Towards High Performance Processing of Streaming Data in Large Data Centers

Supun Kamburugamuve, Saliya Ekanayake, Milinda Pathirage, Geoffrey Fox
School of Informatics and Computing
Indiana University
Bloomington, IN, United States
{skamburu, sekanaya, mpathira, gcf}@indiana.edu

Abstract—**Smart devices, mobile robots, ubiquitous sensors, and other connected devices in the Internet of Things (IoT) increasingly require real-time computations beyond their hardware limits to process the events they capture. Leveraging cloud infrastructures for these computational demands is a pattern adopted in the IoT community as one solution, which has led to a class of Dynamic Data Driven Applications (DDDA). These applications offload computations to the cloud through Distributed Stream Processing Frameworks (DSPF) such as Apache Storm. While DSPFs are efficient in computations, current implementations barely meet the strict low latency requirements of large scale DDDAs due to inefficient inter-process communication. This research implements efficient highly scalable communication algorithms and presents a comprehensive study of performance, taking into account the nature of these applications and characteristics of the cloud runtime environments. It further reduces communication costs within a node using an efficient shared memory approach. These algorithms are applicable in general to existing DSPFs and the results show significant improvements in latency over the default implementation in Apache Storm.**

I. Introduction

Real-time data processing at scale in cloud-based large data centers is challenging due to their strict latency requirements and distributed nature of applications. Modern distributed stream processing frameworks (DSPF) such as Apache Storm [1] provide an effective platform for real-time large scale applications. A parallel real-time distributed streaming algorithm implementing simultaneous localization and mapping (SLAM) [2] for mobile robots in the cloud is a good example of an application requiring low latency parallel processing with strict guarantees. Current DSPFs [3] are designed to cater to traditional event processing tasks such as extract transformation and load (ETL) pipelines, counting, frequent itemset mining, windowed aggregations and joins or pattern detection. The above-mentioned novel applications with strict real-time guarantees demand low-latency synchronous and asynchronous parallel processing of events, which is not a fully explored area in DSPFs.

The work in real-time applications in robotics [2], [4] and research into high performance computing on streaming [3] highlighted the fact that there are opportunities for further enhancements in distributed streaming systems. Particularly in the areas such as low-latency and efficient communication, scheduling of streaming tasks for predictable performance, and high-level programming abstractions. This paper focuses on efficient communication in a DSPF and look at how communication infrastructure of a DSPF can be improved to achieve low latency. Previous work has found that inefficient communication for data distribution operations, such as broadcasting in current DSPF implementations, are limiting the performance of parallel applications when the parallelism of the processing increases. Also, it was identified that communications among processes inside a node can be improved significantly using shared memory approaches.

A Distributed streaming application is generally represented as a graph where nodes are streaming operators and edges are communication links. Data flow occurs through the edges of the graph as streams of events. The operators at the nodes consume these event streams and produce output streams. With naive implementation, a collective operation such as broadcast is done through separate communication links (edges) from the source to each target serially. As shown in various areas such as MPI [5] and computer networks [6] these communications can be made efficient by modeling them based on data structures such as trees.

Apache Storm [1] is an open source distributed stream processing engine developed for large-scale stream processing. Its processing model closely resembles the graph-based data flow model that was described earlier. It is capable of low latency stream processing and has been used for real-time applications [2], [4], [7]. This paper presents the results of the improvements that was made to Apache Storm by implementing several broadcasting algorithms, as well as reducing communication overhead using shared memory. The underlying algorithms used for broadcasting are flat tree, binary tree and ring. Storm utilizes both process-based and thread-based parallel execution of stream operators. The tasks of a Storm streaming application run in different processes in different nodes. Tasks running in the same process can use the memory to communicate while those running in separate processes utilize networks. The communication algorithms are optimized in consideration of the task locality to improve network and inter-process communications. To test the system, a simple stream processing application is used with minimal processing at the parallel tasks for evaluating the behavior of the broadcasting algorithms with different data sizes and tasks.

The remainder of this paper is organized as follows: Section II discusses related work and then Section III presents Apache

Storm architecture in detail. Section IV provides details about our implementation of broadcasting algorithms and Section V provides details on shared memory communication improvements in Apache Storm. Experiments and results are discussed in Section VI and VII. Section VIII presents future work and Section IX conclude the paper.

