Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

High-Performance Iterative Dataflow Abstractions in Twister2:TSet

Pulasthi Wickramasinghe[1] | Niranda Perera[1] | Supun Kamburugamuve[1] | Kannan Govindarajan[1] | Vibhatha Abeykoon[1] | Chathura Widanage[1] | Ahmet Uyar[2] | Gurhan Gunduz[2] | Selahattin Akkas[1] | Geoffrey Fox[1]

[1] School of Informatics, Computing and Engineering, Indiana University, Indiana, USA
[2] Digital Science Center, Indiana University, Indiana, USA

Correspondence
*Pulasthi Wickramasinghe. Email: pswickra@iu.edu

Present Address
2425 N Milo B Sampson Ln, Bloomington, IN, 47408, USA

Summary

The dataflow model is gradually becoming the de facto standard for big data applications. While many popular frameworks are built around this model, very little research has been done on understanding its inner workings, which in turn has led to inefficiencies in existing frameworks. It is important to note that understanding the relationship between dataflow and HPC building blocks allows us to address and alleviate many of these fundamental inefficiencies by learning from the extensive research literature in the HPC community. In this paper, we present TSet's, the dataflow abstraction of Twister2, which is a big data framework designed for high-performance dataflow and iterative computations. We discuss the dataflow model adopted by TSet's and the rationale behind implementing iteration handling at the worker level. Finally, we evaluate TSet's to show the performance of the framework and the importance of the worker level iteration model.

KEYWORDS:
dataflow, big data, mapreduce, batch, stream, iterative

1 | INTRODUCTION

In recent years, the big data domain has seen a massive increase in popularity thanks to the tremendous volume of data that needs to be processed and analyzed in order to gain valuable information. The ever-increasing number of use cases that emerge with the wide adoption of big data in both commercial and scientific communities also contributed to this popularity. This has led to the creation of a wide variety of frameworks that cater to different user requirements. Hadoop[1], Spark[2], and Flink[3] focus on batch processing; Storm[4], Heron[5], and Flink[3] target stream processing; while TensorFlow[6] and PyTorch[7] are for machine learning. In addition frameworks such as Apache Beam[8] provide an unified API for both batch and stream processing, and support many of the above mentioned frameworks as data processing engines underneath. These are just a few popular examples of such frameworks, which provide various higher-level abstractions and API's for end users to program applications while hiding the complexities of parallel programs. As the size of data increases tremendously, there is a necessity to adopt the distributed frameworks for processing and analysing those data. Although each framework has its own abstraction and implementation, it is observable that most frameworks share a common dataflow programming model. Furthermore, these systems provides the ability for the user to develop their applications using the dataflow model. Once an application is developed using the dataflow model, the runtime[9] system takes the responsibility of dynamically mapping the dataflow graph into an execution graph. This graph is then executed on a cluster as a distributed program. Moreover, the users have been exposed to exploit the implicit parallelism exposed by the dataflow graph while developing applications/programs. Another important observation is the similarities that data analytics frameworks have with High Performance Computing (HPC) frameworks at the parallel operator level. For example, most operations supported by big data frameworks can be mapped to operations that are well-established in HPC frameworks, such as gather, reduce, and partition operations, which will be discussed in more detail in Section 2. However, these similarities and even the common model are still not very well-defined in the research literature. A solid understanding of the dataflow model and how each framework has implemented it would help to build more optimized systems. In[10] the authors studied this topic in more detail, and their findings motivated the development of Twister2[11,12], 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. TSet is the dataflow abstraction of Twister2, this was initially introduced by the authors in[13]. Since then several major changes have been introduced into the TSet API with the subsequent releases of Twister2. The major changes are listed below.

- Complete revamping of the TSet user API
- New build layer to support multi-inputs and multi-outputs with TSet's
- Connected dataflow model to support workflow style programming
- Apache Beam runner for compatibility with Apache Beam
- Improved iteration handling to increase performance

In this paper we look at TSet's in more detail and discuss the latest developments, as described in section 4. In section 7 we will look at the performance improvements gained from the improved iteration handling at the TSet level

Iterations are one of the core building blocks of parallel applications. Frameworks built around the dataflow model handle iterations via different approaches. The way iterations are incorporated into a framework has a significant effect on the performance of the framework, especially for more complex algorithms with many iterative elements[10]. Furthermore, it affects usability and even the way frameworks handle fault tolerance. Complex machine learning algorithms, written using frameworks such as OpenMPI using bulk synchronous programming (BSP) model, have much better performance because of their approach to iterations and local data[10]. The handling of iterations in Twister2 is a major distinction between it and other popular frameworks such as Spark[2] and Flink[3]. This distinction also provides Twister2 with great performance advantage over other systems for complex machine learning algorithms which involve iterative computations. This will be discussed further in section 3.2 and performance numbers will be presented in section 7.

While iterations and optimized communications patterns are important for developing fine grained dataflow, coarse grained dataflow which is more commonly known as workflow, provide a different set of requirements. One major requirement is ability to compose several dependent or independent applications to create a single end to end application. Twister2 introduces a concept of "Connected Dataflow" to address such coarse grained dataflow requirements, connected dataflow is described in more detail in section 5.

In this paper, we first examine the generic dataflow model and how different capabilities of the model can be mapped to those in the HPC domain, using MPI as an example in section 2. This is done to better understand the similarities and differences between the two domains. Next, we introduce the dataflow model adopted by Twister2 for stream and batch processing and discuss the rationale behind the model decisions taken in section 3. In section 4 we describe and discuss the TSet API and its basic building blocks, its compatibility with other dataflow models and showcase its completeness. In section 5 we briefly look at the concept of connected dataflow and how the TSet API allows Twister2 to support a connected dataflow model. Finally, in section 7 we present experimental performance results of Twister2. The main contributions of the paper are summarized below:

- An overview 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

2 | COMPARISON OF DATAFLOW AND MPI

MPI is a well known messaging standard that can support various programming models. The BSP moodel is the most common model used to write MPI programs. MPI operations can be used to build APIs for other programming paradigms such as dataflow. However, most people will agree that in its pure form, MPI specification is suited for BSP-style programs as the user needs to define higher-level API's to make it easily programmable in areas such as graph processing and streaming analysis. Dataflow programming model is a widely used framework designed for streaming, data analysis, and graph processing.

