Low Latency Stream Processing: Twitter Heron with Infiniband and Omni-Path

Supun Kamburugamuve[1] Karthik Ramasamy[2], Martin Swany[1] and Geoffrey Fox[1]
[1]*School of Informatics and Computing, Indiana University Bloomington* [2]*Twitter Inc.*

Submission Type: Research

Abstract

Worldwide data production is increasing both in volume and velocity, and with this acceleration, data needs to be processed in streaming settings as opposed to the traditional store and process model. Distributed streaming frameworks are designed to process such data in real time with reasonable time constraints. Twitter Heron is a production ready large scale distributed stream processing framework developed at Twitter. In order to scale streaming applications to large numbers of nodes, the network is of utmost importance. High performance computing (HPC) clusters feature interconnects that can perform at higher levels than traditional Ethernet. In this work the authors present their findings on integrating Twitter Heron distributed stream processing system with two high performance interconnects; Infiniband and Intel Omni-Path.

1 Introduction

With ever increasing data production by users and machines alike, the amount of data that needs to be processed has increased dramatically. This must be achieved both in real time and as batches to satisfy different use cases. Additionally, with the adoption of devices into Internet of Things setups, the amount of real time data are exploding, and must be processed with reasonable time constraints. In real time distributed stream analytics, the large data streams are partitioned and processed in distributed sets of machines to keep up with the high volume data rates. By definition of large-scale streaming data processing, networks are a crucial component in transmitting messages between the processing units for achieving efficient data processing.

High performance computing (HPC) clusters are designed to perform large computations with advanced processors, memory, IO systems and high performance interconnects. For efficient parallel computations at very large scale, the network between the nodes is vital in order to scale the applications to thousands of nodes. These high performance interconnects feature microsecond latencies and large bandwidths. Thanks to recent advancements in hardware, some of these high performance networks have become cheaper to set up than their Ethernet counterparts. With multi-core and many-core systems having large numbers of CPUs in a single node, the demand for high performance networking is increasing exponentially as well.

There are many distributed streaming frameworks available today for processing large amounts of streaming data in real time. Such systems are largely designed and optimized for commodity hardware and clouds. Apache Storm [1] was one of the most popular early systems developed for processing streaming data. Twitter Heron [2] is a next-generation version of Storm with an improved architecture for large-scale data processing. It features a hybrid design with some of the performance-critical parts written in C++ and others written in Java. This architecture allows the integration of high performance enhancements naively rather than going through native wrappers such as JNI. This work take advantage of these architectural enhancements to integrate high performance interconnects naively to Heron to accelerate its communications.

Advanced hardware features are seldom used in the Big Data computing frameworks, mostly because they are accessible only to low level programming languages and most Big Data systems are written on Java platform. Infiniband [3] is an open standard protocol for high performance interconnects that is widely used in today's high performance clusters. Omni-Path [4] is a proprietary interconnect developed by Intel and is available with the latest Knights Landing architecture-based many-core processors.

In this work the authors explore how to leverage these hardware features in the context of distributed streaming framework Twitter Heron. In particular the authors show how they leveraged the Infiniband and Intel Omni-Path communications to improve the latency of the system. The main contribution in this work is to showcase the importance of using high performance interconnects for distributed stream processing. There are many differences in hardware available for communications with different bandwidths, latencies and processing models. Even Ethernet has comparable hardware available to some of the high performance interconnects; it is not our goal to show that one particular technology is superior to others, as different environments may have alternate sets of these technologies and with these implementation Heron can take advantage of such environments.

The remainder of the paper is organized as follows. Section 2 presents the background information on Infiniband and Omni-Path. Section 3 describes the Heron architecture in detail and section 4 the implementation details. Next the experiments conducted are described in sections 5 and results are presented and discussed in section 6. Section 7 presents related work. The paper concludes with a look at future work.

2 Background

2.1 Infiniband

Infiniband is one of the most widely used high performance fabrics. It provides a variety of capabilities including message channel semantics, remote memory access and remote atomic memory operations, supporting both connection-oriented and connectionless endpoints. Infiniband is programmed using the Verbs API, which is available in all major platforms. The current hardware is capable of achieving up to 100Gbps speeds with microsecond latencies. Infiniband does not require the OS Kernel intervention to transfer packets from user space to the hardware. Unlike in TCP, its protocol aspects are handled by the hardware. These features mean less CPU time is required by the network compared to TCP for transferring the same amount of data. Because the OS Kernel is bypassed by the communications, the memory for transferring data has to be registered in the hardware.

2.2 Intel Omni-Path

Omni-Path is a high performance fabric developed by Intel. Omni-Path fabric is relatively new compared to Infiniband and there are fundamental differences between the two. Omni-Path does not offload the protocol handling to network hardware and it doesn't have the connection oriented channels as in Infiniband. Unlike in Infiniband the Omni-Path network chip can be built into the latest Intel Knights Landing processors. Omni-Path supports tagged messaging with a 96 bit tag in each message. A Tag can carry any type of data and this information can be used at the application to distinguish between different messages. Omni-Path is designed and optimized for small high frequency messaging.

2.3 Channel & Memory Semantics

High performance interconnects generally supports two modes of operations called channel and remote memory access. With channel semantics queues are used for message communication. In memory semantics a process can read from or write directly to the memory of a remote machine.

In channel mode, two queue pairs for transmission and receive operations are used. To transfer a message, a descriptor is posted to the transfer queue, which includes the address of the memory buffer to transfer. For receiving a message, a descriptor needs to be submitted along with a pre-allocated receive buffer. The user program queries the completion queue associated with a transmission or a receiving queue to determine the success or failure of a work request. Once a message arrives, the hardware puts the message into the posted receive buffer and the user program can determine this event through the completion queue. Note that this mode requires the receiving buffers to be pre-posted before the transmission can happen successfully.

The remote direct memory access mode is generally called RDMA. With RDMA, two processes preparing to communicate, register memory and share the details with the other party. Read and write operations are used instead of send and receive operations. These are one-sided and do not need any software intervention from the other side. If a process wishes to write to remote memory, it can post a write operation with the local addresses of the data. The completion of the write operation can be detected using the completion queue associated. The receiving side is not notified about the write operation and has to use out-of-band mechanisms to figure out the write. The same is true for remote reads as well.

