Twister:Net - Communication Library for Big Data Processing in HPC and Cloud Environments

Supun Kamburugamuve*, Pulasthi Wickramasinghe*, Kannan Govindarajan*, Ahmet Uyar*,
Gurhan Gunduz*, Vibhatha Abeykoon*, Geoffrey Fox*
*School of Informatics, Computing, and Engineering
Indiana University Bloomington
{skamburu, pwickra, auyar, ggunduz, vlabeyko, gcf}@indiana.edu

***Abstract*—Streaming processing and batch data processing are the dominant forms of big data analytics today, with numerous systems such as Hadoop, Spark, and Heron designed to process the ever-increasing explosion of data. Generally, these systems are developed as single projects with aspects such as communication, task management, and data management integrated together. By contrast, we take a component-based approach to big data by developing the essential features of a big data system as independent components with polymorphic implementations to support different requirements. Cnsequently, we recognize the requirements of both dataflow used in popular Apache Systems and the Bulk Synchronous Processing communication style common in High-Performance Computing(HPC) for different applications. Message passing interface implementations are dominant in HPC but there are no such standard libraries available for big data. Twister:Net is a stand-alone, highly optimized dataflow style parallel communication library which can be used by big data systems or advanced users. Twister:Net can work both in cloud environments using TCP or HPC environments using Message Passing Interface implementations. This paper introduces Twister:Net and compares it with existing systems to highlight its design and performance.**

I. Introduction

A great many big data systems exist today for the purpose of processing the enormous wealth of data available in terms of velocity, volume, and veracity. Streaming processing and batch data processing are the dominant forms of big data analytics, with Function as a Service (FaaS) emerging as a new paradigm. Systems such as Spark [1] and Hadoop primarily focus on batch data, while Heron [2], Flink [3] and Storm target streaming data. As opposed to these systems, the high performance computing (HPC) community uses Message Passing Interface (MPI) and its implementations as their framework of choice for large-scale parallel applications.

From an execution perspective, we can identify four key aspects of a parallel program whether it is designed for data processing or HPC: 1. Acquiring computing resources, 2. Spawning processes/threads and managing them on the allocated resources to execute the user program, 3. Communication layer between the parallel processes, and 4. Managing the data including both static and intermediate data. The HPC community has developed different technologies to abstract out these layers including resource schedulers such as Slurm [4], MPI and OpenMP [5] for process and thread management, communication using MPI, and in-memory distributed data management using PGAS [6]. These systems are mostly independent and allow a user to pick and choose depending on application requirements. For example, one can use only MPI where the application manages both threads and data of the program. In another setting, MPI plus OpenMP can be used where OpenMP manages the threads within an MPI process.

The big data systems are mostly designed in a monolithic approach with the above mentioned functions developed in a single project with tight integration between them. With the advent of Mesos [7] and Yarn [8], the resource scheduling layer is being separated from most big data systems, but other layers mostly remain within the same framework. Such designs make it harder to evolve functionality independently and adhere to standards. Also we note that these systems are designed with assumptions at different components making them suitable for limited set of applications [9].

MPI is the dominant standard for HPC applications whereas big data adopted the dataflow model. Every parallel program including big data applications and HPC applications can be modeled as a graph with nodes doing computations and edges representing the communication. Dataflow programming model has become popular in data analytics due to its simplicity and ease of use. With this model, big data frameworks represent a computation as a generic graph where nodes of the graph can be executed on different machines depending on the requirements of the application. This generic graph structure adopted by big data systems allows one to model both streaming and batch applications.

Big data systems do not view parallel operations in terms of communications but rather as higher level API's on data sets. By doing so they can hide the communications under higher level APIs such as task and data transformation APIs. As a result, such communications are in general loosely defined by separate implementations without a standard specification. Even though every big data framework is designed according to the same dataflow model, different systems have their own primitives with slightly varying semantics depending on the implementations. The communication implementations found in the current frameworks are tightly integrated and not exposed as APIs to be used as libraries without relying on the entire framework. In this paper authors recognize the need to have a separate communication library supporting dataflow

style big data communications.

