Twister2:TSet High-Performance Iterative Dataflow

Pulasthi Wickramasinghe
SICE
Indiana University
Bloomington, IN, USA
pswickra@iu.edu

Supun Kamburugamuve
SICE
Indiana University
Bloomington, IN, USA
skamburu@indiana.edu

Kannan Govindarajan
SICE
Indiana University
Bloomington, IN, USA
kgovind@iu.edu

Vibhatha Abeykoon
SICE
Indiana University
Bloomington, IN, USA
vlabeyko@iu.edu

Chathura Widanage
SICE
Indiana University
Bloomington, IN, USA
cdwidana@iu.edu

Niranda Perera
SICE
Indiana University
Bloomington, IN, USA
dnperera@iu.edu

Ahmet Uyar
DSC
Indiana University
Bloomington, IN, USA
auyar@iu.edu

Gurhan Gunduz
DSC
Indiana University
Bloomington, IN, USA
ggunduz@iu.edu

Selahattin Akkas
SICE
Indiana University
Bloomington, IN, USA
sakkas@iu.edu

Geoffrey Fox
SICE
Indiana University
Bloomington, IN, USA
gcf@indiana.edu

***Abstract*—The dataflow model is slowly becoming the de facto standard for big data applications. While many popular frameworks are built around the dataflow model, very little research has been done on understanding the inner workings of the dataflow model. This has led to many inefficiencies in existing frameworks. It is important to note that understanding the relation between dataflow and HPC building blocks allows us to address and alleviate many of the fundamental inefficiencies in dataflow 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.**

Keywords— dataflow, big data, mapreduce, batch, stream, iterative, parallel programming

I. Introduction

In recent years, the big data domain has seen a massive increase in popularity because of the ever-increasing 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[21], Spark[1], Flink[2] focuses on batch processing, Storm[3], Heron[4], Flink[2] targets on stream processing, and TensorFlow[5], PyTorch[6], for machine learning are just a few popular examples for such frameworks.

Such frameworks provide various higher-level abstractions and API's for end users to program applications hiding the complexities of parallel programs. Even though each framework has its own abstraction and implementation, it is observable that most frameworks share a common dataflow programming model. Once an application is developed using the dataflow model, the runtime [16] system takes the responsibility of dynamically mapping the dataflow graph into an execution graph. This execution graph is then executed on a cluster as a distributed program. Another important observation is the similarities that data analytics frameworks have with 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, partition operations which will be discussed in more detail in section II. However, these similarities and the common model is not very well defined in the research literature. A good understanding of the dataflow model and how each framework has implemented it would help to build more optimized systems. In [15] the authors discussed this topic in more detail, and the findings motivated the development of Twister2 [22] [9], 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's is the dataflow abstraction of Twister2, which will be discussed in more detail in section III.

Iterations are one of the core building blocks of parallel applications. Frameworks built around the dataflow model handle iterations with 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 [15]. 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 [15]. The handling of iterations in Twister2 is a major distinction between Twister2 and other popular frameworks such as Spark[1], and Flink[2]. This will be discussed in more detail in section III.

In this paper, we first discuss the generic dataflow model and how different capabilities of the dataflow model can be

mapped to those in the HPC domain using MPI as an example. 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. Finally, 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

II. Comparison of DataFlow and MPI

MPI is a generic messaging standard that can support different programming models. Most MPI programs are written using the BSP model. 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 widely used frameworks designed for as streaming, data analysis, and graph processing.

Collective operations are arguably the major use case of MPI for parallel programs. 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. MPI collectives in its pure form are hard to use in data applications as their requirements are slightly different. Twister:Net [7] 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 following requirements that make them slightly different from MPI specification based collectives.

DataFlow collectives are driven by following requirements that make them slightly different from MPI specification based collectives.

1. The collectives are between a set of tasks in an arbitrary task graph.
2. Collectives handle data that doesn't fit in memory
3. Dynamic data sizes for operators.
4. Keys are part of the abstraction.
5. Collectives can involve imbalanced data and requires termination detection.

The dataflow programming model is different from the BSP model in many aspects. Dataflow mostly is an event-driven model where user programs a set of event handlers arranged in a graph. The user is hidden from 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 while user handles aspects such as thread and data management. It is a challenge to preserve MPI performance while providing a higher level abstraction that can be used for data analytics and TSets is an attempt to balance both.