II. Related Work

Apart from Apache Storm, there are several other DSPFs available, each solving specific problems with their own strengths. Such open source DSPFs include Apache Samza [8], Apache S4 [9], Apache Flink [10] and Apache Spark Streaming [11], with commercial solutions including Google Millwheel [12], Azure Stream Analytics and Amazon Kinesis. Early research in DSPFs include Aurora [13], Boreiles [14] and Spade [15]. Apache Spark Streaming uses micro-batch jobs on top of its batch engine to achieve streaming computations while Samza is a storage-first processing framework; both target high throughput applications rather than low latency applications. Apache Flink is comparable to Storm in its underlying execution and supports low latency message processing.

Neptune [16] and Heron [17] improve some of the inefficiencies of Apache Storm. Heron addresses issues in the task scheduling, enhances flow control by handling back pressure, and improves connection management for large applications and task isolation for better performance in Storm. Neptune streamlines the throughput of applications by using multiple threads for processing and efficient memory management. Apache Flink employs efficient fault tolerance and message processing guarantee algorithms [18] that are lightweight and introduce minimal overheads to the normal processing flow when compared to the upstream backup [19] mechanisms utilized by DSPFs such as Apache Storm. Further, Nasir et al. [20] utilize a probabilistic algorithm in Storm's load balancing message distribution operator to avoid imbalance work loads in tasks. Adaptive scheduling [21] improves the scheduling of Storm by taking into account the communication patterns among the tasks and Cardellini et al. [22] has improved scheduling to take QoS into account.

In MPI, operations requiring more than simple P2P communications are termed collective operations and the term collective communications is used to refer to underlying communication infrastructures. The collective communications are optimized for HPC applications [5], [23] in technologies such as MPI. There are many such collective operations available including *Gather*, *AllGather*, *Reduce*, *AllReduce*, *Broadcast*, *Scatter*, etc. Multiple communication algorithms apply to each of these, and their suitability depends on the message size, cluster size, number of processes and networks. There has been much discussion over various techniques to choose the best algorithm for a given application and hardware configuration [24], [25]. Recently collective communications are being introduced to batch processing big data solutions requiring rich communication [26], [27]. These improvements have greatly enhanced the performance of applications implemented on top of these platforms. High performance interconnects like RDMA [28] are being studied for MPI applications to further reduce the communication among the processes. Also shared memory communications [29] are being used for inter-process communications in MPI applications. This paper investigates how to bring these improvements to Distributed Stream Processing applications.

III. Apache Storm

Every DSPF consists of two logical layers identified as the application layer and the execution layer. The application layer provides the API for the user to define a stream processing application as a graph. The execution layer converts this user defined graph into an execution graph and executes it on a distributed set of nodes in a cluster.

A. Storm Application Layer

A Storm application called a topology determines the data flow graph, with streams defining the edges and processing elements defining the nodes. A stream is an unbounded sequence of events flowing through the edges of the graph, and each such event consists of a chunk of data. A node in the graph is a stream operator implemented by the user. The entry nodes in the graph acting as event sources to the rest of the topology are termed Spouts while the rest of the data processing nodes are called Bolts. The spouts generate event streams to the topology by connecting to external event sources such as message brokers. From here onwards we refer to both spouts and bolts as processing elements (PEs) or operators. Bolts consume input streams and produce output streams. The user code inside a bolt executes when an event is delivered to it on incoming links. The topology defines the logical flow of data among the PEs in the graph by using streams to connect PEs. A user can also define the parameters necessary to convert this user defined graph into an execution graph. The physical layout of the data flow is mainly defined by the parallelism of each processing element and the communication strategies defined among them. This graph is defined by the user who deploys it to the Storm cluster to be executed. Once deployed, the topology runs continuously, processing incoming events until it is terminated by the user. An example topology is shown in Figure 1 where it has a spout connected to a bolt by a stream and a second bolt connected to the first bolt by another stream.

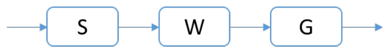


Fig. 1. A sample stream processing user defined graph

B. Storm Execution layer

Storm master (known as Nimbus) converts logical graph of processing elements to an execution graph by taking the number of parallel tasks for each logical PE and the stream grouping III-C into account. For example, Figure 2 displays an execution graph of the user graph shown in Figure 1 where

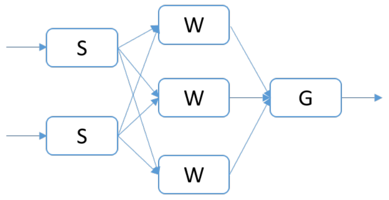


Fig. 2. A sample stream processing execution graph

two instances of S, three instances of W and one instance of G are running. The stream grouping between S and W is a load balancing grouping where each instance of S distributes its output to the 3 W instances in a round-robin fashion. A runtime instance of a node in the execution graph is called a task.