Collective operations are arguably the major use case of MPI for parallel programs. Gather, reduce, allreduce, allgather and scatter are some such well known collective operations. MPI collectives are generalized versions of popular communication patterns for parallel programs. We have identified the same collectives with slightly different semantics used for big data computing. It is difficult to use MPI collectives in their pure form in data applications as their requirements are slightly different. Twister:Net[14] is an attempt to define collective semantics for data analytics jobs and provides an implementation of various operations both using OpenMPI and TCP sockets. Dataflow collectives are driven by the following requirements that make them slightly different from MPI specification-based collectives:

1. The collectives operate between a set of tasks in an arbitrary task graph.
2. Collectives handle data that may be larger than available memory
3. Dynamic data sizes for operators.
4. Keys are part of the abstraction.
5. Data may be imbalanced data and require termination detection.

The dataflow programming model and BSP model differ in many aspects. Dataflow mostly is an event-driven model where the user programs a set of event handlers arranged in a graph. The user is not exposed to important details of the parallel program such as threads, parallel operators and data handling. MPI specification-based BSP programs provide the bare minimum requirements for a parallel program, details such as thread management and data management are left for the end user. It is a challenge to preserve MPI performance while providing a higher-level of abstraction that can be used for data analytics. As such TSet's is an attempt to balance both.

3 | TWISTER2 DATAFLOW MODEL

Dataflow is the preferred choice for processing large-scale data as it hides the underlying details of the distributed processing, coordination, and data management. It also simplifies the process of parallelizing tasks and provides the ability to dynamically determine the dependency between those tasks. In the dataflow programming model, the application is designed as a dataflow graph which can be created either statically or dynamically. The nodes in the dataflow graph consist of task vertices and edges, in which task vertices represent the computational units of an application and edges represent the communication edges between those computational units. A dataflow graph is made of multiple subtasks which are arranged based on the parent-child relationship between the tasks. In other words, it describes the details about how the data is consumed between those units. Each node in the dataflow graph holds the information about the input and its output. The task could be long-running or short-running depending on the type of application. In a static dataflow graph, the structure of the complete graph is known at compile time, whereas in a dynamic dataflow graph, the structure is not known at compile time but it is dynamically defined during the run time of the application. This graph is converted into an execution graph once the actual execution takes place.

It is also important to keep both batch and streaming modes of dataflow in mind when designing the model, focusing on one or the other may result in non optimal performance for one of the modes. For example, Apache Spark[2] was developed as an batch processing engine and therefore handles streaming using an mini batch model, while Apache Flink[3] was developed as a streaming engine. The Twister2 framework has been developed around the dataflow model such that it supports both streaming and batch operations as first-class concepts. This distinction between streaming and batch need to be supported throughout all the layers of the system so that the framework performance equally in both modes of operation. For instance, for a batch application the framework can schedule tasks that need to be executed one after the other in the same location by reusing the resources after the first task is completed, to utilize data locality concepts, but for a streaming application such optimization's will not be possible since each task within the dataflow needs to be executing simultaneously to process incoming data in the data stream.

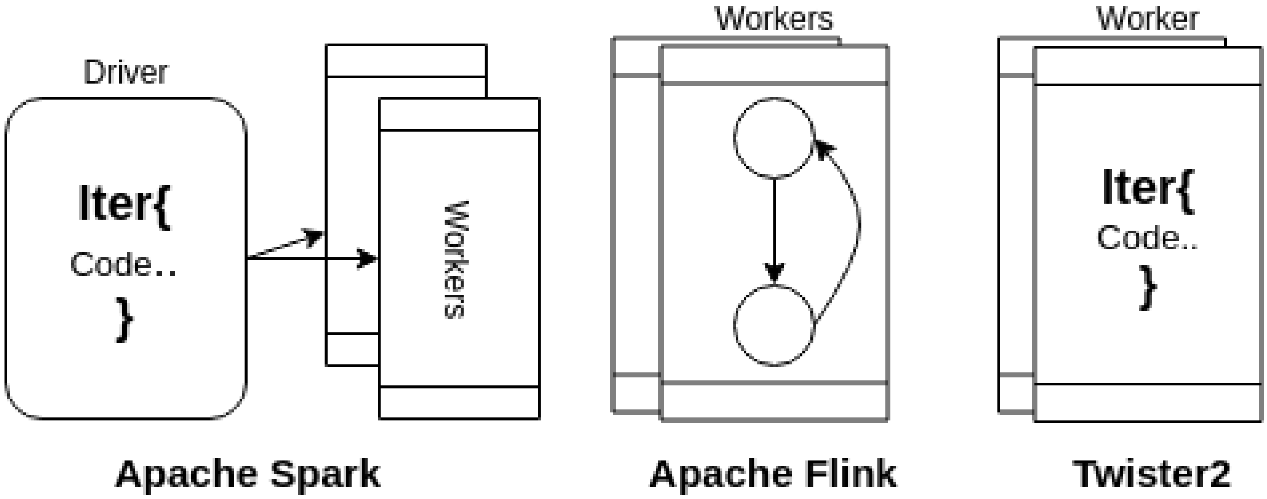

FIGURE 1 Different iteration models in Spark, Flink and Twister2

3.1 | Layered Model

The Twister2 dataflow model can be thought of as a layered structure which consists of 3 layers of abstraction, namely communication layer[14], task layer and the TSet layer (data layer). Each layer has a higher level of abstraction than the previous one. The most powerful aspect of the Twister2 design is that every layer is clearly defined through a set of API's which helps users to compose different implementations for the layers. This structure has two major benefits. First, the end users can choose to implement their application in any layer. And secondly, the framework has the freedom to create optimized implementations for each layer to improve performance. At the communication layer, the dataflow model is presented as a set of processes with data flowing between them through communication channels. At this level, the framework only provides the user with basic dataflow operations which model communication patterns such as gather, reduce, partition, etc.[14], these basic building blocks for the communication layer are derived from knowledge amassed in the HPC community on optimized communication patterns. At the task layer, communications are abstracted out, and the users interact with dataflow through a "Task", so the dataflow model is seen as a set of tasks that pass messages between them. The users model the application as a graph which consists of tasks and the connections between them.

The framework will handle the dataflow between tasks for the defined structure by utilizing the underlying communication layer. At the highest layer, termed as the TSet layer, the dataflow model is expressed as a set of transformations and actions on data. TSet's are similar to RDD's[15] in Spark or DataSets in Flink[3]. At the TSet layer, the user provides a set of transformations that need to be performed and actions that need to be executed to achieve the end result. Table 2 lists the basic transormations and actions that are made available through the TSet API. While the semantics of TSet's are similar to RDD's and DataSets, the difference in the underlying implementation and dataflow model allows TSet's to produce better performance in most cases. Details about the communication and task layer are beyond the scope of this paper, this paper will mainly focus on the TSet layer. However because the TSet layer is directly built on-top of the Task layer some aspects of the Task layer will be discussed.In section 4 we will look at the TSet layer in more detail to understand how it implements the dataflow model.