III. Twister2 DataFlow model

Dataflow is the preferred choice for processing large-scale data. 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 could 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 consists 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 a long-running or short-running which depends on the type of application. In static dataflow graph, the structure of the complete graph is known at compile time whereas, in dynamic dataflow graph, the structure of the graph is not known at compile time 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.

The Twister2 framework has been developed around the dataflow model such that it supports both streaming and batch operations as first-class concepts.

A. Layered Model

The Twister2 dataflow model can be thought of as a layered structure, which consists of 3 layers of abstraction namely communication layer [7], task layer and the TSet layer (data layer), each layer has a higher level of abstraction than the previous layer. The most powerful aspect of the Twister2 design is that each layer is clearly defined through a set of API's which users can use to compose different implementations for each layer. This layered 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. [7]. 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 [17] in Spark or DataSets in Flink [2]. At the TSet

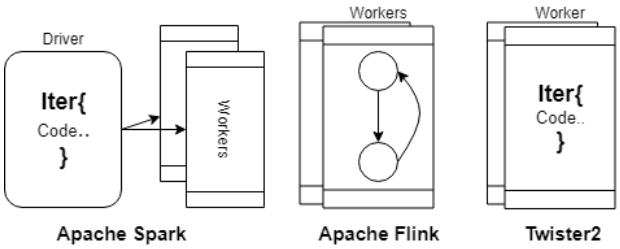

Fig. 1. Iteration model for Spark, Flink and Twister2

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. 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, in section IV we will look at the TSet layer in more detail to understand how the TSet layer implements the dataflow model.

B. *Iterations*

How iterations are handled in the Twister2 dataflow model is an important point that distinguishes it from other frameworks. The ability to handle iterations efficiently is essential for a big data framework, in order to support complex parallel applications. Initially, in Hadoop, the iterative computations needed to be addressed by writing data to disk and reading it back for the next iteration. This was very inefficient and led to the development of iterative MapReduce frameworks such as Spark[1] and Twister[18] which allowed in-memory operations, removing the need to write to disk for 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 done 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 need to 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 [15]. In Flink iterations are embedded into the dataflow graph, the iterations are moved to the worker; however since the iteration are embedded in the dataflow graph, it does not support nested iterations.

Twister2 takes the handling of iterations a step further and manages iterations at the worker level, similar to how iterations are managed in the BSP model. Fig. 1 summarizes how Spark, Flink, and Twister2 manage iterations. This move eliminates the need to gather results at a centralized location before each iteration. However, this does not remove the need for synchronization after each iteration, but 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 VI will attest to this.

TABLE I. TSET API OPERATIONS

TSet	*Source: void => O* *Map: I => O* *FlatMap: I => O* *GroupBy: V => K,V* *Cache: I => I (cached) (batch only)* *Sink: I => I (result)*
Link	*Direct: I => I (1-to-1 mapping)* *Partition: I => I (N-to-M mapping)* *Reduce: I => I (N-to-1 mapping)* *Gather: I => Iter[I] (N-to-1 mapping)* *AllReduce: I => I (N-to-M mapping)* *AllGather: I => Iter[I] (N-to-M mapping)* *KeyedPartition: K, V => K, V (N-to-M mapping)* *KeyedReduce: K, V => K, V (N-to-M mapping) (batch only)* *KeyedGather: K, V => K, Iter[V] ((N-to-M mapping) (batch only)*

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

IV. TSETS

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. It provides the user with an API that allows them to program applications through a set of transformations and actions. It also supports both batch and stream processing, the concepts discussed in the section apply to both modes; if a specific 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 the users to simply define the structure of this graph and then execute it. Underneath, the graph will be converted into a more detailed graph in the task layer which takes into account parallelism of each vertex and data dependencies, before executing it. The TSet API consists of two main entities which are namely "TSet" and "Link." Table I lists the TSet's and Link's that are currently supported by Twister2.

A. *TSet*

A TSet represents a vertex in the dataflow graph. All applications start with a SourceTSet which can represent any datasource. The application graph can be then built by connecting other TSet's with some Link in between them. However, there are 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 *GroupBy*. These restrictions are embedded into the programming API so that the end user will not have to worry about creating invalid graphs.

```
1: CachedTSet points = tc.createSource(...).cache()
2: CachedTSet centers = tc.createSource(...).cache()
3: for t = 0 to Iter do
4:     MapTSet kMap = points.map(...)
5:     kMap.addInput(centers)
6:     AllReduceTLink reduced = kMap.allReduce(...)
7:     centers = reduced.map(...).cache()
8: end for
```

Fig. 2: K-means TSet API pseudocode

B. 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 I Each pattern is implemented in the communication layer using optimized algorithms to gain the best performance, for example, the reduce operation is done using an inverted binary tree pattern to reduce the number of network calls that are made.