Twister2 [10], [9] is our component-based approach to big data whereby the authors are developing the four essential abstractions required by a parallel program independently of each other. Twister:Net is the communication component of Twister2, which has been developed as a stand-alone library for big data applications supporting dataflow communications. Bulk synchronous parallel(BSP) communication as implemented by MPI can be used in Twister2 as well. Twister2 plans to incorporate Harp [11], which is a BSP implementation for big data systems. The dataflow communications in Twister:Net are implemented on top of TCP and MPI communications, allowing it to be deployed in both HPC and cloud environments. In this paper we present, dataflow communications of Twister:Net and compare it to the existing big data frameworks in order to show it can achieve equivalent performance or better with existing frameworks. Also, we look at MPI communications and big data requirements and compare different applications in those settings. The contributions of this paper are:

1) Define a dataflow communication model for big data to include both streaming and batch data
2) Present Twister:Net as an implementation of this model and show that it can achieve better or equal performance compared to streaming and batch systems

The rest of the paper is organized as follows. Section II describes dataflow and communication requirements for big data. Section III explains the Twister:Net library. The experiments conducted to evaluate the system and their results are described in Section IV. We conclude the paper with related work and future work in sections V and VI respectively.

II. Dataflow for Big Data

Dataflow has been recognized and accepted as the preferred mechanism for processing large data sets. A dataflow program models a computation as a graph with nodes of the graph doing user-defined computations and edges representing the communication links between the nodes. The data flowing through this graph is termed as events or messages. It is important to note that even though by definition dataflow programming means data is flowing through a graph, it may not necessarily be the case physically, especially in batch applications. Big data systems employ different APIs for creating the dataflow graph. For example, Flink and Spark provide distributed data set-based APIs for creating the graph while systems such as Storm and Hadoop provide task-level APIs.

For a dataflow program (DFP) or a bulk synchronous program (BSP) executing in parallel, peer-to-peer communications and collective communications are used for sharing data between parallel tasks. Collective communications define data transfer between a set of tasks in contrast to one-to-one communications as in the case of peer-to-peer. The collective communication patterns as identified by the parallel computing communities are available through MPI [12] implementations. Heavily used collective operations include Broadcast, Scatter, Gather, Reduce, AllGather, and AllReduce [13]. The parallel computing community found that these communications can be optimized for latency and throughput using special algorithms that send the messages among the tasks and determine the routing of messages. These algorithms are termed collective algorithms and are available through MPI implementations.

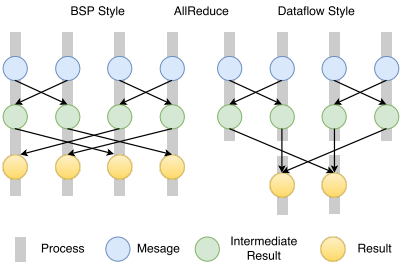

Fig. 1. Dataflow vs. BSP graph structure

MPI has been the standard communication API for high performance computing for the last two decades. It boasts a solid API with a mathematical foundation to support highly scalable parallel applications. There are many implementations of the MPI standard available and the mainstream MPI implementations enable applications to scale to hundreds of thousands of cores due to their superior collective communication algorithms, efficient use of memory and support of different networking hardware.

The graph structure of a parallel program is defined by the communications and execution of tasks. One can build every parallel execution graph using basic communication operations such as send and receive. When developing higher level communication patterns as in collective communications, a certain structure of the graph can be assumed in order to make the communication operations efficient. MPI collective operations assume that the tasks producing the data and the receiving tasks are in the same communicator. Big data relaxes these requirements and allows collective operations between any set of tasks.

A dataflow communication pattern defines the edges of an execution graph. For instance, a single node can broadcast a message to multiple nodes in the graph. This means that there is an edge from a source node to every receiving node. Reduce is the opposite of a Broadcast operation where multiple nodes link to a single node. Apart from these operations, Gather, Join and Union are used extensively. Each of these operations can accept keys and group messages and they are termed keyed operations.

A. Dataflow Communication Requirements

Batch processing deals with complete data sets as opposed to partial data sets as in stream processing. Stream processing does not necessarily imply real-time data analytics and can work on stored data. Batch data processing pipelines for big data are executed stage by stage where whole cluster resources are used by one part of the computation graph at any given

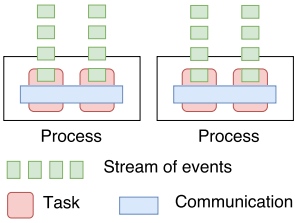

Fig. 2. Model of a communication operator

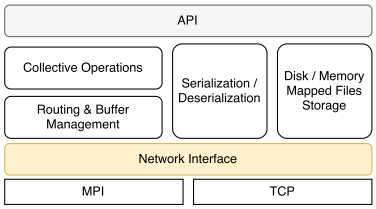

Fig. 3. Twister:Net Architecture

time. On the other hand, stream processing executes every part of the graph on a stream of events as needed. To facilitate the execution of the complete graph, different parts of the dataflow graph need to be deployed across machines and the communications are done from one set of tasks to another.