3.2 | Iterations

Iterations are a key part in many complex parallel applications, how well the framework handles iterations will directly impact the performance of the program. The approach taken for iteration handling in the Twister2 dataflow model is an important point that distinguishes it from other frameworks. Initially, in Apache Hadoop, iterative computations where addressed by writing intermediate data to disk and reading it back for the next iteration. This was very inefficient because of all the I/O operations involved, and led to the development of iterative MapReduce frameworks such as Spark[2] and Twister[16] which allowed in-memory operations, removing the need to write to disk at each iteration. This can be seen as moving the iterations from the client to the driver or master. However performing iterations at the driver, as in Spark, still creates the issue of having to gather results to the driver for each iteration and broadcasting them back. For example, if the algorithm needs to perform a reduce operation at the end of each iteration, the results must be collected at the driver, reduced to generate the answer, and then broadcast back to be used in the next iteration. This creates a bottleneck which hinders the performance of more complex iterative algorithms[10]. In Flink, iterations are embedded into the dataflow graph, the iterations are moved to the worker, however since the iterations are embedded in the dataflow graph, Flink does not support nested iterations.

Twister2 takes the handling of iterations a step further and manages iterations at the worker level, similar to the BSP model. Figure 1 summarizes how Spark, Flink, and Twister2 each handle iterations. This move eliminates the need to gather results at a centralized location before each iteration. Although this does not remove the need for synchronization after each iteration, it allows the framework and algorithms developed utilizing the framework to employ better communications patterns to achieve this. For example, at each iteration, the algorithm can perform an AllReduce operation to achieve synchronization rather than it being handled by a centralized entity such as a driver. The K-Means example discussed in section 7.1 will attest to this.

Implementing this model of iteration handling is straightforward at the communication layer once the BSP model is followed, however there is one major issue when exposing this model at higher level API's such as the task API or the TSet API. In both levels the program is modeled as some dataflow graph so in order to handle iterations the framework needs to embed the iterations into the graph itself as done in Apache Flink[3] or handle iterations within the worker as done in Twister2. This results in having to build dataflow graphs defined within the iteration block per iteration. This is because the dataflow graph defined by the user needs to be converted into an execution graph that can be executed by the task execution layer of the framework. This can be more clearly understood by looking at the KMeans algorithm in Listing. 1, when line 7 is being executed the dataflow graph for 'reducedMap' needs to be built for each iteration, which adds and unwanted overhead to the run-time. This issue is handled in the iterative execution mode which caches the post-build state of the dataflow graph from the first iteration for the subsequent iterations. The code change for this mode of execution are listed in Listing. 2. In Listing. 2 the 'cache()' method takes in an boolean value that specifies that it is a part of a iterative execution and the call in line 9 is used to clean up the environment after the execution is complete.

Listing 1: K-Means TSet API pseudo code

```
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 t = 0 to Iter do
6.    kMapTSet.addInput(centers)
7.    centers = reducedMap.cache()
8. end for;
```

Listing 2: K-Means TSet API pseudo code using iteration optimization

```
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 t = 0 to Iter do
6.    kMapTSet.addInput(centers)
7.    centers = reducedMap.cache(true)
8. end for;
9. reduced.finishIter();
```

The migration of iterations into the workers can be seen as the next logical step; from managing iterations at the client level as in Hadoop to managing them at a centralized entity as in Spark, and finally at the worker level. Such handling of iterations has been tested in the HPC community for decades and proven to be efficient, especially in MPI implementations. This claim is further strengthened in the results that are showcased in section 7 as well as from those in Twister:Net[14].

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 1 Twister2 TSet Organization

4 | TSET'S

TSet API is the highest level of abstraction provided by the Twister2 framework. TSet's are semantically similar to RDD's in Spark and DataSets in Flink. They provide the user with an API that allows them to program applications through a set of transformations and actions. TSet's also support both batch and stream processing. The concepts discussed in the following section apply to both modes; if a topic does not apply to both, it will be specifically noted.

Applications developed using TSet's are modeled as a graph where vertices represent computations and edges represent communications. The TSet programming API allows users to simply define the structure of the graph and then execute it. Underneath, the graph will be converted into a more detailed version in the task layer, which takes into account the parallelism of each vertex and data dependencies before execution. The main goal of this API is to provide a simpler, convenient and intuitive interface for users. Certain restrictions have been introduced to the TSet API (compared to Task and Communication APIs) to preserve the simplicity, while preserving the versatility as much as possible.

The TSet API consists of two main entities; "TSet" and "Link". Table 1 lists the TSet's and Link's currently supported by Twister2. These two entities are the main building blocks of the TSet API, A TSet can be seen as vertices of the graph and Link maps to an edge in the dataflow graph that is defined.

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 2 Twister2 TSet Transformations and Action

4.1 | TSet

A TSet represents a vertex in the dataflow graph. All applications start with a SourceTSet which can represent any data source. The application graph can be then built by connecting other TSet's with some Link in between them. There are a few restrictions on the combinations that can be made when creating the graph. For example, a KeyedReduce cannot be directly applied on a map since the former expects a key-value pair, therefore keyed operations can only be done after a MapToTuple transformation. These restrictions are embedded into the programming API so that the end user will not have to worry about creating invalid graphs.

All the TSets are separated into batch and streaming to closely reflect Twister2 Communication semantics. By functionality, there are two broad categories of TSets.

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

These distinctions allow the API to provide well structured methods for keyed communication operations. They also allow TSets to follow the lower level Task and Communication APIs more intimately.

4.2 | Link

A Link represents an edge in the dataflow graph, which is essentially a form of communication. Twister2 supports several types of Link's to cover all the basic communication patterns, as listed in Table 1. Each pattern is implemented in the communication layer using optimized algorithms to gain the best performance (Table 3). For example, the reduce operation is done using an inverted binary tree pattern to reduce the number of network calls that are made. These optimized communication patterns are described in more detail in Twister:Net[14].

TLinks are synonymous to the Task API communications, and they are designed to reflect Twister2 Communication semantics. These are elaborated in Table 3. Based on the communication message content, there are three categories of TLinks.

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

Just as its counterpart, TLinks are also separated into batch and streaming, as the communication semantics may change based on the operation mode.