2.4 Openfabrics API

Openfabrics [1] provides a library called libfabric that hides the details of common high performance fabric APIs behind a uniform API. Because of the advantage of such an API, we chose to use libfabric as our programming library for implementing the high performance communications for Heron. Libfabric is a thin wrapper API and it supports different providers including Verbs, Aries interconnect from Cray through GNI, Intel Omni-Path, and Sockets. The libfabcirc API closely resembles the structure of Verbs API. For example, for channel semantics, it provides send/receive operations with completion queues to get notifications about the operations.

[1]https://www.openfabrics.org/

2.5 TCP vs High performance Interconnects

TCP is one of the most successful protocols in the brief history of computers. It provides a simple yet powerful API for transferring data reliably across the Internet using unreliable links and protocols underneath. One of the biggest advantages of TCP is its wide adoption and simplicity to use. Virtually every computer has access to a TCP-capable adapter and the API is solid across different platforms. Even with these advantages, it has its own drawbacks, especially when the performance requirements are high.

TCP provides a streaming API for messaging where the fabric does not maintain message boundaries. The messages are written as a stream of bytes to the TCP and it is up to the application to define mechanisms such as placing markers in between messages to mark the boundaries of the messages. On the other hand, Infiniband and Omni-Path both support message boundaries for message transfers.

Most of the high performance interconnects have drivers that make them available through the TCP protocol stack. The biggest advantage of IPoIB is that an existing application written using the TCP stack can use high performance interconnect without any modifications to the code. It is worth noting that the native use of the interconnect through its API always yields better performance than using it through TCP/IP stack.

One of the biggest challenges to TCP comes from its buffer management and data copying. A typical TCP application allocates memory in user space and the TCP stack needs to copy data between user space and Kernel space. Also each TCP call involves a system call which does a context switch of the application. The flow control mechanism of the TCP can be a bottleneck for some latency sensitive applications as well. The high performance fabrics do not have these issues because they typically do not include system calls and the hardware is capable of copying data directly to user space buffers.

The protocol handling part of the TCP is executed by the host CPU, which allocates valuable resources for doing tasks that can be handled by hardware. Infinband takes this into consideration and offloads the protocol processing aspects to hardware while Omni-Path still involves the CPU for protocol processing.

3 Twitter Heron

Heron is a distributed stream processing framework developed at Twitter. It is open-sourced and available in github [2] for others to use. Heron is the successor to Apache Storm with many enhancements to the underlying engine architecture. It retains the same Storm API, allowing applications written in Storm to be deployed with no or minimal code changes.

[2]https://github.com/twitter/heron

3.1 Heron Data Model

A stream is an unbounded sequence of high level objects named events or messages. The streaming computation in Heron is referred to as a Topology. A topology is a graph of nodes and edges. The nodes represent the processing units executing the user defined code and the edges between the nodes indicate how the data (or stream) flows between them. There are two types of nodes: spouts and bolts. Spouts are the sources of streams. For example, a Kafka spout can read from a Kafka queue and emit it as a stream. Bolts consume messages from their input stream(s), apply its processing logic and emit new messages in their outgoing streams.

Heron has the concept of a user defined graph and an execution graph. The user defined graph defines how the processing units are connected together in terms of message distributions. On the other hand, the execution graph is the layout of this graph in actual nodes with network connections and computing resources allocated to the topology to execute. Nodes in the user graph can have multiple parallel instances (or tasks) running to scale the computations. The user defined graph and the execution graph are referred to as logical plan and physical plan respectively.

3.2 Heron Architecture

The components of the Heron architecture are shown in Fig. 1. Each Heron topology is a standalone long-running job that never terminates due to the unbounded nature of streams. Each topology is self contained and executes in a distributed sandbox environment in isolation without any interference from other topologies. A Heron topology consists of multiple containers allocated by the scheduler. These can be Linux containers, physical nodes or sandboxes created by the scheduler. The first container, referred to as the master, always runs the Topology Master that manages the topology. Each of the subsequent containers have the following processes: a set of processes executing the spout/bolt tasks of the topology called Heron instances, a process called a stream manager that manages the data routing and the connections to the outside containers, and a metrics manager to collect information about the instances running in that container.

Each Heron instance executes a single task of the topology. The instances are connected to the stream manager running inside the container through TCP loop-back connection. It is worth noting that Heron instances always

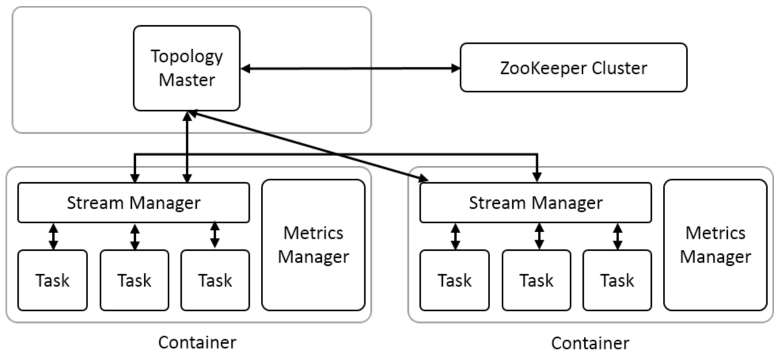


Figure 1: High level architecture of Heron. Each outer box shows a resource container allocated by a resource scheduler like Mesos or Slurm. The arrows show the communication links between different components.

connect to other instances through the stream manager and they do not communicate with each other directly even if they are on the same container.

The stream manager acts as a bridge between Heron instances. It forwards the messages to the correct instances by consulting the routing tables it maintains. A message between two instances in different containers goes through two stream managers. Containers can have many Heron instances running in them and they all communicate through the stream manager. Because of this design, it is important to have highly efficient data transfers at the stream manager to support the communication requirements of the instances.

Heron is designed from the ground up to be extensible, and most of the important parts of the core engine are written in C++ rather than JVM, the default language of choice for Big Data frameworks. The rationale for the use of C++ is to leverage the advanced features offered by the OS and hardware. Heron instances and schedulers are written in Java while stream manager and topology master are written in C++.