After converting logical graph to execution graph, master node takes care of scheduling of the execution graph and also manages the stream processing applications running in the cluster. Each slave node runs a daemon called a supervisor, which is responsible for executing a set of worker processes which in turn execute the tasks of the execution graph. Tasks in an execution graph will get assigned to multiple workers running in the cluster. Figure 3 shows one configuration of the example topology assigned to two nodes both running two workers. Each worker can host multiple tasks of the same graph, and the worker assigned a thread of execution to every task. If multiple tasks run in the same worker, multiple threads execute the user codes in the same worker process.

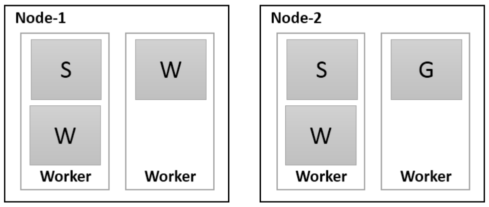


Fig. 3. Storm task distribution in multiple nodes

C. Communication

The communication strategy between two nodes in the user graph is called stream grouping. A stream grouping defines how the messages are distributed from one set of parallel tasks of the first processing element to another set of parallel tasks of the second processing element. As an example, in Shuffle grouping, messages are sent from each task of the first component to all the other tasks of the second component in a round-robin fashion, thereby achieving load balancing of the events for the distributed tasks. Other communication strategies include patterns like key-based distributions, broadcasting and direct message sending.

The default implementation of Apache Storm uses a TCP-based binary protocol for sending messages between tasks. Connections are established between worker processes which carry messages with a destination task ID. Workers then use this in determining the correct task to deliver their message. Storm uses Kryo Java object serialization to create byte messages from the user objects that need to be transferred. Communication between the tasks running in the same worker happen through the process memory via object references; no serialization or deserialization is involved.

This design implies that there is a single TCP port at which every worker is listening for incoming messages. These ports are known across the cluster and workers connect to each other using these ports. The design reduces the number of connections required for an application. With the default implementation, the tasks within the nodes also communicate using TCP, which can be efficient because of the loopback adapter but can be further improved using shared memory-based communications.

IV. Broadcast

Broadcasting is a widely used message distribution strategy in Apache Storm. Broadcasting involves a task instance sending a message to all the tasks of another node in the user graph. When this is applied to a continuous stream of messages, the broadcasting happens continuously for each message. Let's assume propagation delay of l_t for TCP and l_s for processes inside a node; transmission delay of m_t for TCP and m_s for for processes inside a node and there are n nodes participating in broadcast each having w workers. The default implementation of Storm serially sends the same message to each task as a separate message and this method is inefficient due to the following reasons: 1. Max latency is at least $m_s \times n + l_t$. 2. If the message size is M it takes at least $M \times n$ network bandwidth of the broadcasting node. 3. A worker can run multiple tasks and the same message is sent to the worker multiple times.

The three algorithms developed reduce deficiencies 1 and 2 mentioned above to varying degrees and eliminate 3 completely. They use a tree model to arrange the edges of the broadcast part of execution graph and take advantage of the fact that communication among the processes in a single computer is less expensive compared to inter machine communications. The workers are mapped to nodes of the tree instead of individual tasks because communication cost is zero between the tasks running in the same worker due to in-memory message transfers. To preserve the worker locality within a node, a machine participating in the broadcast operation uses at most one incoming message stream and one outgoing message stream for the broadcasting operation. This rule reduces the network communication drastically because it minimizes inter-machine communications and maximizes intra-node communications. Figure 4 shows an example message distribution for the broadcast operation with the three algorithms with 2 machines each running 4 workers.

