Apache Spark Streaming and HarmonicIO: A Performance and Architecture Comparison

Preprint · July 2018

3 authors:

Ben Blamey
Cardiff University
6 PUBLICATIONS **24** CITATIONS

Andreas Hellander
Uppsala University
60 PUBLICATIONS **828** CITATIONS

Salman Toor
Helsinki Institute of Physics (HIP)
30 PUBLICATIONS **316** CITATIONS

Apache Spark Streaming, Kafka and HarmonicIO: A Performance and Architecture Comparison for Enterprise and Scientific Computing

Ben Blamey*, Andreas Hellander* and Salman Toor*
* Department of Information Technology, Division of Scientific Computing, Uppsala University, Sweden
Email: {Ben.Blamey, Andreas.Hellander, Salman.Toor}@it.uu.se

***Abstract*—This paper presents a benchmark of stream processing throughput comparing Apache Spark Streaming (under file-, socket- and Kafka-based stream integration), with a prototype P2P stream processing framework, *HarmonicIO*. Maximum throughput for broad range of stream processing loads are measured, in particular, those with large message sizes (up to 10MB), and heavy CPU load – loads more typical of scientific computing use cases (such as microscopy), than enterprise contexts. A detailed exploration of the performance characteristics of these integrations under varying loads reveals a complex interplay of performance trade-offs, uncovering the boundaries of good performance for each framework and integration. Based on these results, we suggest which frameworks and integrations are likely to offer good performance for a given load. Broadly, the advantages of Spark's rich feature set comes at a cost of sensitivity to message size in particular, whereas the simplicity of HarmonicIO offers more robust performance, especially for raw CPU utilization.**

***Keywords*-Stream; Spark, HarmonicIO, high-throughput microscopy; Performance; HPC; Benchmark; XaaS; HASTE.**

I. Introduction

A number of stream processing frameworks have gained wide adoption over the last decade or so (Apache Flink (Carbone et al., 2015), Apache Spark Streaming (Zaharia et al., 2016), Flume (Apache Flume, 2016)); suitable for high-volume, high-reliability stream processing workloads. The development of these frameworks has been motivated by a variety of use-cases, including the analysis of data from cloud, web and mobile applications; Apache Flume for example specifically is similarly designed for the analysis of server application log data. Apache Spark improves upon the Apache Hadoop framework (Apa, 2011) for distributed computing, and was later extended with streaming support – whilst Apache Flink was later developed primarily for stream processing (in Flink, batching is handled as a special case of streaming, whilst in Spark the opposite is true).

These frameworks boast high throughput performance, scalability, data security, processing guarantees, and efficient, parallelized computations of processing operations; together with high-level stream processing APIs (augmenting familiar map/reduce with stream-specific functionality such as windowing). All of which make thems attractive for scientific computing – such as imaging applications in the life-sciences, and simulation output more generally.

Previous studies have shown that such frameworks are capable of processing message streams of very high frequency (the order of 1 million or more messages per second), but focus on use cases with textual rather than binary content, and of size perhaps a few KB. Additionally, the computational cost of processing an individual message may be relatively small (e.g. parsing JSON, and applying some business logic). By contrast, in scientific computing domains messages can be much larger (order of several MB).

Our motivating use case is the development of a cloud pipeline for the processing of streams of microscopy images, for biomedical research applications. Existing systems for working with such datasets have largely focused on offline processing: our online processing, (processing the 'live' stream), is relatively novel for the image microscopy domain.

Electron microscopes generate high-frequency streams of large, high-resolution image files (message sizes 2-10Mb), and feature extraction is computationally intensive. This is typical of many scientific computing use cases: where files have binary content, with execution time dominated by the per-message 'map' stage. Limiting ourselves to streaming (rather than batch) processing, we might consider file sizes not exceeding 10MB or so.

In this paper, we investigate how well enterprise-grade stream processing frameworks (such as Apache Spark) handle application loads more characteristic of scientific computing, for example, microscopy image stream processing, under conditions representative of both enterprise and scientific computing use cases. In particular, we investigate how well typical configurations of these frameworks perform when processing streams of messages with sizes on the order of megabytes, and/or with heavy per-message CPU cost – contrasting with the focus in previous benchmarking studies on small messages and lighter map stages.

We undertook performance benchmarking of Apache Spark Streaming (ASS), in various configurations, varying the message size and the computational cost of processing each message – simulating a wide spectrum of use cases, expanding on previous studies.

For comparison, we measured the performance of HarmonicIO (Torruangwatthana et al., 2018) – a research prototype with a which has a primarily P2P-based architecture, under the same conditions.