3.2.1 Acknowledgements

Heron uses an acknowledgement mechanism to provide at least once message processing semantics that ensures the message is always processed in the presence of process/machine failures. It is possible that the message can be processed more than once. In order to achieve at least once, the stream manager tracks the messages flowing through the system and if a failure occurs, it notifies the spout. When a bolt emits a message, it anchors the new message to the parent message and this information is sent to originating stream manager (in the same container) as a separate message. The new message is processed in the next bolt thereby generating another set of new messages and so on. One can view this as a tree of messages with the root being the anchored message. When every new message finishes its processing, a separate message is again sent to the originating stream manager. Upon receiving such control messages for every emit in the message tree, the stream manager marks the message as fully processed.

3.2.2 Processing pipeline

Heron has a concept called max messages pending with spouts. When a spout emits messages to a topology, this number dictates the amount of in-flight messages that are not fully processed yet. The spout is called to emit messages only when the current in-flight message count is less than the max spout pending messages.

3.3 Heron Stream Manager

Stream manager is responsible for routing messages between instances inside a container and across containers. It employs a single thread that use event-driven programming using non-blocking socket API. A stream manager receives messages from instances running in the same container and other stream managers. These messages are Google protocol buffer [5] serialized messages packed into binary form and transmitted through the wire. If a stream manager receives a message from a spout, it keeps track of the details of the message until all the acknowledgements are received from the message tree. Stream manager features a in-memory store and forward architecture for messages and can batch multiple messages into single message for efficient transfers. Because messages are temporarily stored, there is a draining function that drains the store at a user defined rate.

4 Implementation

Even though high performance interconnects are widely used by HPC applications and frameworks, they are seldom used in big data systems in a native fashion. Fur-

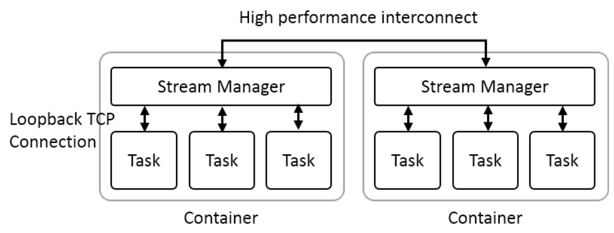


Figure 2: Heron high performance interconnects are between the stream managers

thermore, experiences in using these interconnects in big data systems are lacking in the public domain. In this implementation, Infiniband and Omni-Path interfaces with Heron through its stream manager, as shown in Fig. 2. Infiniband or Omni-Path message channels are created between each stream manager in the topology. These then carry the data messages going between the stream managers. The control messages that are sent between stream manager and topology master still use the TCP connections. They are not frequent and do not affect the performance of the data flow. The TCP loop-back connections from the instances to the stream manager are not altered in this implementation. Both Infiniband and Omni-Path implementations use channel semantics for communication. A separate thread is used for polling the completion queues associated with the channels. A credit based flow control mechanism is used for each channel along with a configurable buffer pool.

4.1 Bootstrapping

Infiniband and Omni-Path require information about the communication parties to be sent out-of-band through other mechanisms like TCP. Infiniband uses the RDMA(Remote direct memory access) Connection manager to transfer the required information and establish the connections. RDMA connection manager provides a socket-like API for connection management, which is exposed to the user in a similar fashion through Libfabric API. The connection manager also uses the IP over Infiniband network adaptor to discover and transfer the bootstrap information. Omni-Path has a built-in TCP server for discovering the endpoints. Because Omni-Path does not involve connection management, only the destination address is needed for communications. This information can be sent using an out-of-band TCP connection.

4.2 Buffer management

Each side of the communication uses buffer pools with equal size buffers to communicate. Two such pools are used for sending and receiving data for each channel. For receiving operations, all the buffers are posted at the beginning to the fabric. For transmitting messages, the buffers are filled with messages and posted to the fabric for transmission. After the transmit is complete the buffer is added back to the pool. The message receive and transmission completions are discovered using the completion queues. Individual buffer sizes are kept relatively large to accommodate the largest messages expected. If the buffer size is not enough for a single message, the message is divided in to pieces and put into multiple buffers. The message itself carry the length of the overall message and this information can be used to assemble the pieces if they are divided.

The stream manager de-serializes the protocol buffer message in order to determine the routing for the message and handling the acknowledgements. The TCP implementation first copies the incoming data into a buffer and then use this buffer to build the protocol buffer structures. This implementation can directly use the buffer allocated for receiving to build the protocol message.

4.3 Flow control at communication level

Neither Infiniband nor Omni-Path implement flow control between the communication parties, and it is up to the application developer to implement the much higher level functions. With reliable connections, Infiniband provides an error retry functionality for messages. In some situations this can be used as an in-build flow control mechanism. But for this application, we found that once the error retry started to happen, the stream of messages slowed down significantly before we could detect the situation at the software level to handle it appropriately.

This implementation uses a credit-based approach for flow control. The credit available for sender to communicate is equal to the number of buffers posted into the fabric by the receiver. Credit information is passed to the other side as part of data transmissions, or by using separate messages in case there are no data transmissions to send it. Each data message carries the current credit of the communication party as a 4-byte integer value. The credit messages do not take into account the credit available to avoid deadlocks, otherwise there may be situations where there is no credit available to send credit messages. Credit is sent when half of the credit is consumed rather than waiting for full credit to be used to reduce waiting time by the sending side.

4.4 Infiniband

The current implementation uses connection-oriented endpoints with channel semantics to transfer the messages. The messages are transferred reliably by the fabric and the message ordering is guaranteed. The completions are also in order of the work request submissions.

4.5 Intel Omni-Path

Intel Omni-Path does not support connection-oriented message transfers employed in the Infiniband implementation. The application uses reliable datagram message transfer with tag-based messaging. Communication channels between stream managers are overlaid on a single receive queue and a single send queue. Messages coming from different stream managers are distinguished based on the tag information they carry. The tag used in the implementation is a 64-bit integer which carries the source stream manager ID and the destination stream manager ID. Even though all the stream managers connecting to a single stream manager are sharing a single queue, they carry their own flow control by assigning a fixed amount of buffers to each channel. Unlike in Inifiniband, the work request completions are not in any order of their submission to the work queue. Because of this, the application keeps track of the submitted buffers and their completion order explicitly.

4.6 IP Over Fabric

IP over Fabric or IP over Infiniband(IPoIB) is a mechanism to allow regular TCP application to access the underlying high performance interconnects through the TCP API. For using IPoIB heron stream manager TCP sockets are bound to the IPoIB network interface explicitly without changing the existing TCP processing logic.