A. Flat tree

Flat tree algorithm broadcasting has a root level branching factor equal to the number of nodes with active topology workers participating in the broadcasting operation. This means

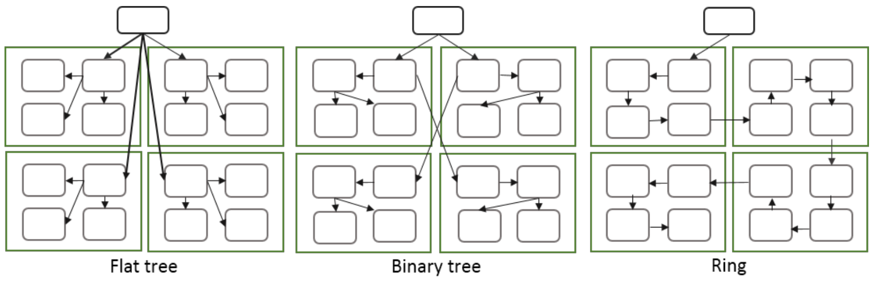


Fig. 4. Example broadcasting communications for each algorithm in a 4 node cluster with each machine having 4 workers. The outer green boxes show cluster machines and inner small boxes show workers. The top box displays the broadcasting worker and arrows illustrate the communication among the workers

the broadcast nodes first send the message to a worker in each node. After the designated worker in the node receives the message, it is distributed to workers running in that node sequentially or using a binary tree. The max latency observed by the workers will be $n \times m_t + l_t + (w-1)m_s + l_s$ for flat tree distribution inside the node.

B. Binary Tree

Binary tree algorithms broadcast to two workers in the first level and these two workers broadcast to another four workers. When picking the first two workers the algorithm always tries to use two workers in two nodes. The worker receiving the messages from upper machine broadcast to an worker inside its machine and another worker in another machine if there are such workers and machines available. The algorithm gives high priority to tasks in the same worker of the broadcast task and tasks in other workers in the same node in that order. So the tree always expands through those workers in the first levels if there are such workers. The max latency will be of the order $log_2 n \times (l_t + m_t) + log_2 w(l_s + 2m_s)$.

C. Ring

As shown in Figure 4, the ring starts at the broadcast worker and goes through all the workers participating in the broadcast. It always connects the workers of a single node first before reaching to the next node. Two variants of the ring algorithm are used. In the first variant the ring starts from one node and ends at the last node connecting all the tasks. In the second case two communications are started from the root and each of these connects half of the nodes in the broadcast creating a bidirectional ring. The broadcast takes the task locality into account and always starts the ring from tasks running in the same worker as broadcast tasks and then connects the workers in the same machine. The ring topology used with a stream of messages becomes a communication pipeline as incoming messages are routed through the workers while other messages transmitted before are still going through the worker. The max latency will be $(w-1)(l_s + m_s)n + (n-1)(l_t + m_t)n$ for full ring and half of this for bidirectional ring.

V. Shared Memory Communications

DSPFs use TCP messages to communicate within processes of a machine. Apache Storm, for example, may create up to $(w-1) * w$ TCP connections to communicate among tasks running with w workers in a node. Such communication poses a significant bottleneck considering the fact that they occur within a single node. As an efficient alternative, this work implements a Java shared memory maps-based communication between intra-node workers.

While Java has built-in support for memory maps, it does not have consistency guarantees to be used as an inter-process communication technique. Therefore, we implement a custom multiple writer-singer-reader styled memory map-based queuing system. This implementation is safe to use across multiple platforms and it is possible to use either the file system or main memory to persist messages. In Linux systems the special $tmpfs$ directory, usually mounted as $/dev/shm$, points to the main memory, which is more efficient than using a regular file in this queue implementation.

In the shared memory implementation each Storm worker has a memory mapped file allocated to it with a single reader. This reader continuously reads its file for new messages written by workers. The file is written and read using fixed size message chunks, meaning a message will be broken into multiple parts during communication. Also, these chunks include a unique ID and a sequence number to guarantee correctness and to match messages with the corresponding writer as it is possible for messages to arrive in a mixed order due to multiple writers writing to the same file. Figure 6 shows the structure of a packet. Each packet contains a UUID to identify which message it belongs to, the source task, destination task, total number of packets and the current packet number. A multiple part messaging is used because the file size is fixed and the message sizes do not always fit the remaining file size. Also multiple writers can write to the file without waiting for a single writer to finish with a large message. The implementation was started with open source shared memory bus [1] and deviated from it because of the continuous streaming requirements.