A big data application requires the data to be partitioned in a hierarchical manner due to memory limitations. In general data is first partitioned according to the number of parallel tasks and then each partition is again split into smaller partitions. This hierarchical approach is implicit in streaming applications, as only a small portion of the data is available at any given time.

Because of the data locality and processing requirements, the computation graph of a dataflow program can contain a different number of parallel tasks at different stages. Furthermore, streaming applications require the nodes of the graph to be deployed in different CPUs in order to handle a higher rate of messages. Big data applications mostly deal with unstructured data like text records. These records do not have specific data sizes defined. Because of this, the communication operations cannot assume the data sizes across the participating tasks. When transferring data, keys are used to group the data.

The communication requirements of dataflow programs for big data are summarized below. Twister:Net is designed according to these requirements.

1) The communications are between a set of tasks in an arbitrary task graph.
2) Handle communications larger than available memory.
3) Dynamic data sizes across communicating tasks.
4) Batch is modeled as a special case of streaming.
5) Keys are part of the abstraction.

III. Twister2:Net

The abstractions of Twister2 include: 1. Data Access, 2. Resource, 3. Communication, 4. Task, 5. Data Management. Each layer is generic and can have multiple implementations

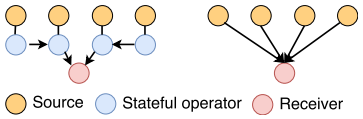

Fig. 4. Reduce Operation as a Graph Extension, **left:** Optimized graph **right:** Default graph

to support the needs of specific applications. For example, Twister2 supports different resource schedulers including HPC schedulers like Slurm and big data schedulers such as Aurora [14], Mesos [7] and Kurbenetes [15].

Twister:Net is an open source library [1] implemented using Java language. Its base operators are non-blocking and one can build blocking communications by using the non-blocking semantics. The architecture of Twister:Net is shown in Figure 3. The bottom layer is the network access API, which allows one to plug in different networking providers such as OpenMPI [16] and TCP. This layer assumes a reliable channel by the underlying provider. The MPI implementation uses the ISend/IRecv operations to build the dataflow operations and by default we use OpenMPI [17]. TCP implementation uses Java NIO and creates MPI ISend/IRecv semantics to be used by Twister:Net.

On top of this layer the buffer management and data serialization are implemented. The data serialization frameworks are pluggable and we use Kryo as the default serializer for Java objects. Also various communication operations are built on these layers. The disk-based storage can be used by operations when they need to transfer data between disk and memory in order to handle larger data transfers. The top level provides the API layer for the base abstractions.

A. Communication Model

The dataflow communications are modeled as an extension to the dataflow graph. The generic form of a dataflow communication as modeled by Twister:Net is as follows.

```
Operation(S, D, E, M, T,
                  Optional Op..., C)
KeyedOperation(S, D, E, M,
                  T, K, KT,
                  Optional Op..., C)
S = Source tasks, D = Destination tasks,
E = Edges, M = Message, T = Data Type,
Op = Optional Stateful operators
C = Callback
K = Key
KT = Key data type
```

For dataflow style collectives, the source tasks and destinations tasks can be mutually exclusive. Each operation needs an edge identifier (Integer) to distinguish from other operators happening simultaneously. Some operators may require more than one edge number. We support Java Objects and primitive array types as message types and will expand to other formats such as protocol buffers. If keyed operations are used, each message can have a key.

[1] https://github.com/DSC-SPIDAL/twister2

The communication model of Twister:Net is shown in Fig. 2. Each operation created in a single process can be shared by multiple tasks in that process and can accept a stream of messages. For streaming cases, the stream is unbounded, while for batch cases bounded stream is assumed with the last message marked as the end of the stream. For the streaming mode of communications, the operators do not buffer data and forward them accordingly. For batch operations, the operators can buffer data to increase the throughput. With batch communication a special flag is used by sources to signal the end of flow.

1) Streaming mode - Each operation only considers a single datum as an entity and an infinite stream of events
2) Batch mode - Multiple data items are considered into a single operation with a finite stream of events

Furthermore, each of these communication modes can operate with keys. If required, a communication can use the disks in order to handle data sets that do not fit the available memory.