5 Experiments

An Intel Haswell HPC cluster called Juliet was used for the Infiniband experiments. The CPUs are Intel Xeon E5-2670 running at 2.30GHz. Each node has 24 cores (2 sockets x 12 cores each) with 128GB of main memory, 56Gbps Infiniband interconnect and 1Gbps dedicated Ethernet connection to other nodes. Intel Knights Landing(KNL) cluster was used for Omni-Path tests. Each node in KNL cluster has 72 cores (Intel Xeon Phi CPU 7250F, 1.40GHz) and is connected to a 100Gbps Omni-Path fabric and 1Gbps Ethernet connection. There are many variations of Ethernet, Infiniband and Omni-Path performing at different message rates and latencies. We conducted the experiments in the best available resources to us, even though tests like 10Gbps Ethernet would be very interesting to see as well.

We conducted several micro-benchmarks to measure the latency and throughput of the system. In these experiments the primary focus was given to communications

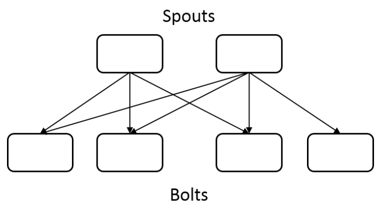


Figure 3: Topology A: Shallow topology with a spout and bolt connected in a shuffle grouping. The spout and bolt run multiple parallel instances as shown in the figure.

and no computation was conducted in the bolts. The tasks in each experiment were configured with 4GB of memory. A single Heron stream manager was run in each node.

5.1 Experiment Topologies

To measure the behavior of the system, two topologies shown in Fig. 3 and Fig. 4 is used. Topology A in Fig. 4 is a deep topology with multiple bolts arranged in a chain. The parallelism of the topology determines the number of parallel task for bolts and spout in the topology. Each adjacent component pair is connected by a shuffle grouping. Topology B in Fig. 3 is a two-component topology with a spout and a bolt. Spouts and bolts are arranged in a shuffle grouping so that the spouts load balance the messages among the bolts.

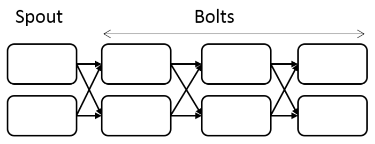


Figure 4: Topology B: Deep topology with a spout and multiple bolts arranged in a chain. The parallelism of the topology defines how many instances of bolts and spout in each stage.

In both topologies the spouts generated messages at the highest sustainable speed with acknowledgements. The acknowledgements acted as a flow control mechanism for the topology. The latency is measured as the time it takes for a Tuple to go through the topology and its corresponding acknowledgement to reach the spout. Since Tuples generate more Tuples when they go through the topology, it takes multiple acknowledgements to complete a Tuple. In topology A, it needs control Tuples equal to the number of Bolts in the chain to complete a single tuple. For example if the length of the topology is 8, it takes 7 control Tuples to complete the original Tuple. Tests with large messages run with maximum of 10 in flight messages through the topology and tests with small messages run with 100

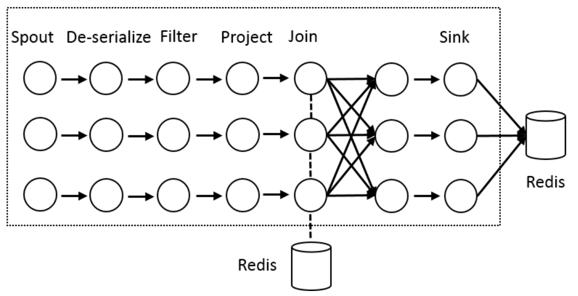


Figure 5: Yahoo Streaming benchmark with 7 stages. The join bolts and sink bolts communicate with a Redis server.

in flight messages. Also the store and forward draining rate is set to 1ms; which is the lowest value allowed.

5.2 Yahoo streaming benchmark

For evaluating the behavior with a more practical streaming application, we used a benchmark developed at Yahoo![3] to test streaming frameworks. We modified the original streaming benchmark to support Heron and added additional features to support our case. The modified benchmark is available open source in Github [4]. It focuses on an advertisement application where ad events are processed using a streaming topology as shown in Fig. 5. We changed the original benchmark to use a self-message-generating spout instead of reading messages through Kakfa [6]. This was done to remove any bottlenecks and variations imposed by Kafka.

The benchmark employs a multiple stage topology with different processing units. The data is generated as a JSON object and sent through the processing units, which do de-serialization, filter, projection and join operations on each tuple. At the joining stage, it uses a Redis [7] database to query data about the tuples. Additionally, the last bolt saves information to the Redis database. At the filter step about 66% of the tuples are dropped. We use an acking topology and measured the latency as the time it takes for a tuple to go through the topology and its ack to return back to the spout. For our tests we used 16 nodes with 8 parallelism at each step, totaling 56 tasks. Each spout was sending 100,000 messages per second, which gave 800,000 messages per second in total. Note that this topology accesses the Redis database system for 33% messages it receives.

[3]https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[4]https://github.com/iotcloud/streaming-benchmarks.git

6 Results & Discussion

Fig. 6 and Fig. 7 show the latency of the Topology A for large message sizes. Fig. 6 shows the latency with message sizes and Fig. 7 shows the latency with varying parallel instances. Infiniband performed the best while Ethernet showed the highest latency. It is interesting to see that Ethernet latency increases linearly while the other two were more on a curve. We should note that the Infiniband implementation only improved the connections between the stream managers while keeping the loop back connections, which are on TCP stack between the instances and stream managers.

Fig. 8 and Fig 9 show the latency for of the Topology A for small messages with varying sizes and parallelism. For small messages, we see both IPoIB and Ethernet performing at a similar level while Infiniband was performing much better. When increasing the parallelism, the latency increased as expected and Infiniband showed a smaller increase than IPoIB and Ethernet.

Fig. 10, Fig. 11, Fig. 12 and Fig 13 shows the results for Topology B with different message sizes and different parallel spout instances. The results are similar to the Topology A with much small latencies because of the shallow topology. For most practical applications, the minimum latencies will be within the results observed in these two topologies as these represent the minimum possible topology and a deep topology.