C. Lazy Evaluation

Applications are evaluated in a lazy manner. That is the graph will not be evaluated unless special transformation operations are done. Currently, *cache()* and *sink()* are the two action operations that would result in a graph execution.

D. Caching

The TSet API allows users to cache intermediate results that need to be used more than once during the computation. Calling the *cache()* method on a TSet result in the current graph to be evaluated and the results to be cached in-memory as a *CachedTSet*, which could be utilized in two ways. First, users can treat it as a source and build a new graph with it as a starting point. Secondly, it can be set as an input to a separate TSet so the cached values can be accessed during computation steps.

The concepts in TSets can be better understood by analyzing a simple algorithm. Fig. 2 shows the pseudocode for implementing the K-Means algorithm using the TSet API. The first two lines load the points and centers and cache them in memory. Line 4, 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. These new centers values are all-reduced to collect results from the parallel map operation (line 6), and finally, the last map operation (line 7) averages the values to generate the new centers, which are again added as an input for the next iteration (line 5). The structure of the dataflow graph for the K-Means algorithm is shown in Fig. 3.

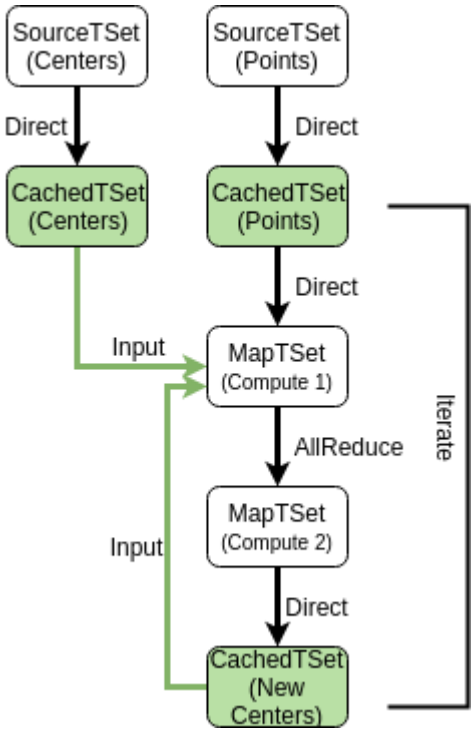

Fig. 3. K-Means TSet API dataflow graph

V. 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. In [10] the authors present dataflow process networks, which can be seen as a formal definition of the dataflow model to some extent. [11] introduces a layered dataflow model building on top of [10] to represent big data analytics frameworks which follow the dataflow paradigm, they introduce three layers of dataflow models which describe different levels of abstractions that are present in modern data analytics frameworks. They show how several popular frameworks map into those three layers. Those three layers can be loosely mapped to the communication, task and data layer of Twister2. In [12] the authors describe how Google cloud dataflow adopts the dataflow model in its design and implementation.

Timely dataflow is another dataflow model which is implemented in the Naiad[8] 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 [19] 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 to that, the user has to program the dataflow graph in the Dryad explicitly. Turbine [20] 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 to the user which is different from Dryad and similar to the Turbine model.

The relation between dataflow and MPI at an operator level is not well defined within the research literature to our knowledge. However, there is work done in this area which discusses the role of dataflow for parallel programs such as MPI programs. In [13] authors discuss the importance of understanding dataflow within MPI programs and introduce a dataflow analysis framework for an MPI implementation. In [14] the authors introduce OpenStream which is a dataflow extension to OpenMP and discuss the advantages of such a model.

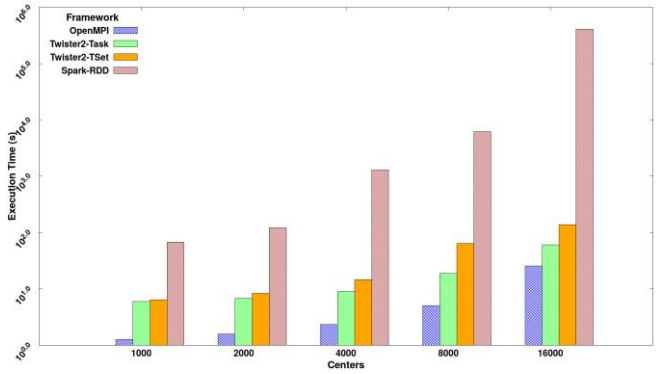

Fig. 4. K-Means job execution time on 16 nodes with varying centers, 2 million data points with 128-way parallelism for 100 iterations.