Twister:Net has implemented the following dataflow collective operations; 1. Reduce, 2. AllReduce, 3. KeyedReduce, 4. Gather, 5. AllGather, 6. KeyedGather, 7. Partition, and 8. Broadcast. Fig. 4 shows the dataflow graph of reduce and how it is optimized using a tree structure and stateful operators. For operations like Reduce and Gather messages flow through this optimized graph. We implemented the AllReduce operation as a reduce + broadcast and AllGather operation as gather + broadcast. For keyed operations, we create an optimized routing to each destinations of the operation. Messages are routed according to the correct destination using these underlying structures. For example for a keyed reduce, we create multiple trees each pointing to a single destination.

Stateful operations Communication operators can keep state about streams of messages passing through them. For example with batch operations, messages can be gathered and presented once the operation completes. For example such stateful operators can be used to create a combiner for Hadoop like gathers.

Thread Safety Different threads can progress the communication as well as perform message packing and callback handling. Depending on the underlying channel implementation, the network side of the communication can be thread-safe or not. For example, MPI can be compiled with different levels of thread safety, and if such a compilation is used the channel can be thread-safe.

Dynamic messages Unlike in MPI programs where mostly structured data is used, the dataflow communications deal with unstructured data such as text records. With MPI a user has to take additional steps like knowing the data sizes (which may require additional operations), serialization before using operations like Reduce, Gather on such data. Twister:Net by default support such data and hides the details from the user. Because of this realization, the framework has to allocate memory to receive the incoming messages as user is not aware of the size of the messages the operation receives.

Buffer Management & Back Pressure The buffers are managed internally by the framework. When a higher level message is submitted, it is serialized and put into an available buffer. For parallel operations, it is important to balance the communications such that one source cannot overproduce data. At each stage of the communication, the library buffers the data up to a configurable amount. Once these buffers are filled the operations will not accept messages from sources or the network. Buffering can be used to increase the throughput of the operations at the expense of latency. Because of the relaxed constraints of the message sizes, fixed size buffers are employed by sending and receiving sides. If a submitted message does not fit the buffer size used, it is packed into multiple buffers before being sent out. The framework deploys a message length-based protocol to unpack such messages at the receiver.

Initialization Since we are targeting for deployment on different environments, Twister:Net can use a pluggable architecture to bootstrap. When used with MPI, the MPI handles the connection management and the framework delegates to MPI. For TCP and other potential transports, a TCP-based bootstrapping can be used. Every process needs to know a master TCP process and its address. This can be a client that submits the job or a master process of the job. The workers of the communication use a simple handshake protocol with the master to send it port numbers which the master distributes among the workers in order for them to know the port numbers and network addresses and thus establish the communication.

Key A message can contain a key of any data type supported by Twister:Net. When keys are used, an actual message will contain the key and the content as separate bytes. Each message contains the overall length of the message and the key length if a variable length key is used. If a fixed size key such as a primitive type is used, the key length is not included in the message.

B. Communication Spilling to Disk

Big data and HPC applications deal with large data sets that sometimes cannot be processed and analyzed in-memory even with large clusters. As a solution, Twister:Net sends data to disk when a configurable amount of in-memory messages are accumulated. We implemented direct file based version and memory mapped files based version using LmdbJava[18] for achiving this feature. When messages are received, they are put into a queue and when this queue becomes full, it is saved in to the disk. If key based communication is used, the values are sorted according to keys before saving to disk. Each such buffer is saved to a separate file. Sorting and saving is done by a separate thread. After all the data is received, the saved values are retrieved from the the disk or memory mapped files and served to the user operator. For example, a gather operation would collect all the results sent from participating tasks and store them. Further we found that the LmdbJava implementation doesn't scale to very large data sets in the gigabytes range and used large amounts of memory. Filesystem based approach performed better for some such large operations.

IV. Evaluation

We conducted several micro-benchmarks and developed two applications to compare the performance of Twister:Net with existing frameworks. We compared Twister:Net performance with Apache Spark, Flink, and Heron in different situations as they are designed to process specific workloads. In results DFW and BSP indicates dataflow results and MPI respectively.

The experiments were conducted in two clusters. One cluster had 16 nodes of Intel Platinum processors with 48 cores in each node and 56Gbps Infiniband and 10Gbps network connections. The other cluster had Intel Haswell processors with 24 cores in each node with 128GB memory, 56Gbps Infiniband, and 1Gpbs Ethernet connections. 32 nodes of this cluster were used for the experiments.

A. Micro-benchmarks

Fig. 5 and Fig. 6 show the bandwidth utilization and latency of Twister:Net in a two-node setting where one task sends messages to another task in a different node. Even though we use MPI underneath with other layers added, the latency shows that the overhead is minimal. The bandwidth utilization of Twister:Net is less compared to MPI because a new byte buffer is created for each receiving message. For the MPI test, we did not create a new buffer for every message received, thus increasing the bandwidth.