Fig. 14,15,16,17 shows the latency results of Omni-Path implementation in the KNL cluster for large and small messages. IPoFabric is the IP driver for Omni-Path. The KNL machine has much less powerful CPU's and hence the latency was higher compared to Infiniband in the tests.

Fig 18 shows the latency of the Topology A with 100 and 10 infligt messages with 128K message size. It is clear from the graphs that Ethernet implementation cannot function properly with 100 in-flight state and Infiniband works as expected.

Fig. 19 shows the latency distribution seen by the Yahoo stream benchmark with Infiniband fabric. For all three networking modes, we have seen high spikes at the 99th percentile. This was primarily due to Java garbage collections at the tasks which is unavoidable in JVM-based streaming applications. The store and forward functionality of the stream manager contributes to the distribution of latencies as a single message can be delayed up to 1ms randomly at stream manager.

The results showed good overall results for Infiniband and Omni-Path compared to the TCP and IPoIB communication modes. The throughput of the system is bounded by the CPU usage of stream managers. For TCP connections, the CPU is used for message processing as well as the network protocol. Infiniband, on the other hand, uses

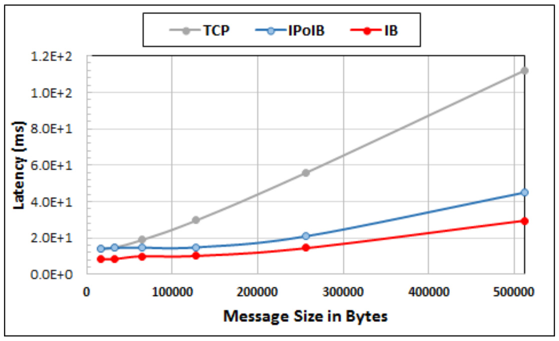


Figure 6: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism of 2. The experiment is conducted with 8 nodes. The message size varies from 16K to 512K bytes.

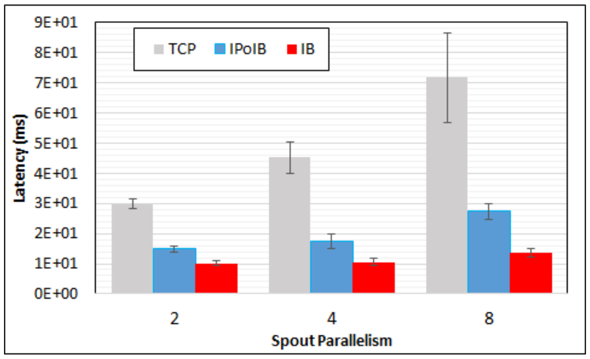


Figure 7: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism varying from 2 to 4. The experiment is conducted with 8 nodes. The message size is 128K bytes.

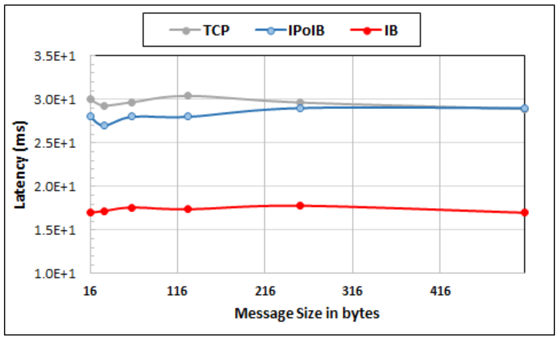


Figure 8: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism of 2. The experiment is conducted with 8 nodes. The message size varies from 16 to 512 bytes.

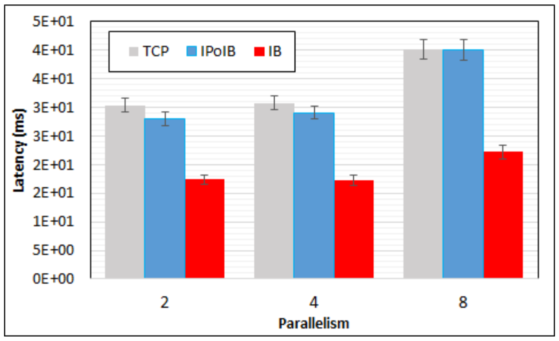


Figure 9: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism varying from 2 to 4. The experiment is conducted with 8 nodes. The message size is 128 bytes.

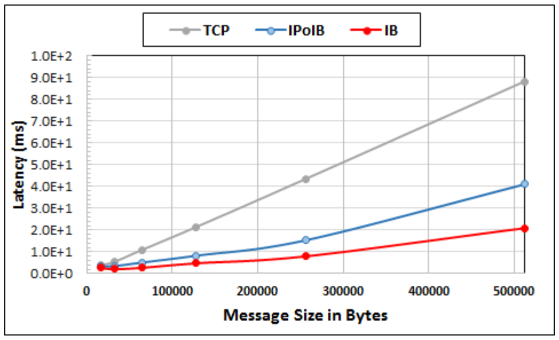


Figure 10: Latency of the Topology B with 32 parallel bolt instances and 16 parallel spout instances. The experiment is conducted with 8 nodes. The message size varies from 16K to 512K bytes.

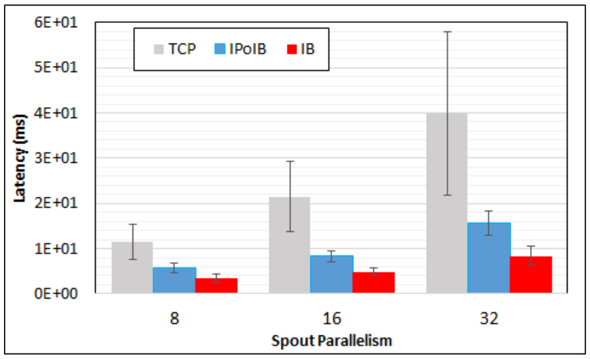


Figure 11: Latency of the Topology B with 32 parallel bolt instances and 8 to 32 parallel spout instances. The experiment is conducted with 8 nodes. The message size is 128K bytes.