4.3 | Lambdas, Functions and Operations

TSets are aimed to provide a convenient interface to quickly develop data flow logic. One key aspect there is the introduction of simplified functions to capture user logic. While using Task API, users would have to endure extending Twister2 Compute

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, Iterator<T>>>	m to n	IteratorTLink<Tuple<K, Iterator<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<JoinedTuple<K, U, V>>	m to n	IteratorTLink<JoinedTuple<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 3 Twister2 Communication semantics

interfaces to build the logic, whereas in TSets, they can simply generate the data flow logic using lambda functions. The API would translate that logic into Compute interfaces in the background. TSet API does not limit the users to just using Lambda functions. The end users are still free to create compute functions by directly implementing the relevant interfaces, One advantage of this approach is that user has access to more run-time meta-data as opposed to using Lambda functions because the TSet Context object is not available to functions written as lambda functions.

4.4 | Lazy Evaluation

Applications are evaluated in a lazy manner. That is, the graph will not be evaluated unless special transformation operations are invoked. Currently, foreach(...), cache() and sink(...) are such action operations that would result in graph execution. Figure 2 depicts an example TSet execution.

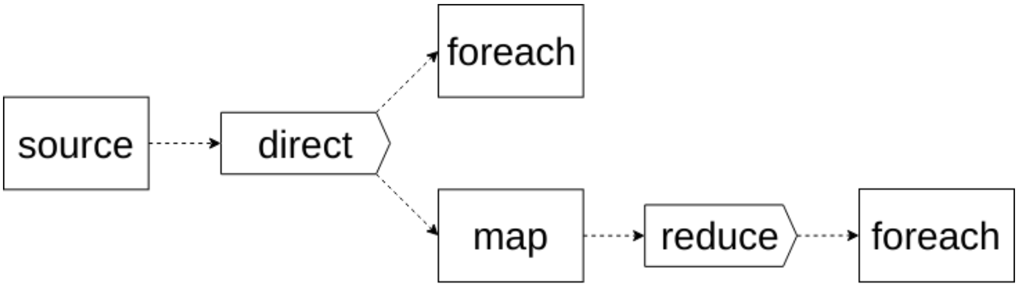

FIGURE 2 Example TSet execution

When an action is called on a particular TSet, a subgraph will be created, by tracing backwards to determine the lineage of the corresponding TSet. In the example provided, when the foreach operation is called on the Reduce TLink, a sub-graph will be created which includes *src, direct, map, reduce*, and *foreach* in order.

Alternatively, users can use the TSetEnvironment to execute the entire graph. This would build the entire TSet graph by traversing breadth-first, starting from the sources of the graph. In the example provided, *foreach* logic would need to be replaced by *map* TSets and then the graph can be executed entirely. In the streaming mode, this would be the default execution mechanism, because, in a streaming application, the data flow graph would be executed entirely, and executing a sub-graph is an undefined mode of operation for the streaming mode.

4.5 | Caching

The TSet API allows users to cache intermediate results that need to be used more than once during the computation. This can specially be beneficial if an iterative computation is involved. Using an cached result would remove the need to evaluate the graph repeatedly for each iteration. Calling the cache() method on a TSet results in the current graph to be evaluated and the results cached in-memory as a CachedTSet, which could be utilized in two ways.

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

4.6 | KMeans Walk through

The concepts in TSet's can be better understood by analyzing a simple algorithm, to this end we will look at how KMeans can be implemented using the TSet API. Listing. 1 shows the pseudo-code for implementing the K-Means algorithm using the TSet API. The first two lines load the points and centers and cache them in memory. The 'createSource' method takes in an source function that is responsible of reading the data in. Line 3 calls a map transformation on the cached points TSet. This map function contains the logic to calculate the new center allocations for each data point. New centers values are all-reduced to collect results from the parallel map operation (line 4), and finally, the last map operation (line 4) averages the values to generate the new centers, which are again added as an input for the next iteration (line 6). Both usages of caching mentioned in section 4.5 are used in this example, first the points CachedTSet is used as a starting point for the KMeans dataflow graph and secondly the centers CachedTSet is used as an input for the Kmeans calculation at line 6. The structure of the dataflow graph for the K-Means algorithm is shown in Figure 3. It is important to note that non on the transformations are actually evaluated until the 'cache()' action is invoked at line 7.

4.7 | Apache Beam Compatibility

Apache beam[8] is an open source framework that provides a unified model for defining both batch and stream data processing pipelines. Apache Beam[8] has gained a lot of attraction and has been widely adopted in both the academia and industry alike. Once defined using the unified programming model, the application/program which is called a pipeline in the Beam, is executed using one of the many distributed data-processing back-ends. Apache Spark[2], Google Cloud Dataflow[17], Apache Flink[3], Apache Apex[18] are some of the popular distributed processing back-ends supported by Beam. In order to support the execution of Beam pipelines each backend needs to convert the Beam program into a program that is understood by the back-end. For example if a user executes a Beam pipeline on a Spark back-end the framework will convert this program to a program based on Spark DataSet's. Being able to fully support Apache Beam as a distributed back-end opens up data processing frameworks to a larger audience as well as validates the completeness of the data processing framework.

Twister2 fully supports Apache Beam programs and can run Beam pipelines as a distributed data processing back-end. This is achieved by converting the Beam pipline into dataflow graphs using the TSet API. In order to fully support Beam pipelines data-processing back-ends need to support the following five primitives. All these primitives are easily supported using only a few constructs that are made available in the TSet API. (A PCollection is an unordered bag of elements in Apache Beam)