The structure of a single file is shown in Figure 5. To achieve multiple writers, a shared long integer is stored at the beginning of the file to mark the used space. When a writer intends to write a packet, it incrementally alters this

[1]https://github.com/caplogic/Mappedbus

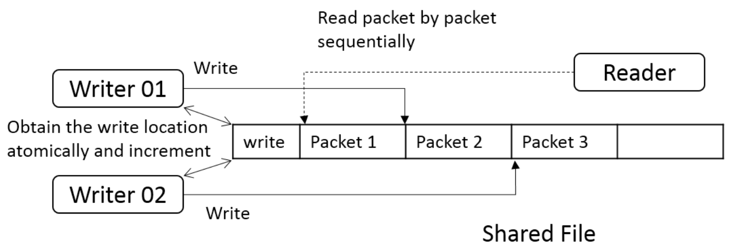
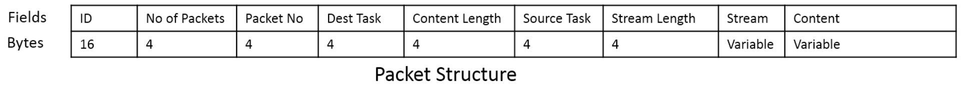

Read packet by packet sequentially
Reader
Write
Writer 01
Obtain the write location atomically and increment
write
Packet 1
Packet 2
Packet 3
Writer 02
Write
Shared File

Fig. 5. The structure of the shared memory file



Fields	ID	No of Packets	Packet No	Dest Task	Content Length	Source Task	Stream Length	Stream	Content
Bytes	16	4	4	4	4	4	4	Variable	Variable

Packet Structure

Fig. 6. The structure of the data packet sent through shared memory

field by the packet size atomically and gets the new address. Then it writes the packet to the allocated space. Because of the atomic increment of the value, multiple writers can write to the same file at the same time. When the file limit is reached the writer allocates a new file and starts writing to that. While allocating a new file, the writers acquire a lock using another shared file to prevent multiple writers allocating the same file. With the current design of Storm, a separate reader has to poll the file for messages and these are put into a queue where the processing threads pick the message out. In the future these two can be combined into one thread for reading and processing. The source code of the improvements are available in github repository [2].

VI. Experiment Setup

The experiments are conducted on a dedicated cluster using 10 nodes, each having 4 Intel Xeon E7450 CPUs at 2.40GHz with 6 cores, totaling 24 cores per node; and 48GB main memory running Red Hat Enterprise Linux Server release 5.11 (Tikanga) OS. The nodes are connected using a 1GB/s standard Ethernet connection. The nimbus and Zookeeper were running in one node, 8 Supervisors were running in 8 nodes each with 4 workers, and RabbitMQ [30] was running in another node. Each worker was configured to have 2GB of memory. This configuration creates 32 workers in the 8 nodes. Our experiments used 32 workers in the system.

Figure 7 shows the topology used for experiments, which includes a receive spout (S), broadcasting bolt (B), set of worker bolts (W) and a gather bolt (G). The experiments were set up to measure the latency and throughput of the application when message size and number of parallel tasks change. To measure the latency, a client sends a message to the spout R using RabbitMQ including the message generation time. Then this message is broadcast to N worker bolts W by broadcast bolt B and they send these messages to Gather bolt G. Finally the message is transmitted back to the client with the original timestamp and round trip time is calculated. Using this setup we avoid time measurements across different machines which can be inaccurate due to time skew. To measure number of messages transferred per second using the broadcast operation, bolt B generated a set of messages and sent them through W to G at which point G measured the time it takes to completely receive the messages.

Two sets of experiments are conducted with TCP-based and shared memory-based messaging. All the latency results are for round trip latency, and theoretically a constant including the message broker overhead should be subtracted from latency to get the true time, but since this is constant for all the experiments and because we are comparing the results of previous and new implementations, we did not consider this cost.

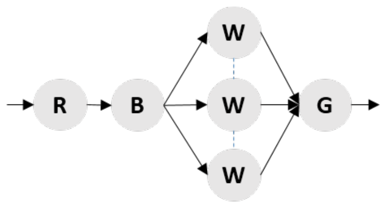


Fig. 7. Storm application graph

VII. Results & Discussion

The first experiment was conducted to measure the effect of memory mapped communications compared to the default TCP communications among the worker processes in a node. A topology with a communication going from worker to worker sequentially was used to measure the difference between the two. The communication connects the workers in one node and then connects to a worker in another node. All the workers in the Storm cluster were used for the experiment. The difference between the TCP latency and memory mapped latency is shown in Figure 8. Figure 8 shows the latency observed with the default TCP implementation. With only 10 tasks running in parallel and 32 workers available in