1) Streaming Benchmarks: We have conducted several micro-benchmarks with Apache Heron streaming engine to observe how Twister2 performs in a streaming setting. The first experiment used a data re-balance communication where a set of N tasks distributed the data received among another set of N tasks. In the next benchmark, we used a more involved reduce communication where a set of tasks sends messages to a single task in a reduce operation. For the last benchmark, the broadcast operation is used. A feedback loop is established from destination tasks to the source tasks to control the flow of the data and to facilitate the latency measurements. Without such a loop, it is harder to measure latency as the tasks are deployed on different machines. The experiments were conducted on 16 nodes with 256 parallel tasks on the Haswell cluster. The feedback loop carries a constant size message with the original message ID.

Fig. 7 shows that Twister:Net communication times are well below those of Heron. There are many reasons to explain these results: 1. Heron does not implement optimized communication algorithms, 2. Heron uses stream managers as message routers, meaning the messages generated by parallel instances in a single node go through a single process, 3. It uses both protocol message serializations and Kryo-based object serialization for a single message.

2) MPI Benchmarks: We implemented experiments with OpenMPI Java binding to compare Twister:Net performance with direct OpenMPI performance. The experiments were conducted in order to observe whether there are any drastic performance drops compared to OpenMPI operations in a one-to-one mapping setting. We used two tests with one using Reduce operation and another with Gather operation. In one test we use a fixed size integer message while for other we simulate and dynamic object with different sizes. The results show that Twister:Net Reduce operation is slightly slower than that of MPI for fixed size messages. The gather operation of Twister:Net is slower than MPI due to its relaxed buffer management compared to MPI. As a first implementation we believe these are acceptable results as big data systems perform much slower.

3) Flink Benchmarks: We compared the performance of Twister:Net with Flink for measuring the total time. Fig. 8 shows the results of two streaming operations with Flink and Twister2. This test was conducted on 32 nodes with 640 parallelism. For Reduce operation, each parallel task generates 1000 messages, and for Partition operation each task generates 1 million messages. The total time to finish these messages was measured. The results show that the partition operation of Flink has equal performance to Twister:Net for Ethernet but Twister:Net Reduce operation has far superior performance. Flink doesn't implement optimized reduce operations as in this paper hence the performance is less.

B. Applications

1) Terasort: Terasort is a popular benchmark to measure the performance of data processing systems. With Terasort, the data is partitioned into equal-sized chunks so that each task in the job gets the same amount of data. Next, the algorithm collects a set of sample records per task and sorts this sample set to determine an ordered partitioning for the complete data set. In the third step, the generated global partitioning is used to send records to the correct task. This phase is generally known as a shuffling phase. Finally, after all the records are collected, each task will sort the local set of records and write the results to disk. This will result in a globally sorted data set across all the tasks. The parallel version of Terasort is described in Algorithm [19]. The authors detailed and discussed Terasort implementations with Spark, Flink, and MPI in [19].

The Twister2 implementation of Terasort uses Gather, Broadcast, and Partition operations. Initially a task reads the data partition assigned to it. Then a Gather operation is performed to collect the sample set of records from every task. This collected sample data set is used to create the ordered partitioning, which is a set of keys of size N-1 where N is the number of tasks. Afterwards, this key set is broadcast to all tasks. In order to shuffle the data to the correct tasks, the Partition operation is used. The data is read part by part from disk and send over the dataflow operation. The receiving records are sorted by the framework using the disks. After all the records are received, every task reads the records and writes the results to the disk. Fig. 11 shows the total time of Terasort on 16 nodes with BSP-style implementation, dataflow implementation and Flink. The BSP implementation was performing slightly better than the DFW implementation due to the algorithm used to shuffle the data, which is a rotating shuffle algorithm [19]. Also it shows the running time of terasort on a 32 node cluster with 1 terabytes and

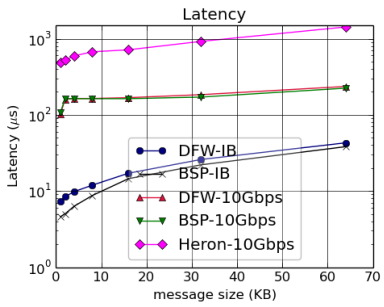

Fig. 5. Latency of MPI and Twister:Net with different message sizes on a two-node setup

500 Gigabytes data sets and the dataflow implementation performed equally well compared to BSP implementation.

2) K-means: We implemented the K-means algorithm using Twister:Net dataflow style and compared its performance with a K-means implementation of Spark and a BSP-style MPI implementation. We used the K-means algorithm implemented in Mllib of Spark 2.1.3. The dataflow style uses a task which is invoked by the threads of the process. This tasks computes and outputs the centers which are sent using the AllReduce operation. Once the results of the AllReduce operation available the thread updates the new centers of the task and executes the task again until the given number of iterations are reached.