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

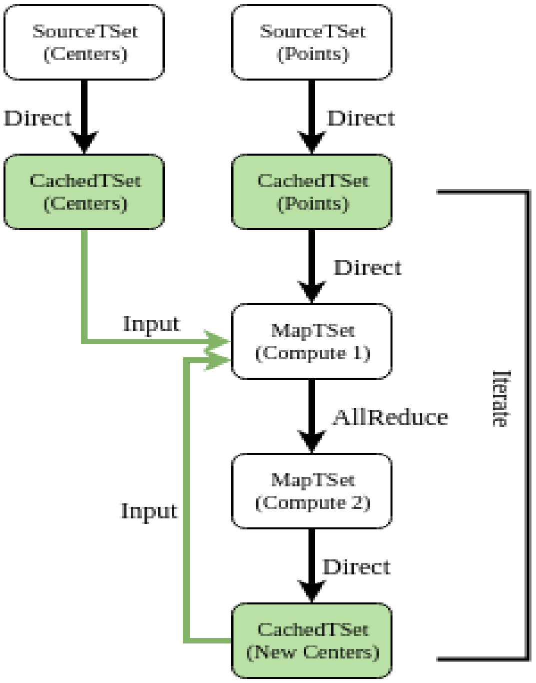

FIGURE 3 K-Means TSet API dataflow graph

5. Window - set the windowing strategy for a PCollection

5 | CONNECTED DATAFLOW

In big data analytics, it is essential to understand the size of the problem which could be further divided into parallelizable tasks. The coarse-grained parallelism[19] divides the larger problem into a small number of tasks and each task may be able to perform some set of operations which is also an example for task parallelism. However, fine-grained parallelism[19 20] divides the process into a large number of small tasks. However, it is important that the big data frameworks have the support for both coarse-grained and fine-grained parallelism in order to achieve better performance for the big data analytics process. The existing big data systems have traditionally adopted the coarse-grain parallelism to represent the computation in terms of Direct Acyclic Graphs(DAG). In Twister2, we introduced a new term named "Connected Dataflow" to support the coarse-grained parallelism. Using the connected dataflow, Twister2 is able to compose multiple independent or dependent dataflow graphs into a single entity. The connected dataflow graph consists of multiple dataflow graphs which are arranged based on the hierarchy of the input and output data. The data flows from one dataflow graph to another through memory and without using the disk. Internally, each dataflow graph consists of multiple subtasks which consists of parent-child relationships between the tasks. In other words, a dataflow graph has multiple task vertices and edges to connect those task vertices. The vertices represent the characteristics of computations and edges represent the communication between those computations. The representation of connected dataflow graph is shown in Fig. 4. The structure of dataflow graph 1 (DG1) and dataflow graph 2 (DG2) may or may not be identical in nature. Each dataflow graph may have a source, compute, and sink tasks. The different dataflow graphs are connected through communication edges(for example broadcast) in this example.

6 | RELATED WORK

While there is no formal agreed-upon definition for dataflow in the research literature, the term *dataflow* is used to denote various versions of the same underlying model. Lee et al[21] present dataflow process networks, which can be seen as a formal definition of the dataflow model to some extent. Misale et al[22] introduced a layered dataflow model building on top of[21] to represent big data analytics frameworks which follow the dataflow paradigm. The framework consists of three layers of dataflow models

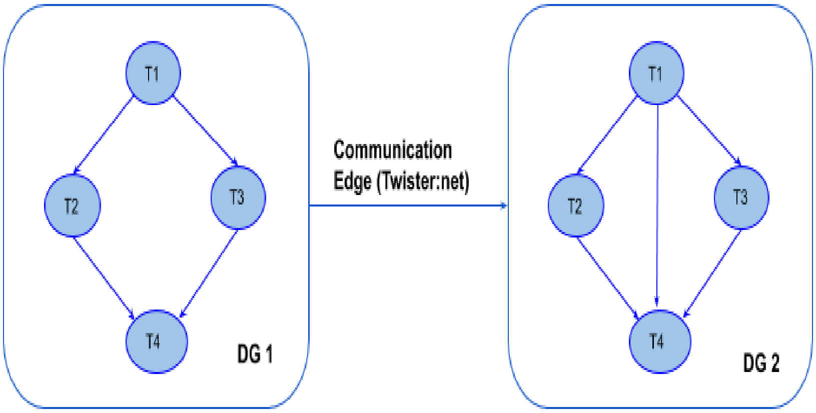

FIGURE 4 Connected Dataflow graph example

which describe different levels of abstractions present in modern data analytics frameworks. They also show how several popular frameworks map into those three layers. Those three layers can be loosely mapped to the communication, task and data layers of Twister2. Akidau et al[17] describe how Google cloud dataflow adopts the dataflow model in its design and implementation. Until Apache Spark 2.0[23] which has implemented with both coarse-grained and fine-grained mode when it is running over the Mesos. In the coarse-grained mode, each Spark executor runs as a single task. The spark executors are designed based on the following variables namely executor memory, executor cores, and number of executors.

Timely dataflow is another dataflow model which is implemented in the Naiad[24] system for iterative and incremental distributed computation processing. It works on the principle of stateful dataflow model, in which the dataflow nodes contain the mutable state and the edges carry the unbounded streams of data. Dryad[25] is a high-performance distributed execution engine for running coarse-grained data parallel applications which are embedded with an acyclic dataflow graph. In general, a Dryad application combines computational vertices with communication links to form a dataflow graph. In addition, the user has to program the dataflow graph in Dryad explicitly. Hierarchical Task Graph[26] is an intermediate way to expose and exploit the coarse-grain parallelism of an application. The main objective of the technique is to provide a program dependency representation without a cyclic dependency. It hides the cyclic dependencies by encapsulating within a super node. Connected Components in Apache Flink[3] is able to identify the larger part of the graph which are connected based on all the task vertices of the graph with the same component ID. It is an iterative algorithm which is implemented based on the delta iteration. Turbine[27] is a distributed many-task dataflow engine which uses extreme-scale computing to evaluate program overhead and generation of tasks. The execution model of the system breaks parallel loops and invocation of concurrent functions into fragments for the execution of tasks in a distributed manner. 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 which discusses the role of dataflow for parallel programs such as MPI applications. Strout et al[28] discuss the importance of understanding dataflow within MPI programs and introduce a dataflow analysis framework for an MPI implementation. Pop et al[29] introduce OpenStream, a dataflow extension to OpenMP, and discuss the advantages of such a model.

Frameworks such as Apache Beam[8] strive to provide users with a single API which can be used to develop parallel data processing applications. Apache Beam was developed as the programming model for Google Cloud Dataflow[17]. Li et al[30] discuss the challenges faced when adding IBM streams[31] as an distributed data processing backend for Beam.

7 | EVALUATION

In order to evaluate the performance of Twister2 at both the Task and TSet layers and to understand the overheads created by each layer of abstraction, we developed and evaluated a couple of applications. To this end, Twister2 is evaluated against identical (algorithmically) implemented versions of the same algorithms and applications in OpenMPI (v4.0.1) and Spark (v2.4). This evaluation focuses more on understanding the performance of Twister2 Task layer and TSet layer. A more detailed evaluation of

Twister2 which involves other frameworks such as Flink, Storm, Heron, etc. can be found at[14]. The applications implemented to evaluate the performance are K-Means and Distributed SVM. Additionally, one more test is performed to understand the overheads created when additional nodes are added to a dataflow graph.

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. K-Means and SVM are the algorithms discussed in this paper, considering iterative-based and ensemble-based designing respectively.