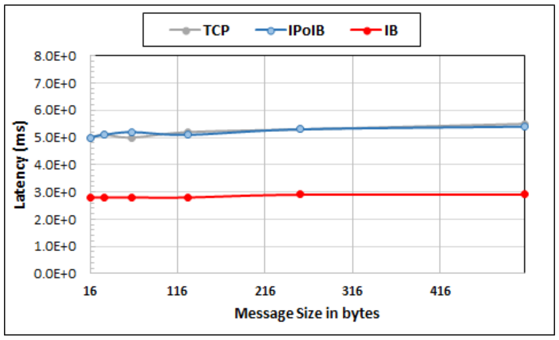


Figure 12: Latency of the Topology B with 32 parallel bolt instances and 16 parallel spout instances. The experiment is conducted with 8 nodes. The message size varies from 16 to 512 bytes.

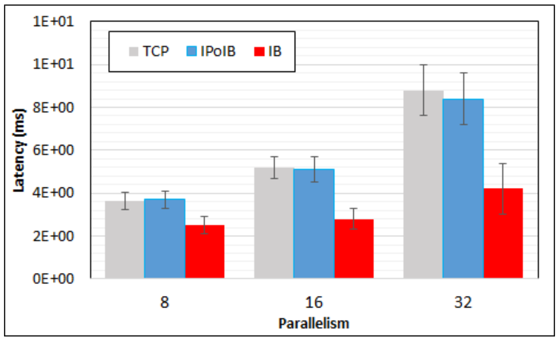


Figure 13: Latency of the Topology B with 32 parallel bolt instances and 8 to 32 parallel spout instances. The experiment is conducted with 8 nodes. The message size is 128 bytes.

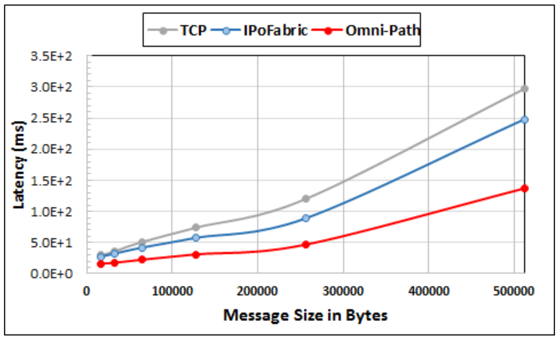


Figure 14: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism of 2. The experiment is conducted with 4 nodes. The message size varies from 16K to 512K bytes.

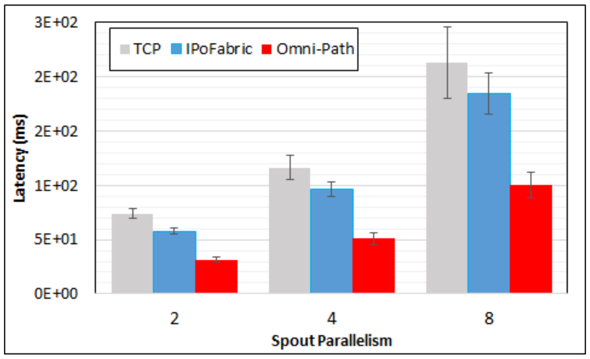


Figure 15: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism varying from 2 to 4. The experiment is conducted with 4 nodes. The message size is 128K bytes.

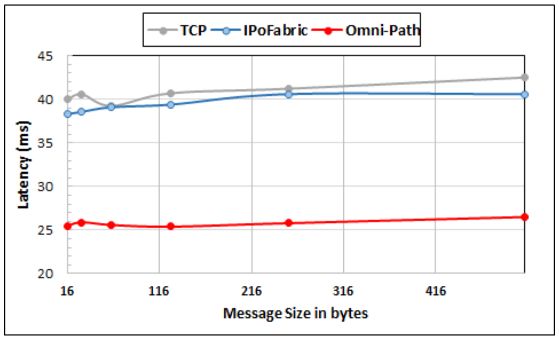


Figure 16: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism of 2. The experiment is conducted with 4 nodes. The message size varies from 16 to 512 bytes.

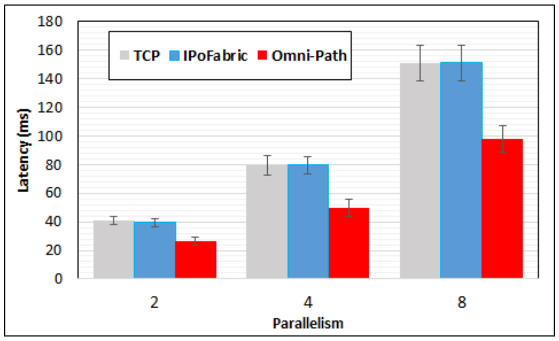
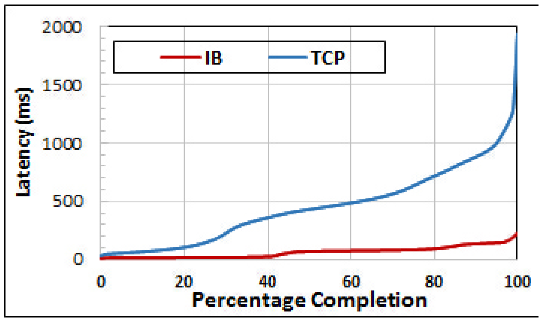
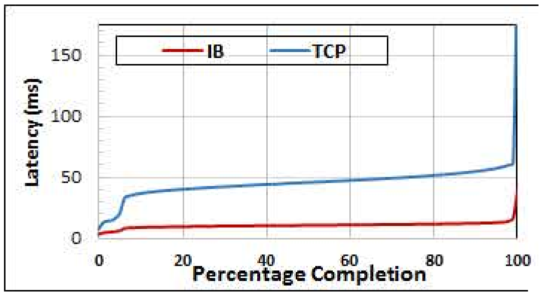


Figure 17: Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with parallelism varying from 2 to 4. The experiment is conducted with 4 nodes. The message size is 128 bytes.



(a) 100 Inflight messages



(b) 10 Inflight messages

Figure 18: Percent of messages completed within a given latency for the Topology A with in-flight messages at 100 and 10 with 128K messages and parallelism 4.

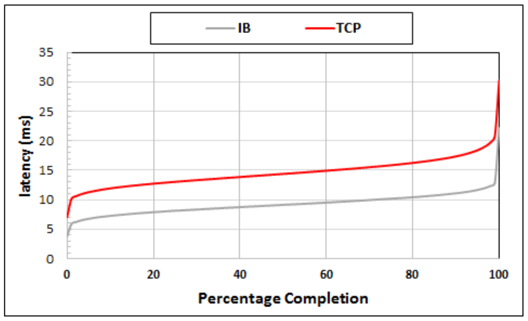


Figure 19: Percent of messages completed within a given latency for the Yahoo! streaming benchmark with Infiniband network. The experiment was conducted with 8 nodes and each stage of topology have 8 parallel tasks.

the CPU only for message processing at the stream manager, yielding better performance.