Fig 10 shows the total time for running K-means on a 16-node cluster with Spark, Twister:Net dataflow and BSP-style operations with IB and 10Gbps networks. We use 320 parallel tasks for execution. The tests were conducted with a data set of 2 million points with 2 features and varying number of centers. The algorithm ran for 100 iterations for each test. The dataflow style program is written using Twister:Net performed equal to the BSP-style K-means program. Spark did not perform well for this algorithm because of the rapid creation of tasks for each iteration and the heightened communications.

V. Related Work

Message Passing Interface (MPI) [20] is primarily responsible for addressing messaging in parallel computing. It mainly supports two types of communications, point-to-point and collective communication. The traditional reduce/aggregate communication pattern in Spark [21] sends all the partitions reduced data values into the driver program. To reduce such bottlenecks, Apache Spark 1.1 [22] has introduced new communication patterns named TreeReduce and TreeAggregate which are based on the multi-level aggregation tree technique. In this technique, the data partitions are combined into a small set of executors in a partial manner before they are sent to the driver program, which reduces the load of the driver program and improves the performance.

Apache Flink [3] follows the data-streaming paradigm, thereby providing a unified architecture for both batch and streaming processing in the programming model and execution engine. In Flink, the streams distribute the data based on the various communication patterns, namely point-to-point, broadcast, re-partition, fan-out and merge. In Storm and Heron [23], a spout is a source of input data streams for the topology and a bolt is a component which processes those topologies. The stream grouping is an important concept which defines the method to partition the data stream into bolt tasks. It consists of eight built-in custom stream grouping concepts, namely shuffle grouping, field grouping, partial key grouping, all grouping, global grouping, none grouping (similar to shuffle grouping), direct grouping, and local or shuffle grouping. Harp [11] is a framework which is mainly designed to run big data analytic algorithms on High Performance Computing architectures. It is comprised of two main layers, computation and communication. The communication library is implemented similar to MPI collective communication operations which are highly optimized for big data analytics and machine learning algorithms. COMPS [24] is a task-based environment for Spark-like applications in HPC. Authors looked at performance of MPI, Spark and Flink for machine learning algorithms previously [19] and found MPI outperforms Spark and Flink for complex algorithms.

MRNet [25] is a software-based reduction network specifically designed for scalable tools to achieve scalable performance and multicast support. It uses a communicator for representing groups of network points. Similar to MPI, MRNet provides the support for point-to-point and multicast or broadcast communications. DataMPI [26] communication library is an extended MPI specification to achieve Hadoop-like communications. Currently it only supports single program and multiple data (Common), but they intended to support MapReduce, Streaming, and Iterations in the future. Twister:Net is going beyond Hadoop-like communications to a more general dataflow-style job communication which supports both batch and streaming process. There are many ongoing efforts to incorporate HPC features into existing big data frameworks, including RDMA support for frameworks such as Spark [27], Hadoop [28], and Apache Heron [29]. The authors previously worked on improving Apache Storm's performance using collective algorithms [30]. MPIgnite is an effort to bring BSP-style communication into Spark [31]. These efforts are targeted towards individual frameworks, and Twister:Net is a generic framework that models big data communications in a generic fashion.

VI. Conclusions & Future Work

Twister:Net is an optimized big data communication library for both cloud and HPC technologies. With Twister:Net we acknowledge the need to have different types of communications for different applications, particularly streaming, data pipelines and complex algorithms including machine learning. In addition, Twister:Net defines the communication requirements of big data in a separate library without integrating to any particular big data framework. The results indicate that by using Twister:Net as a pure communication library, one can build highly efficient applications. In Twister:Net, we do not

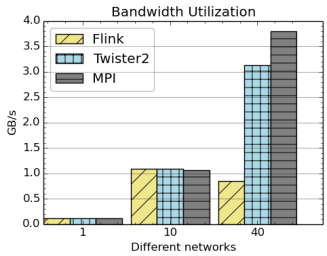

Fig. 6. Bandwidth utilization of Flink, Twister2 and OpenMPI over 1Gbps, 10Gbps and IB