[2]https://github.com/iotcloud/jstorm

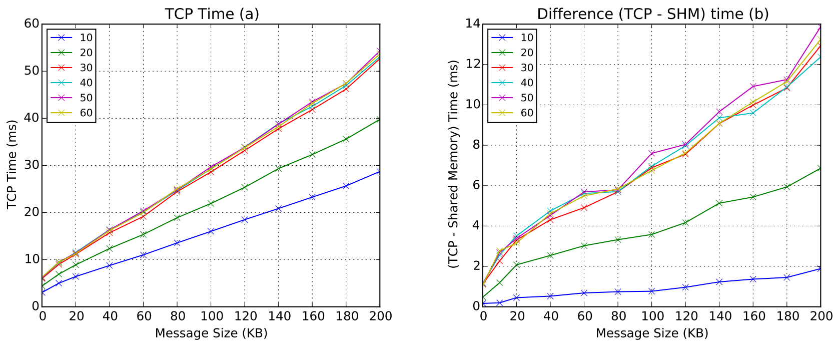


Fig. 8. Relative Importance of Shared Memory Communication compared to TCP in a broadcasting ring, (a) The time for TCP communications. (b) Y-axis shows the difference in latency for TCP implementation and shared memory implementation (TCP - SHM)

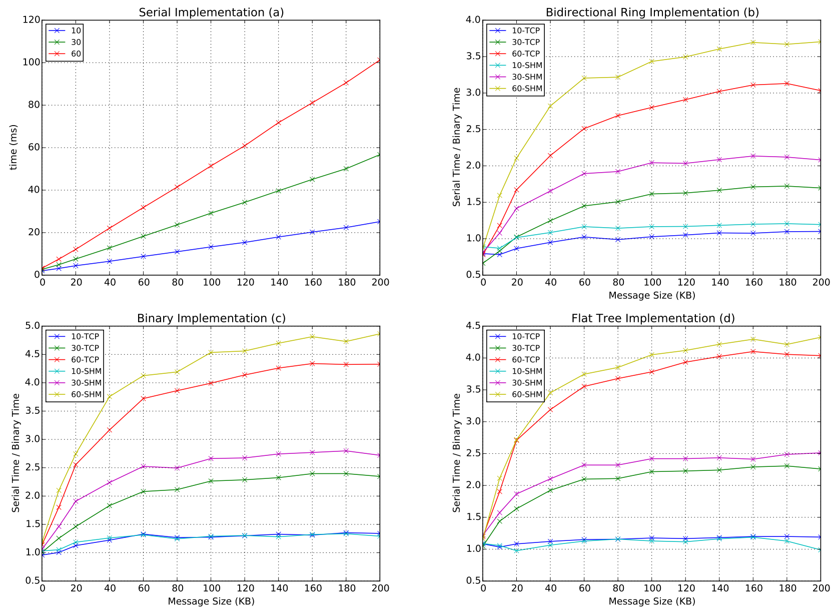


Fig. 9. Latency of serial, binary tree, flat tree and bi-directional ring implementations. Different lines show varying parallel tasks with TCP communications and shared memory communications(SHM).

8 nodes, 6 nodes will have 1 task each and 2 nodes will have 2 tasks apiece. This means the use of memory mapped communication is minimal with only 10 tasks. Because of this the difference between the two communications is practically zero. When increasing the number of tasks and the message size beyond 10K, it is clear from Figure 8 that we are gaining significant latency improvement by using memory mapped files, especially when the number of tasks increases. Beyond 30 parallel tasks all the workers in the cluster are used by the topology, and because in-memory messaging is used between the tasks inside a worker there is practically no difference between latency for 30 and 60 tasks.

Figure 9b, 9c and 9d shows the gain in latency with bidirectional ring, binary tree and flat tree-based broadcasting algorithms, compared to the default serial implementation latency shown in Figure 9a. The Y-axis of the graphs shows the improvements made compared to original time, i.e. $Originaltime/Improvedtime$. The binary tree algorithm performs the best among the three with about 5 times latency gain compared to the default algorithm for small message sizes which are most common in distributed streaming processing applications. Shared memory implementations show a further decrease in latency compared to the default TCP implementation for communications in a single node. The ring topology has the least latency decrease and flat tree falls between binary tree and ring. The effect of shared memory improvements are less for binary tree and flat tree compared to ring because they are dominated mostly by the TCP communication among the nodes.