The results showed much higher difference between TCP and Infiniband for large message sizes. For small messages the bandwidth utilization is much lower than large messages. This is primarily due to the fact that CPU is needed to process every message. For larger messages, because of the low number of messages transferred per second, the CPU usage is low for that aspect. For smaller messages, because of the large number of messages per second, the CPU usage is much higher. Because of this, for small messages, the stream managers saturate the CPU without saturating the communication channel. For practical applications that require large throughput, Heron can bundle small messages into a large message in-order to avoid some of the processing overheads and transfer overheads. This makes it essentially a large message for the stream manager and large message results can be observed with elevated latencies for individual messages.

The KNL system used for testing Omni-Path has a large number of processes with low frequencies. In order to fully utilize such a system, multiple threads need to be used. For our implementation we did not explore such features specific to KNL and tried to first optimize for the Omni-Path interconnect. The results show that Omni-Path performed considerably better than the other two options. In this work the authors did not try to pick between Omni-Path or Infiniband as a better interconnect as they are tested in two completely different systems under varying circumstances. The objective of the work is to show the potential benefits of using interconnects to accelerate stream processing.

7 Related Work

In large part, HPC and Big Data systems have evolved independently over the years. Despite this, there are common requirements that raise similar issues in both worlds. Some of these issues are solved in HPC and some in Big Data frameworks. As such, there have been efforts to converge both HPC and Big Data technologies to create better systems that can work in different environments efficiently. SPIDAL [8] and HiBD are two such efforts to enhance the Big Data frameworks with ideas and tools from HPC. This work is part of an ongoing effort by the authors to improve stream engine performance using HPC techniques. Previously we showed [9] how to leverage shared memory and collective communication algorithms to increase the performance of Apache Storm. Also the authors have looked at various available network protocols for Big Data applications [10].

There are many distributed stream engines available today including Apache Spark Streaming [11], Apache Flink [12], Apache Apex [13] and Google Cloud Data flow [14]. All these systems follow the data flow model with comparable features to each other. Stream bench [15] is a benchmark developed to evaluate the performance of these systems in detail. These systems are primarily optimized for commodity hardware and clouds. There has been much research done around Apache Storm and Twitter Heron to improve its capabilities. [16] described architectural and design improvements to Heron that improved its performance much further. [17, 18] looks at schdueling streaming tasks in Storm to optimize for different environments.

Infiniband has been integrated into Spark [19] where the focus is on the Spark batch processing aspects rather than the streaming aspects. Spark is not considered a native stream processing engine and only implements streaming as an extension to its batch engine, making its latency inadequate for low latency applications. Recently Infiniband has been integrated into Hadoop [20], along with HDFS [21] as well. Hadoop, HDFS and Spark all use Java runtime for their implementations, hence the RDMA was integrated using JNI wrappers to C/C++ codes that invoke the underlying RDMA implementations.

High performance interconnects have been widely used by the HPC community, and most MPI(Message passing Interface) implementations have support for a large number of interconnects that are available today. Some early work that describes in detail about RDMA for MPI can be found in [22]. There has even been some work to build Hadoop-like systems using the existing MPI capabilities [23]. Photon [24] is a higher level RDMA library that can be used as a replacement to libfabric.

8 Conclusions

Unlike other Big Data systems which are purely JVM-based, Heron has a hybrid architecture where it uses both low level and high level languages appropriately. This architecture allows the addition of high performance enhancement such as different fabrics natively rather than going through additional layers of JNI as done in high performance Spark and Hadoop. At the very early stages of the high performance interconnect integration to Heron, we have seen good performance gains both in latency and throughput. The architecture and the implementations can be improved further to reduce the latency and increase the throughput of the system.

Even though the authors use Twitter Heron in this paper, the work is equally applicable to other distributed stream processing engines such as Apache Flink [12], Apache Spark [25] and Apache Apex [13]. Since these systems are JVM-based, instead of implementing communications natively, it would need to go through JNI.

9 Future Work

Past research has shown that the remote memory access operations of Infiniband are more efficient than using channel semantics for transferring large messages. A hybrid approach can be adopted to transfer messages using both channel semantics and memory semantics. It is evident that the CPU is a bottleneck at the stream managers to achieve better performance. The protocol buffer processing is a dominant CPU consumer at the stream managers. A more streamed binary protocol that does not require protocol buffer processing at the stream manager can avoid these overheads. Instances to stream manager communication can be improved with a shared memory approach to avoid the TCP stack. With such approach the Infiniband can be improved to directly use the shared memory for the buffers without relying on data copying.

Because of the single-process single-threaded approach used by Heron processes, many core systems such as Knights Landing cannot get optimum performance out of Heron. Having a hybrid architecture where multiple threads are used for both communication and computation utilizing the hardware threads of the many core systems can increase the performance of Heron in such environments.

Since we are using a fabric abstraction to program Infiniband and Omni-Path with Libfabric, the same code can be used with other potential high performance interconnects, though it has to be evaluated in such environments to identify possible changes. Heron architecture can be further improved to gain the maximum performance of the interconnects by introducing other advanced capabilities like shared memory communications between stream managers and instances.

A Throughput of the system

In this section we present the throughput results observed with Topology B. Even though we include throughput results in this section, it is not a primary focus on our investigation as throughput is a function of available bandwidth and CPU. With much higher bandwidth Ethernet and Infiniband these results will be different.

We present the message rates observed with Topology B. The experiment was conducted with 32 parallel bolt instances and varying numbers of spout instances. Fig. A.20 shows the results of running this experiment with 16 spouts and varying message size from 16K to 512K bytes. The graph shows that Infiniband had the best throughput, while IPoIB achieved second-best and Ethernet came in last. Fig. A.21 shows the throughput for 128K messages with varying number of spouts. When the number of parallel spouts increases, the IPoIB and Ethernet maxed out at 16 parallel spouts, while Infiniband kept on increasing. Fig. A.23 and Fig A.22 shows throughput results for small messages. Fig A.24 shows throughput of the system for large message size with Omni-Path. Again the throughput is much less compared to Infiniband primarily due to slower CPUs.