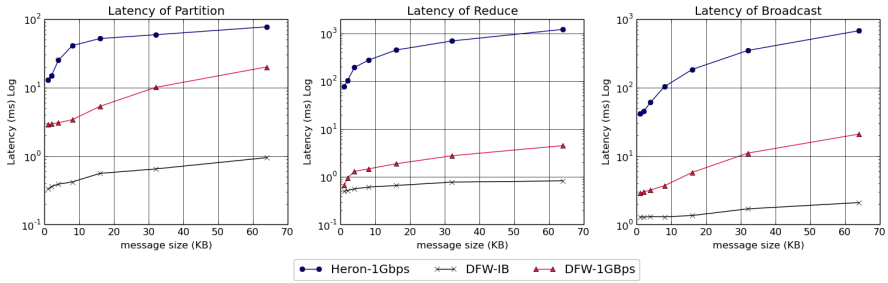

Fig. 7. Latency of Heron and Twister:Net for Reduce, Broadcast and Partition operations in 16 nodes with 256-way parallelism

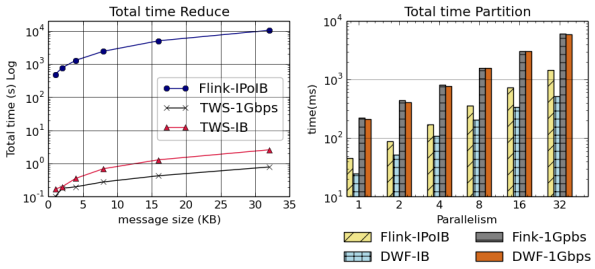

Fig. 8. Total time for Flink and Twister:Net for Reduce and Partition operations in 32 nodes with 640-way parallelism. The time is for 1 million messages in each parallel unit, with the given message size

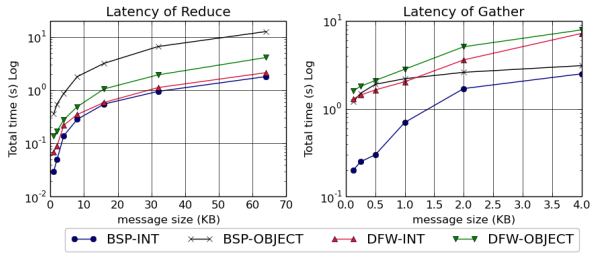

Fig. 9. Latency for OpenMPI and Twister:Net for Reduce and Gather operations in 32 nodes with 256-way parallelism. The time is for 1 million messages in each parallel unit, with the given message size

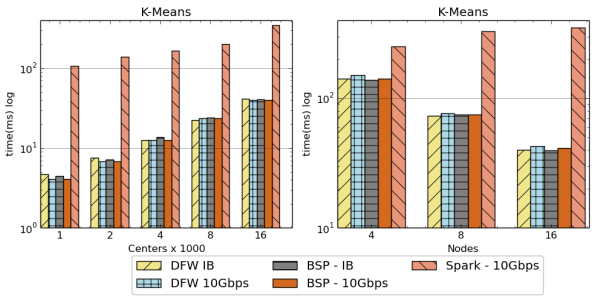

Fig. 10. Left: K-means job execution time on 16 nodes with varying centers, 2 million points with 320-way parallelism. Right: K-Means wth 4,8 and 16 nodes where each node having 20 tasks. 2 million points with 16000 centers used.

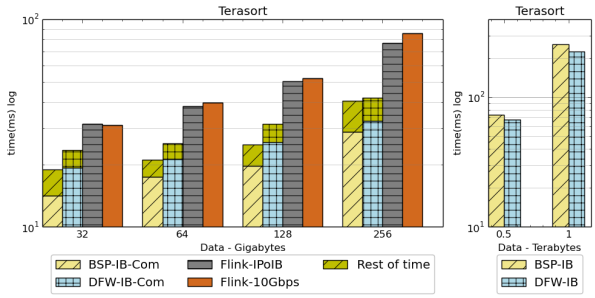

Fig. 11. Left: Terasort time on a 16 node cluster with 384 parallelism. BSP and DFW shows the communication time. Right: Terasort on 32 nodes with .5 TB and 1TB datasets. Parallelism of 320.

consider fault tolerance and leave that to the frameworks using the communication library.