7.1 | K-Means

With the ever increasing volume of data, big data analytic techniques are being used more and more for data analysis. One such analytical process is the painstaking process of clustering numerous datasets containing large numbers of records with high dimensions. Traditional sequential clustering algorithms are unable to scale to the level required to manage larger data-sets because of memory space and time complexities. Therefore, the parallelization of data clustering algorithms is paramount when dealing with big data. K-Means clustering is an popular iterative algorithm, hence it requires a large number of iterative steps to find an optimal solution. This procedure increases the processing time of clustering, however parallel KMeans implementations are able to handle large data-sets by performing the calculations in a distributed manner. Twister2 provides both a dataflow task graph-based and TSet-based approach to distribute the tasks in a parallel manner and aggregate the results, which reduces the processing time of the K-Means clustering process. The pseudocode for the K-Means algorithm is given in Listing. 1 and Listing. 2 , the second utlizes the iteration optimization described in section 3.2.

Figure. 5 shows the total compute time (data loading times are not included) for K-means on a 16-node cluster with a parallelism of 128 for OpenMPI, Spark, Twister2 Task and Twister2 TSet layers, The Tset layer has two implementations which correspond to Listing. 1 (TSet) and Listing. 2 (TSet Iterative). Each run was done with 2 million data points with 2 features and a varying number of centers starting from 1000 centers and increasing up to 16000 centers for 100 iterations. As seen in the graph 5 both Task layer implementation and the TSet Iterative implementation perform on the same level as OpenMPI based implementation. This shows that while the TSet API provides a very high level abstraction for developing complex parallel applications it adds very little overheads to the program. It can be clearly seen that Apache Spark performs 3-4x slower than both TSet Iterative and Task implementations of Twister2. The basic TSet based implementation which is based on the algorithm in Listing. 1 shows the overheads that are added by the repeated build step for the dataflow graphs as discussed in section 3.2. The efficient handling of iterations in Twister2 is the main reason it is able to outperform Spark for iterative algorithms such as K-Means. A detailed evaluation of the Twister2 communication framework and how it stacks up against OpenMPI and other frameworks are presented in Twister:Net[14], Twister:Net is the underlying communication framework that is used by the Task layer, and by the TSet API (through the Task layer).

7.2 | SVM

Support Vector Machines (SVM) algorithm is one of the prominent machine learning algorithms used by most of the researchers in vivid domain sciences. SVM can be implemented in various ways. Three major types of implementations on SVM are Matrix decomposition methods, sequential minimal optimization-based methods, and stochastic gradient descent-based methods. The latter is proven to be efficient in terms of both computation and communication. Implementation can be done as an iterative or ensemble model. Ensemble method is a very popular type of implementation among domain scientists. Twister2 SVM implementation uses SGD-based ensemble model, and the designed model is highly efficient training-wise. Twister2 SVM was evaluated on a 16-node cluster with variable parallelism with all node usage enabled. In this research, the idea is to showcase how different implementation methods can be used to train SVM and understand which methods are optimized based on the purpose of the application programmer. In the ensemble model, the computation takes place using the implicit communication and computation models in each framework. In the developed model, weight calculation is done in each process until the given number of iterations are computed (till convergence) and in the final step, a communication is done to synchronize the model over the parallel workers.

OpenMPI is one of the prominent choices by high performance computing community in application programming. The programming model in MPI is mainly favorable for bulk synchronous parallel model. Allreduce collective communication has

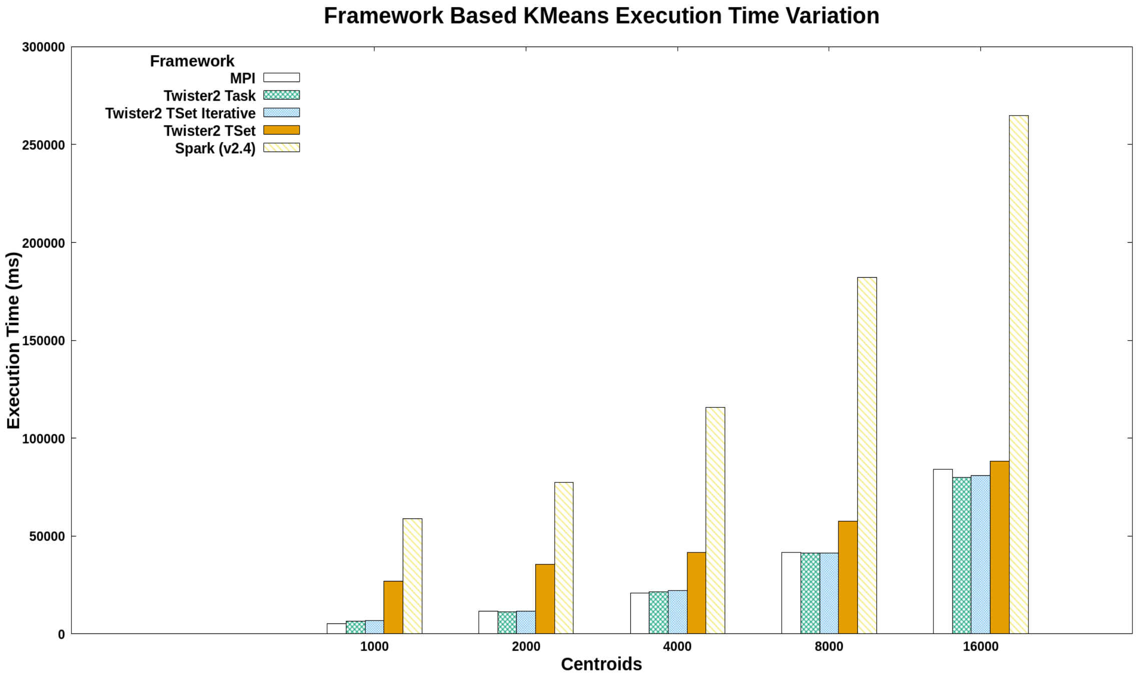

FIGURE 5 K-Means job execution time on 16 nodes with varying centers, 2 million data points with 128-way parallelism for 100 iterations.

been used as the main model synchronization tool in OpenMPI to implement the application. Spark implementation based on RDD uses a basic driver and worker model in Spark. For model synchronization, there is a call from each worker to driver and model in synchronized in the final stage. In Twister2 framework, we implemented the SVM model in two different ways. One method resembles to a similar way like Spark as far as programming abstraction is considered. This method is known as TSet based implementation. The other programming abstraction is a task graph based programming model similar to that of Apache Storm API. Twister2 underneath uses ISend, IRecv function calls to implement its own communication model based on tree based optimization. The task level abstraction provides a convenient programming model in writing these applications. From the conducted experiments it is evident that the Twister2 Task and Twister2 TSet APIs are performing better than the Apache Spark RDD based computation model. Even though OpenMPI based implementation is much faster than the other implementations. But the gap in performance time in the MPI-based implementation and Twister2-based implementation is lesser than that of Twister-based implementations and Spark RDD-based implementations. The results of the experiments are shown in Figure 6.

