimitive(runtime, this),
      bis.getPrimitive(runtime, this)};

  long primitive = MklDnnMemory.PrimitiveCreate2(gradWeightPrimDesc, srcs, indexes, srcs.length,
      dsts, dsts.length, this);

  long[] tempSum = new long[srcs.length + dsts.length];
  System.arraycopy(srcs, 0, tempSum, 0, srcs.length);
  System.arraycopy(dsts, 0, tempSum, srcs.length, dsts.length);
  updateGradWMemoryPrimitives = tempSum;
  accGradientPrimitives = new long[]{primitive};

  _gradOutputFormatsForWeight = new MemoryData[]{realDiffDst};
  return _gradOutputFormatsForWeight;
}

@Override
public Activity updateGradInput(Activity input, Activity gradOutput) {...}

@Override
public void accGradParameters(Activity input, Activity gradOutput) {
  if (updateGradWTensors == null) {
    ArrayList<Tensor> buffer = new ArrayList<>();
    buffer.add((Tensor) input);
    buffer.add((Tensor) gradOutput);
    buffer.add(gradWeight.nativeDnn());
    buffer.add(gradBias.nativeDnn());
    updateGradWTensors = new Tensor[buffer.size()];
    buffer.toArray(updateGradWTensors);
```

Figure 10.4: MKL-DNN Linear code