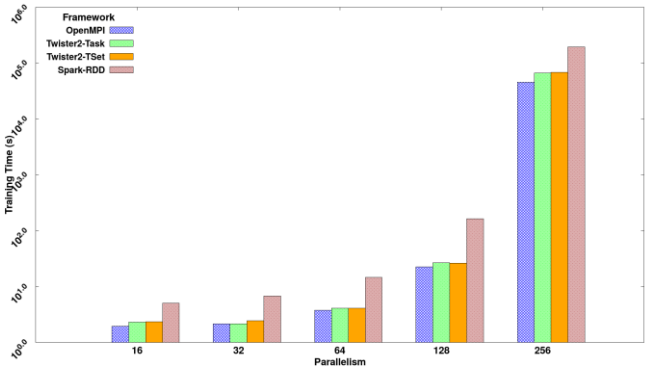

Fig. 5. SVM job execution time for 320K data points with 2000 features and 500 iterations, on 16 nodes with varying parallelism

VI. Evaluation

In order to test the performance of Twister2 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 (v3.1.2) 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 [7]. The applications implemented to evaluate the performance are namely K-Means and Distributed SVM.

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 and 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.

A. Kmeans

The need to process large amounts of continuously arriving information has led to the exploration and application of big data analytics techniques. Likewise, the painstaking process of clustering numerous datasets containing large numbers of records with high dimensions calls for innovative methods. Traditional sequential clustering algorithms are unable to handle it. They are not scalable in relation to larger sizes of data sets, and they are most often computationally expensive in memory space and time complexities. Yet, the parallelization of data clustering algorithms is paramount when dealing with big data. K-Means clustering is an iterative algorithm hence, it requires a large number of iterative steps to find an optimal solution, and this procedure increases the processing time of clustering. Twister2 provides both dataflow task graph-based and TSet-based approach to distribute the tasks in a parallel manner and aggregate the results which reduce the processing time of the K-Means clustering process. The pseudocode for the K-Means algorithm is given in Fig. 2.

Fig. 4 shows the total running time for K-means on a 16 node cluster with a parallelism of 128 for OpenMPI, Spark, Twister2 Task and Twister2 TSet layers. Each run was done with 2 million data points with 2 features and a varying number of centers. The results show that Twister2 out performs Spark, it also shows that the TSet layer adds a small but expected overhead over the Task layer in Twister2. BSP style K-means outperforms both systems which are expected because of the low level abstractions and minimal overheads in OpenMPI. The efficient handling of iterations in Twister2 is the main reason Twister2 is able to outperform Spark for iterative algorithms such as K-Means.

B. SVM

In the supervised learning algorithm domain, Support Vector Machines (SVM) is one of the prominent algorithms used by most of the researches related to vivid domain sciences. In the SVM algorithm, there are three major types of implementations. Matrix decomposition methods, sequential minimal optimization based methods, and stochastic gradient descent based methods. The latter is proven to be computation and communication wise efficient. 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 training wise highly efficient. Twister2 SVM was evaluated on 16 node cluster with variable parallelisms with all node usage enabled. From Fig. 5 we can observe that both Twister2 implementations outperform Spark and manage to perform at the same level as OpenMPI.

C. 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 overhead added when a single dataflow node is added to the dataflow graph. To test this, we created two dataflow graphs, with one map vertice 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. Fig. 6 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 pipelined unit, removing the overhead. However, even with

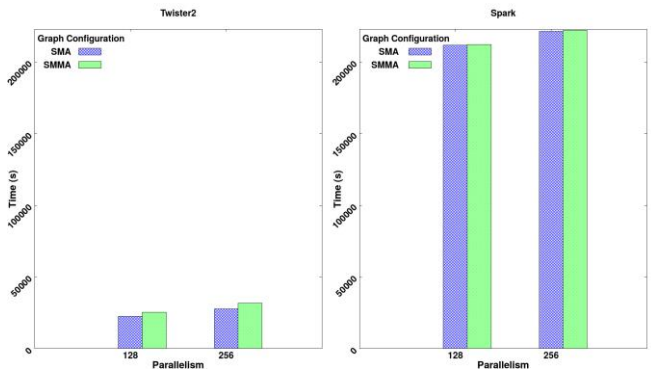

Fig. 6. Execution time for Source-Map-AllReduce (SMA) and Source-Map-Map-AllReduce(SMMA) graph configurations. With 200K data points and 100 iterations

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 the future updates to address these overheads.