Figure 9 shows the arithmetic average latency of the

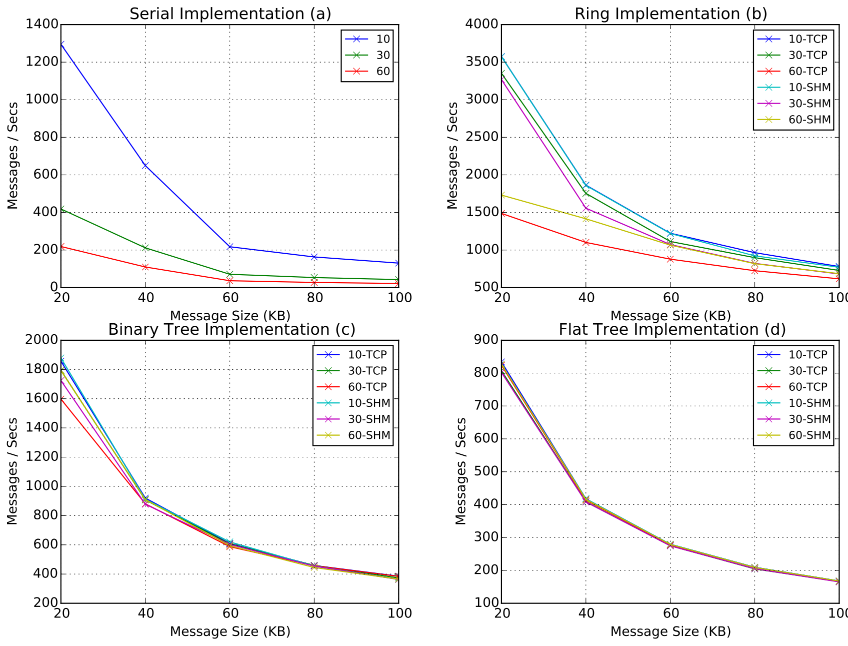


Fig. 10. Throughput of serial, binary tree, flat tree and ring implementations. Different lines show varying parallel tasks with TCP communications and shared memory communications (SHM)

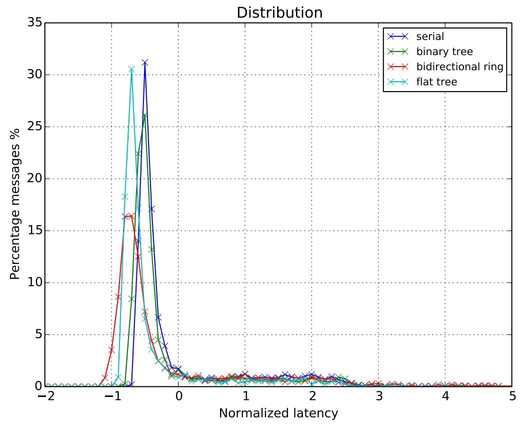


Fig. 11. The distribution of message latencies for 100K messages with 60 tasks. The X-axis shows normalized latencies and Y-axis shows the percentage of messages observed for a latency

topology. By looking at the distribution of individual latencies observed in the default implementation and improved versions, we concluded that there is no significant change in the latency distributions after applying the algorithms. It was observed that the variations in latency are mostly due to Java garbage collections and the original and improved results do not show any significant deviation in distribution due to that fact.

Figure 9 shows a micro-benchmark for broadcasting operation with default throughput 10a and ring 10b, binary tree 10c and flat tree 10d throughput. As expected the ring topology performs the best compared to the other algorithms. The shared memory latency has minimal effect to the throughput because it is dominated by the TCP connections between the nodes. In the ring implementation the throughput for 60 parallel tasks is about half of 30 and 10 parallelism. With 0 parallelism, one worker hosts two worker tasks. These two worker tasks needs to send the same message they receive to the gather bolts, resulting in the low throughput. This effect is not seen in the other algorithms because the network is saturated at the broadcast worker.

VIII. Future Work

The algorithm selection for the collective operation has to be done manually considering the message size and required behavior of the application. This process can be automated and the framework can choose the collective algorithms at the runtime. Making scheduling decisions while taking into account the communication algorithms can further improve the latency. Both process-based parallelism and thread-based parallelism are used in a loose way in DSPFs using a few threads at different stages for processing a single message. Some threads are shared among all the tasks running in the worker and some are dedicated to specific tasks. This model makes it hard to analyze the performance of the applications in a deterministic way when the number of tasks increases and especially when tasks perform different functions. A better model is needed for those applications which demand deterministic processing with clear bounds on latency. Furthermore the communications supported by the DSPFs do not include rich communication operations such as scatter, gather, and barrier needed for true parallel streaming computations. Investigating how to incorporate such operations to DSPFs would be a great addition.