7.3 | Dataflow Node

There are many tests that can be performed to understand the overheads in dataflow systems. One important test would be to evaluate the increased overhead when a single dataflow node is added to the dataflow graph. The test results will also highlight the importance of a task execution optimizer that merges consecutive tasks when possible. To test this, we created two dataflow graphs with one map vertex that performs no operation. The first graph is "source-map-allreduce" and the second is "source-map-map-allreduce". The operations were done for 200K data points and 100 iterations in both frameworks. Figure. 7 shows the results of running these two configurations with parallelisms of 128 and 256 for both Twister2 (Task layer) and Spark. From the results, we can observe a slight overhead in Twister2 framework. This overhead is not present in Spark because Spark stages consecutive map operations and runs them as a single pipeline unit, removing the overhead. This is performed through an optimizer that evaluates the dataflow graph and mergers map operations when possible. Even with the overhead, it can be observed that Twister2 framework outperforms Spark by a large factor. This is because of the optimized communication patterns that are employed by Twister2 and its efficient model for iterations. Task execution optimizations such as pipelining will be introduced into Twister2 in future updates to address these overheads.

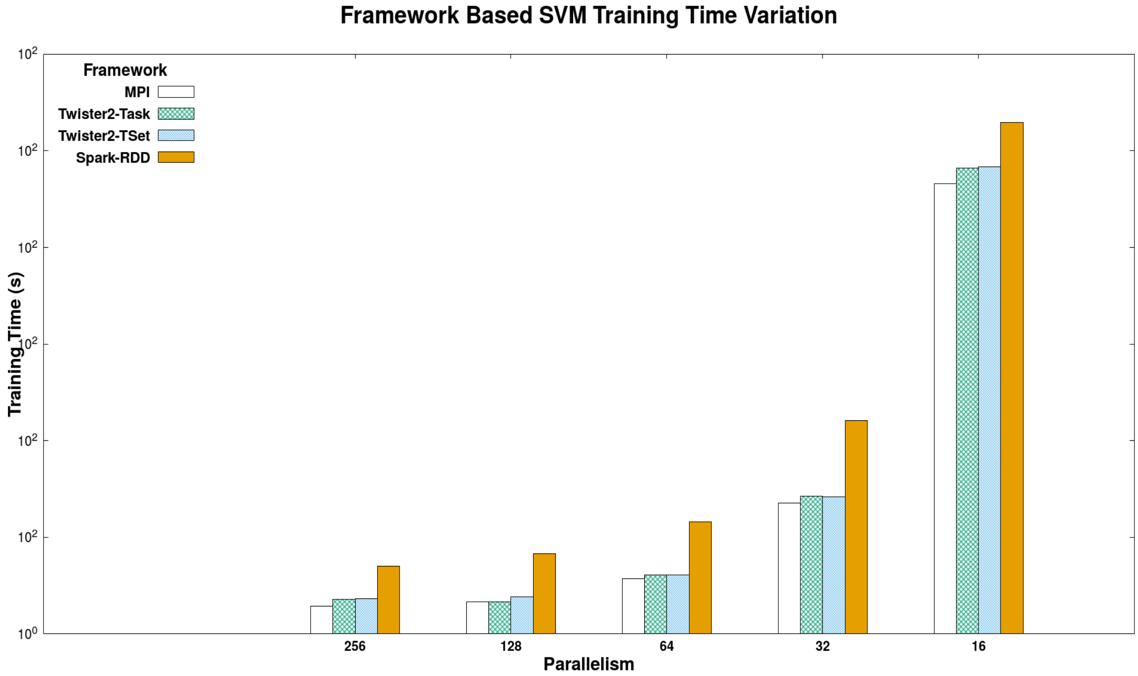

FIGURE 6 SVM job execution time for 320K data points with 2000 features and 500 iterations, on 16 nodes with varying parallelism

8 | CONCLUSION AND FUTURE WORK

This paper presented TSet's, a hybrid approach for dataflow-based iterative programs. As seen in the evaluation results the TSet implementation achieved comparable performance compared to OpenMPI and much better performance compared to Apache Spark for K-Means and SVM applications. Even for the dataflow node experiment, even though the overhead was larger in TSet program, overall TSet performed much better than Apache Spark. We believe there are many aspects of the framework that can be further improved to reduce overhead and draw it closer to OpenMPI performance.

Twister2 provides a dataflow model that is similar to Apache Spark, where a central scheduler is used for allocating parallel tasks. Such a design is more suitable for workflow-style applications but unsuitable for parallel applications, as the overhead of the central scheduler (distributing tasks to workers at each iteration as a "sequential bottleneck") is high. We are actively working on incorporating checkpointing-based fault tolerance to the system for both streaming and batch applications. While Twister2 supports Apache Beam piplines, The Twister2 distributed processing back-end, also known as a runner, is based on the initial version of Apache Beam runner, recently Apache Beam has introduced a newer design for the runners named "portable runners" which allow Beam piplines to be written in any of the supported SDK's such as Java, Python, GO, etc. We are actively working on developing a Twister2 portable runner for Apache Beam and integrating the finished runner directly with the Apache Beam project.

Twister2 follows a well-defined layered model to handle data flow applications. Currently, the framework is built entirely upon Java, and development has started on plugging in a C++ Twister2 Communication layer for better performance. The goal is to manage memory optimally, because Java garbage collection has been a bottleneck for high performance applications. Garbage collection pauses have been a limiting factor in many data analytics frameworks based on Java, including Apache Spark[32].

There is an opportunity for an optimizer in the Twister TSet API. An optimizer would parse the user's TSet Graph and eliminate redundant data operations. Furthermore, it would also be possible to merge subsets of operations that could be executed on a single execution.