VII. Conclusion and Future Work

This paper presented a hybrid approach for dataflow based iterative programs. The implementation achieved reasonable performance compared to OpenMPI and much better performance compared to Spark for K-Means and SVM applications. We believe there are many aspects of the framework that can be improved further to reduce the overheads and make it closer to OpenMPI performance.

Twister2 provides a dataflow model that is similar to Spark where a central scheduler is used for allocating parallel tasks. Such a central scheduler is more suitable for workflow style applications and not suitable 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.

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] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, M. J. Franklin et al., "Apache spark: a unified engine for big data processing, "Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine, "Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4, 2015.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulka-rni, J. Jackson, K. Gade, M. Fu, J. Donhamet al., "Storm@ twitter,"inProceedings of the 2014 ACM SIGMOD international conference onManagement of data. ACM, 2014, pp. 147–156.

[4] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M.Patel, et al., "Twitter heron: Stream processing at scale," inProceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015, pp. 239–250.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean et al., "Tensorflow: A system for large-scale machine learning," in12th{USENIX}Symposium on Operating Systems Design and Implementation ({OSDI}16), 2016, pp. 265–283.

[6] N. Ketkar, "Introduction to pytorch," inDeep learning with python.Springer, 2017, pp. 195–208.

[7] S. Kamburugamuve, P. Wickramasinghe, K. Govindarajan, A. Uyar, G. Gunduz, V. Abeykoon, and G. Fox, "Twister:Net-communication library for big data processing in hpc and cloud environments," in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD).IEEE, 2018, pp. 383–391.

[8] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013, pp.439–455.

[9] S. Kamburugamuve, K. Govindarajan, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Twister2: Design of a Big Data Toolkit" in Concurrency and Computation, Practice and Experience Special issue for EXAMPI 2017 workshop November 12 2017 at SC17 conference, Denver CO 2017. Published as https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.5189.

[10] E. A. Lee and T. M. Parks, "Dataflow process networks, "Proceedings of the IEEE, vol. 83, no. 5, pp. 773–801, 1995.

[11] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay, "A comparison of big data frameworks on a layered dataflow model, "Parallel Processing Letters, vol. 27, no. 01, p. 1740003, 2017.

[12] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fern´andez-Moctezuma, R. Lax 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, vol. 8, no. 12, pp. 1792–1803, 2015.

[13] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for mpi programs," in Parallel Processing, 2006. ICPP 2006. International Conference on. IEEE, 2006, pp. 175–184.

[14] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of OpenMP streaming programs, "ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 4, p. 53, 2013.

[15] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake, and G. C. Fox, "Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink, "The International Journal of High-Performance Computing Applications, vol. 32, no. 1, pp. 61–73, 2018.

[16] Jinquan Dai, Jie Huang, Shengsheng Huang, Bo Huang, and Yan Liu. 2011. HiTune: dataflow-based performance analysis for big data cloud. In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11). USENIX Association, Berkeley, CA, USA, 7-7.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley et al., "Resilient Distributed Datasets:A Fault-tolerant Abstraction for In-memory Cluster Computing,"in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'12. Berkeley, CA,USA: USENIX Association, 2012, pp. 2–2.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, andG. Fox, "Twister: a runtime for iterative mapreduce," in Proceedings of the 19th ACM international symposium on high performance distributed computing. ACM, 2010, pp. 810–818.

[19] M. Isard, M. Budui, Y. Yu, A. Birrell, and D Fetterly, "Dryad:distributed data-parallel programs form sequential building blocks," in ACM SIGOPS operating systems review, vil. 41, no. 3. ACM, 2007,pp. 59-72.

[20] Wozniak, J.M., Armstrong, T.G., Maheshwari, K., Lusk, E.L., Katz, D.S., Wilde, M., & Foster, I.T. (2013). Turbine: A Distributed-memory Dataflow Engine for High Performance Many-task Applications. *Fundam. Inform., 128*, 337-366.

[21] Tom White. 2009. Hadoop: The Definitive Guide (1st ed.). O'Reilly Media, Inc..

[22] Twister2 Big Data Hosting Environment: A composable framework for high-performance data analytics. [Online]. Available: https://github.com/DSC-SPIDAL/twister2