In summary, we find that stream integrations for Apache Spark suitable for small messages (TCP, Kafka) perform very well, whereas file-based stream processing adapts poorly to the message sizes in our study. However, the excellent performance of TCP-streaming and Kafka is sensitive to message load (both performing poorly for larger messages in our study). The overheads associated with Kafka impact the overall cluster CPU utility, relative to the more lightweight HarmonicIO; which performs robustly over a broad range of message sizes

We conclude that robust performance over wide domain of potential applications is an a attractive performance characteristic of HarmonicIO, and whilst lacking many features of Apache Spark, and not matching performance in the enterprise use case, it offers robust performance across a wide range of loads characteristic of scientific computing applications (including microscopy image analysis). Its lack of functionality for replication and fault tolerance perhaps presents a barrier to adoption in production pipelines for analysis of data from physical experiments, if not reproducible numerical simulations.

This paper contributes:

- A performance comparison of an industry standard framework (Apache Spark) for stream processing to a streaming framework tailored for scientific use cases (HarmonicIO).
- An analysis of these results, and comparison with theoretical bounds – relating the findings to the architectures of the framework configurations. Our findings show, albeit for our setup, that the relative performance of the configurations vary considerably according to the application loads – quantifying where these transitions occur.
- A benchmarking toolset for Apache Spark, able to determine maximum throughput for file and TCP streaming, with tunable message size and CPU load per message – to explore this domain as a continuum.
- Recommendations for choices of frameworks and their integration with data sources, whilst highlighting some limitations of both frameworks especially for scientific computing use cases, and the microscopy image use case in particular; and discussion of challenges and techniques for benchmarking.

II. BACKGROUND: STREAM PROCESSING OF IMAGES IN THE HASTE PROJECT

Our main motivation for considering streaming applications where messages are relatively large binary objects (BLOBs) and where each processing task can be quite CPU intensive comes from our work on new smart cloud systems for prioritizing and organizing data from high-throughput (Wollman and Stuurman, 2007), and high-content imaging (HCI) experiments in which highly automated experimental setups are used to screen molecular libraries and assess the effects of compounds on cells using microscopy imaging. In the HASTE project[1] – a collaboration between Uppsala University, Stockholm University, Vironova AB and AstraZeneca, we are investigating methodology for near real-time filtering and control of image streams from such HCI platforms. Our goal of *online* analysis of the microscopy image stream allows both the quality of the images to analyzed (highlighting any issues with the equipment, sample preparation, etc.) as well as detection of characteristics (and changes) in the sample itself *during the experiment*.

[1] `http://haste.reserach.it.uu.se`

A recent masters thesis characterized potential streaming rates for a production HCI platform at one of the industry collaborators in the HASTE project, and found that current setups would be able to produce 38 frames/second with image sizes on the order of 10Mb (Lugnegård, 2018). Clearly, such image streams have different characteristics than many enterprise stream analytics applications. In particular:

- Messages are binary (not textual, JSON, XML, etc.)
- Messages are larger (order MBs, not bytes or KBs)
- The initial map phase can be computationally expensive, and perhaps dominate execution time.

Our goal is to create a general pipeline able to process streams with these characteristics (and image streams in particular) with an emphasis on spatial-temporal analysis. Furthermore, we hope to offer SCaaS – *Scientific Computing as a Service* with HASTE; to allow domain scientists to work with large datasets in an economically efficient way without needing to manage infrastructure and software themselves. The enterprise ASS framework has many of the features needed to build such a platform, in particular a rich set of APIs suitable for scientific applications, and demonstrated excellent performance for small messages with computationally light map tasks. However, it is not clear how well this performance translates to the regimes of interest to the HASTE project. This paper explores the performance of ASS for a wide range of application characteristics, and to compare and contrast it to to our research prototype streaming framework HarmonicIO; developed within the HASTE project.

III. EXISTING BENCHMARKING STUDIES

Several studies have investigated the performance of Spark, Flink and related platforms. However, these studies has tended to focus on small messages with textual content, with a focus on sorting, joining and other stream operations. Under an experimental setup modeling a typical enterprise stream processing pipeline (Chintapalli et al., 2016), Flink and Storm were found to have considerably lower latencies than Spark (owing to its micro-batching implementation), whilst Sparks throughput was significantly larger. The input was small JSON documents for which the initial processing – i.e. parsing, is a cheap operation, and integrated the stream processing frameworks under test with Kafka (Kreps et al., 2011) and Redis (Salvatore Sanfilippo, 2009) – this is advantageous in that it models a realistic enterprise system, but with each component having its own performance characteristics, it makes it difficult to get a sense of maximum performance of the streaming frameworks in isolation. In their study the data is preloaded into Kafka, the first operation is *reading* the data from Kafka. In our study, we also investigate ingress bottlenecks by writing and reading data to Kafka during the benchmarking, to get a full measurement of sustained throughput.