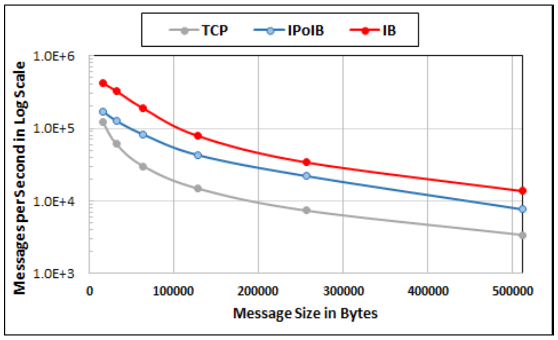


Figure A.20: Throughput of the Topology B with 16 parallel spout instances and 32 bolt instances arranged in a shuffle grouping running in 16 nodes. The message size varies from 16K to 512K bytes.

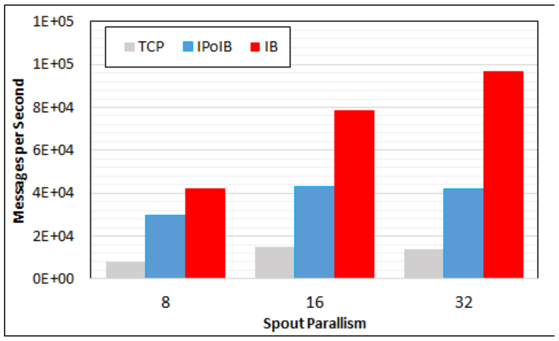


Figure A.21: Throughput of the Topology B with 8,16 and 32 parallel spout instances and 32 bolts instances arranged in a shuffle grouping running in 16 nodes. The message size is 128K bytes.

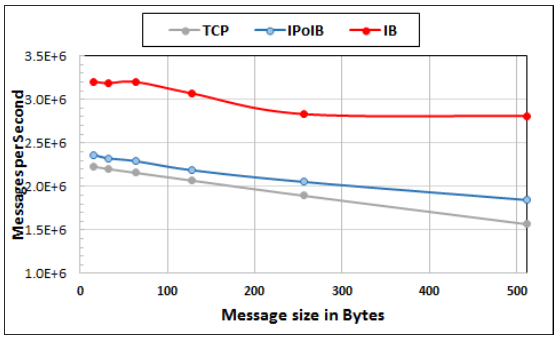


Figure A.22: Throughput of the Topology B with 16 parallel spout instances and 32 bolt instances arranged in a shuffle grouping running in 16 nodes. The message size varies from 16 to 512 bytes.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and NSF RaPyDLI 1415459. We thank Intel for their support of the Juliet system, and extend our gratitude to the FutureSystems team for their support with the infrastructure. We would like to thank Twitter Heron team for their support of this work and doing a wonderful job at developing Heron.

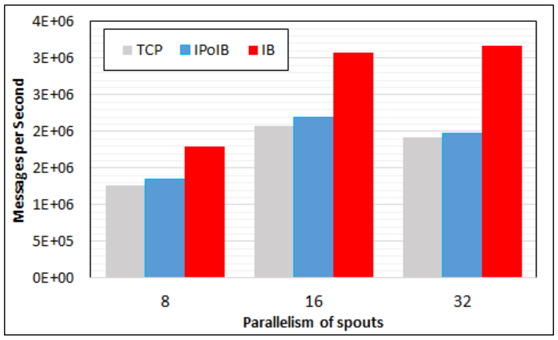


Figure A.23: Throughput of the Topology B with 8, 16 and 32 parallel spout instances and 32 bolt instances arranged in a shuffle grouping running in 16 nodes. The message size is 128 bytes.

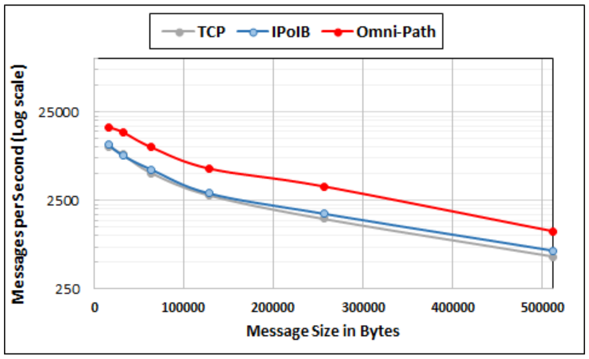


Figure A.24: Number of messages per second with Omni path and large number of messages.

References

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[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] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[4] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics," in *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 2015, pp. 1–9.

[5] K. Varda, "Protocol buffers: Googles data interchange format," *Google Open Source Blog, Available at least as early as Jul*, 2008.

[6] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[7] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[8] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and hpc convergence," in *Workshop on Big Data Benchmarks*. Springer, 2015, pp. 3–17.

[9] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox, "Towards High Performance Processing of Streaming Data in Large Data Centers," in *HPBDC 2016 IEEE International Workshop on High-Performance Big Data Computing in conjunction with The 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016), Chicago, Illinois USA*, 2016.

[10] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul, "Efficient data transfer protocols for big data," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*. IEEE, 2012, pp. 1–9.

[11] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.

[12] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Data Engineering*, p. 28, 2015.

[13] "Apache Apex: Enterprise-grade unified stream and batch processing engine." [Online]. Available: https://apex.apache.org/

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[15] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014, pp. 69–78.

[16] M. Fu, A. Agrawal, A. Floratou, G. Bill, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang, "Twitter heron: Towards extensible streaming engines," *2017 IEEE International Conference on Data Engineering*, Apr 2017. [Online]. Available: http://icde2017.sdsc.edu/industry-track

[17] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.

[18] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed qos-aware scheduling in storm," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015, pp. 344–347.

[19] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in *High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on*. IEEE, 2014, pp. 9–16.

[20] 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*. IEEE, 2013, pp. 641–650.

[21] N. S. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 35.

[22] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 295–304.

[23] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: extending mpi to hadoop-like big data computing," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 829–838.

[24] E. Kissel and M. Swany, “Photon: Remote memory access middleware for high-performance runtime systems,” in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1736–1743.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “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*. USENIX Association, 2012, pp. 2–2.