IX. Conclusion

The results show significant performance improvements in Apache Storm when applying collective algorithms for streaming applications along with shared memory-based communications. The algorithms implemented take the worker distribution and nature of the parallelism of streaming applications into account. Binary tree algorithm showed the best latency for a reasonable range of message sizes with both shared memory messaging and TCP messaging. The shared memory improvements are important specially with the advancement of processor technology that increasingly adds many cores into a single chip. Throughput is best when using the ring algorithms and no significant gains are shown in throughput with shared memory messaging because of the dominance of TCP communication. Bidirectional ring is a compromise between the throughput and latency for applications requiring both. The techniques described in this paper are not limited to Storm and are equally applicable to other DSPFs.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and AFOSR FA9550-13-1-0225 awards. We would also like to express our gratitude to the staff and system administrators at the Digital Science Center (DSC) at Indiana University for supporting us on this work.

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. Kamburugamuve, H. He, G. Fox, and D. Crandall, “Cloud-based Parallel Implementation of SLAM for Mobile Robots.”

[3] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu, “Survey of distributed stream processing for large stream sources,” Technical report. 2016. Available at http://dsc.soic.indiana.edu/publications/ survey_distributed_stream_frameworks.pdf, Tech. Rep., 2016.

[4] H. He, S. Kamburugamuve, and G. C. Fox, “Cloud based real-time multi-robot collision avoidance for swarm robotics.”

[5] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, “Performance analysis of MPI collective operations,” *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[6] J. E. Wieselthier, G. D. Nguyen, and A. Ephremides, “On the construction of energy-efficient broadcast and multicast trees in wireless networks,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2. IEEE, 2000, pp. 585–594.

[7] S. Kamburugamuve, L. Christiansen, and G. Fox, “A Framework for Real Time Processing of Sensor Data in the Cloud,” *Journal of Sensors*, vol. 2015, 2015.

[8] “Apache Samza,” https://samza.apache.org/, accessed: 2016.

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[10] “Apache Flink,” https://flink.apache.org/, accessed: 2016.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “Millwheel: fault-tolerant stream processing at internet scale,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[13] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[14] “The Design of the Borealis Stream Processing Engine., author=Abadi, Daniel J and Ahmad, Yanif and Balazinska, Magdalena and Cetintemel, Ugur and Cherniack, Mitch and Hwang, Jeong-Hyon and Lindner, Wolfgang and Maskey, Anurag and Rasin, Alex and Ryvkina, Esther and others, booktitle=CIDR, volume=5, pages=277–289, year=2005.”

[15] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, “SPADE: the system s declarative stream processing engine,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1123–1134.

[16] T. Buddhika and S. Pallickara, “NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments.”

[17] 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.

[18] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, “Lightweight Asynchronous Snapshots for Distributed Dataflows,” *arXiv preprint arXiv:1506.08603*, 2015.

[19] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, “High-availability algorithms for distributed stream processing,” in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 779–790.

[20] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, “The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines,” *arXiv preprint arXiv:1504.00788*, 2015.

[21] 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.

[22] 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.

[23] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, “Optimization of MPI collective communication on BlueGene/L systems,” in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 253–262.

[24] A. Faraj and X. Yuan, “Automatic generation and tuning of MPI collective communication routines,” in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 393–402.

[25] A. Faraj, X. Yuan, and D. Lowenthal, “STAR-MPI: self tuned adaptive routines for MPI collective operations,” in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 199–208.

[26] B. Zhang, Y. Ruan, and J. Qiu, “Harp: Collective communication on hadoop,” in *IEEE International Conference on Cloud Engineering (IC2E) conference*, 2014.

[27] T. Gunarathne, J. Qiu, and D. Gannon, “Towards a Collective Layer in the Big Data Stack,” in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 236–245.

[28] J. Liu, J. Wu, and D. K. Panda, “High performance RDMA-based MPI implementation over InfiniBand,” *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[29] R. L. Graham and G. Shipman, “MPI support for multi-core architectures: Optimized shared memory collectives,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2008, pp. 130–140.

[30] A. Videla and J. J. Williams, *RabbitMQ in action*. Manning, 2012.