We are actively working on the rest of the components of the Twister2 big data stack including task scheduling/management, dataflow graphs, fault tolerance, and data abstraction layers. Combining these with the communication libraries presented here could offer the functionality of current big data systems for batch and streaming but with HPC performance as shown in this paper. The communication library can be integrated to other big data systems such as Heron and Spark.

Operations such as reduction require information inside a complex message. This can be costly for high level object-based messages where we need to de-serialize the complete message. Having an option to look at only the parts of the messages without de-serialization can decrease the latency and increase the throughput. It is possible to build such a framework with binary protocols like Google Protocol Buffers. The current big data communications and operations do not have standard semantics available, meaning different frameworks have slightly different APIs. It would be better to have standard APIs so that users can work with uniform APIs.

We are working on incorporating more collective algorithms into the library that can perform well in different circumstances, like throughput critical applications. At the moment Twister:Net supports only MPI-based and TCP-based communications, but we intend to include further networks. There are other dataflow operations such as Unions and Joins that we are working to integrate with the library. Twister:Net relies on flow control at the network layer for handling back-pressure

at the application level. When multiple communications share the same underlying network channel between nodes, one operation can slow down the others. The communication library needs a back-pressure mechanism as an add-on feature that can be used as required. We would like to integrate the dataflow-style communications directly into an MPI implementation like OpenMPI. Also having the communication implemented directly on top of an RDMA library such as Photon [32] would be useful.

ACKNOWLEDGMENTS

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, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[2] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[3] 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.

[4] A. B. Yoo, M. A. Jette, and M. Grondona, ""SLURM: Simple Linux Utility for Resource Management",," D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds.

[5] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[6] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065950

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[8] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[9] "Twister2:Design of aBig Data Toolkit," 2017, Technical Report. [Online]. Available: http://dsc.soic.indiana.edu/publications/Twister2.pdf

[10] G. Fox, "Components and rationale of a big data toolkit spanning hpc, grid, edge and cloud computing," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/3147213.3155012

[11] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective communication on hadoop," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 228–233.

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2004, pp. 97–104.

[13] R. Thakur and W. D. Gropp, *Improving the Performance of Collective Operations in MPICH*.

[14] R. DelValle, G. Rattihalli, A. Beltre, M. Govindaraju, and M. J. Lewis, "Exploring the Design Space for Optimizations with Apache Aurora and Mesos," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 537–544.

[15] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[16] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104.

[17] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A High-Performance, Heterogeneous MPI," in *2006 IEEE International Conference on Cluster Computing*, Sept 2006, pp. 1–9.

[18] LMDB Java. [Online]. Available: https://github.com/lmdbjava/lmdbjava/

[19] 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.

[20] "MPI: A Message-Passing Interface Standard Version 3.0," 2012, Technical Report. [Online]. Available: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[21] Spark Communication. Accessed: March 02 2018. [Online]. Available: https://www.gitbook.com/book/umbertogriffo/apache-spark-best-practices-and-tuning/details/

[22] Spark Improvements 1.1. Accessed: March 02 2018. [Online]. Available: https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html/

[23] Storm and Heron Communication Concepts. Accessed: March 02 2018. [Online]. Available: http://storm.apache.org/releases/2.0.0-SNAPSHOT/Concepts.html/

[24] J. Conejero, S. Corella, R. M. Badia, and J. Labarta, "Task-based programming in COMPSs to converge from HPC to big data," *The International Journal of High Performance Computing Applications*, vol. 0, no. 0, p. 1094342017701278, 0. [Online]. Available: http://dx.doi.org/10.1177/1094342017701278

[25] P. C. Roth, D. C. Arnold, and B. P. Miller, "Mrnet: A software-based multicast/reduction network for scalable tools," in *Supercomputing, 2003 ACM/IEEE Conference*, Nov 2003, pp. 21–21.

[26] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: Extending mpi to hadoop-like big data computing," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, p. 829.

[27] X. Lu, D. Shankar, S. Gugnani, and D. K. D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 253–262.

[28] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 641–650.

[29] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox, "Low Latency Stream Processing: Apache Heron with Infiniband & Intel Omni-Path," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/3147213.3147232

[30] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox, "Towards High Performance Processing of Streaming Data in Large Data Centers," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1637–1644.

[31] B. L. Morris and A. Skjellum, "Mpignite: An mpi-like language and prototype implementation for apache spark," *CoRR*, vol. abs/1707.04788, 2017. [Online]. Available: http://arxiv.org/abs/1707.04788

[32] E. Kissel and M. Swany, "Photon: Remote memory access middleware for high-performance runtime systems," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1736–1743.