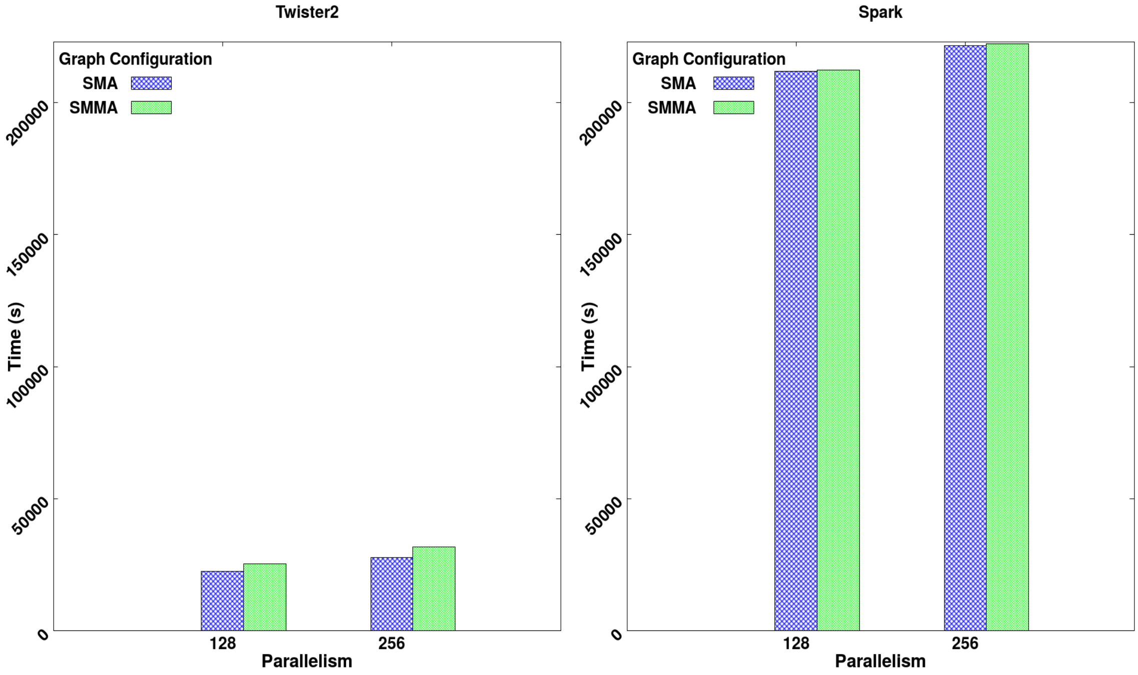

FIGURE 7 Execution time for Source-Map-AllReduce (SMA) and Source- Map-Map-AllReduce(SMMA) graph configurations. With 200K data points and 100 iterations

9 | ACKNOWLEDGMENT

This work was partially supported by NSF CIF21 DIBBS 1443054 and the Indiana University Precision Health initiative. We thank Intel for their support of the Juliet and Victor systems and extend our gratitude to the FutureSystems team for their support with the infrastructure.

References

1. White T. *Hadoop: The definitive guide*. " O'Reilly Media, Inc." . 2012.
2. Zaharia M, Xin RS, Wendell P, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM* 2016; 59(11): 56–65.
3. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 2015; 36(4).
4. Toshniwal A, Taneja S, Shukla A, et al. Storm@ twitter. In: ACM. ; 2014: 147–156.
5. Kulkarni S, Bhagat N, Fu M, et al. Twitter heron: Stream processing at scale. In: ACM. ; 2015: 239–250.
6. Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning. In: ; 2016: 265–283.
7. Ketkar N. Introduction to pytorch. In: Springer. 2017 (pp. 195–208).
8. Apache Beam: An advanced unified programming model . Accessed: Aug 28 2019.
9. Dai J, Huang J, Huang S, Huang B, Liu Y. HiTune: dataflow-based performance analysis for big data cloud. *Proc. of the 2011 USENIX ATC* 2011: 87–100.
10. Kamburugamuve S, Wickramasinghe P, Ekanayake S, Fox GC. Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink. *The International Journal of High Performance Computing Applications* 2018; 32(1): 61–73.

11. Twister2: Flexible, High performance data processing. .

12. Kamburugamuve S, Govindarajan K, Wickramasinghe P, Abeykoon V, Fox G. Twister2: Design of a big data toolkit. *Concurrency and Computation: Practice and Experience* 2017: e5189.

13. Wickramasinghe P, Kamburugamuve S, Govindarajan K, et al. Twister2: TSet High-Performance Iterative Dataflow. In: IEEE. ; 2019: 55–60.

14. Kamburugamuve S, Wickramasinghe P, Govindarajan K, et al. Twister: Net-communication library for big data processing in hpc and cloud environments. In: IEEE. ; 2018: 383–391.

15. Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: USENIX Association. ; 2012: 2–2.

16. Ekanayake J, Li H, Zhang B, et al. Twister: a runtime for iterative mapreduce. In: ACM. ; 2010: 810–818.

17. Akidau T, Bradshaw R, Chambers C, 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* 2015; 8(12): 1792–1803.

18. Apache Apex: Enterprise-grade unified stream and batch processing engine. Accessed: Aug 28 2019.

19. Loshin D. Chapter 14 - High-Performance Business Intelligence. In: Loshin D. , ed. *Business Intelligence (Second Edition)*MK Series on Business Intelligence. Morgan Kaufmann. second edition ed. 2013 (pp. 211 - 235)

20. Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Software available from tensorflow.org.

21. Lee EA, Parks TM. Dataflow process networks. *Proceedings of the IEEE* 1995; 83(5): 773–801.

22. Misale C, Drocco M, Aldinucci M, Tremblay G. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters* 2017; 27(01): 1740003.

23. Spark$_{C}oarsegrained.Accessed : Feb$242019.

24. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: a timely dataflow system. In: ACM. ; 2013: 439–455.

25. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. In: . 41. ACM. ; 2007: 59–72.

26. HTG. Accessed: Feb 24 2019.

27. Wozniak JM, Armstrong TG, Maheshwari K, et al. Turbine: A distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae* 2013; 128(3): 337–366.

28. Strout MM, Kreaseck B, Hovland PD. Data-flow analysis for MPI programs. In: IEEE. ; 2006: 175–184.

29. Pop A, Cohen A. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 2013; 9(4): 53.

30. Li S, Gerver P, MacMillan J, Debrunner D, Marshall W, Wu KL. Challenges and experiences in building an efficient apache beam runner for IBM streams. *Proceedings of the VLDB Endowment* 2018; 11(12): 1742–1754.

31. Hirzel M, Andrade H, Gedik B, et al. IBM streams processing language: analyzing big data in motion. *IBM Journal of Research and Development* 2013; 57(3/4): 7–1.

32. Maas M, Harris T, Asanović K, Kubiatowicz J. Trash day: Coordinating garbage collection in distributed systems. In: ; 2015.