With an extension to the this benchmark (Grier, 2016), Spark was shown to outperform Flink in terms of throughput by a factor of 5, achieving frequencies of more than 60MHz. Again, as with previous studies, Kafka integration is used, and the focus is on small messages. Other studies follow a similar vein: (Qian et al., 2016) used small messages (60 bytes, 200 bytes), and lightweight pre-processing (i.e. 'map') operations: e.g. grepping and tokenizing strings, with an emphasis on common stream operations such as sorting and joining, (or reducing with add operations to get word counts).

Indeed, sorting is seen as something of a canonical benchmark for distributed stream processing. For example, Spark previously won the GraySort contest (Xin, 2014), where the frameworks ability to shuffle [2] data between worker nodes is exercised. Marcu et. al. (2016) offer a comparison of Flink and Spark on familiar BigData benchmarks (grepping, wordcount, and graph algorithms like PageRank (Page et al., 1999)), and give a good overview of performance optimizations in both frameworks. As with other studies, they benchmark for different algorithms, offering detailed recommendations for each.

To the authors knowledge there is no existing work benchmarking stream processing with Apache Spark, or related frameworks, with messages larger than a few KB, and with map stages which are more computationally expensive than tokenization, JSON parsing, etc.

HarmonicIO, which is a research prototype streaming framework with a peer-to-peer architecture, developed as part of the HASTE project specifically with scientific computing applications in mind, has been benchmarked exclusively for large messages (1-10MB) (Torruangwatthana et al., 2018).

Our approach in this paper differs from previous studies in that we look at performance characteristics as a continuum over the message size and map-function-cost parameter space. This allows us to reason about how variations in message size and processing load can be expected to affect performance. We compare Apache Spark (under various integration approaches) with HarmonicIO across this domain, to get an overview of performance characteristics as a function of these parameters.

IV. STREAM PROCESSING FRAMEWORKS

This section introduces the two frameworks selected for study in this paper, Apache Spark and HarmonicIO. Apache Spark competes with other frameworks such as Flink, Flume, Heron in offering high performance at scale, with features relating to data integrity (such as tunable replication), processing guarantees, fault tolerance, checkpointing, and so on – whereas HarmonicIO is a research prototype – much simpler in implementation, and is built around direct use of TCP sockets for direct, high-throughput P2P communication.

[2] https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations

A. Apache Spark

Apache Spark stands out as an attractive framework for scientific computing applications due to its many high-level APIs, such as built-in support for scalable machine learning.

Apache Spark was originally developed for batch operations, with a focus on in-memory caching of intermediate results to improve on the performance of Hadoop, where data is typically written to a distributed file system and read at each stage. Spark facilitates a more interactive user experience, allowing more ad-hoc analysis, something which was difficult with Hadoop. This smart caching behavior is combined with functionality to track the lineage of the calculations, to support deterministic re-calculation in the event of errors and node failure, together with a host of other features, built around the Resilient Distributed Dataset (RDD) (Zaharia et al., 2012). Spark can scale successfully to 1000s of nodes (Xin, 2014).

Spark Streaming was a later addition, leveraging the batch functionality for a streaming context by creating a new batch every few seconds (the batch interval). As with batch operations, data is further subdivided into partitions for distribution and scheduling. The newer *Direct DStream* integration approach with Kafka is used in this study. The Streaming API augments map/reduce operators (familiar from a batch processing context) with new functionality specifically for streams, such as windowing.

Stream processing pipelines are built from these operators, and the frameworks themselves manage the processing parallelization and data transfer – with various guarantees in relation to fault handling. These features, and the maturity of the frameworks themselves, together with the extent of their support community is attractive for our microscopy use case, and indeed scientific computing applications more widely.

B. HarmonicIO

HarmonicIO (Torruangwatthana et al., 2018) is a peer-to-peer distributed processing framework, intended for high throughput in the case of medium and large message sizes. It is designed to be highly elastic, and it separates the user's application from the framework through the use of docker containers. HarmonicIO's smart architecture will favor P2P message transfer, but fall back to using a queue buffer when necessary to absorb fluctuations in input or processing rate. The processing engines function according to a simple loop: pop a message from the master queue (if any exists) otherwise wait to receive a message directly from the streaming source over TCP; process it; and repeat. Per-node daemons aggregate availability information at the master, from where clients query the status of available processing engines. The master node also manages a message buffer queue.

Being a research prototype, it understandably lacks many features of more mature frameworks such as resilience, error handling, guaranteed delivery etc., and does not offer any *reduce* functionality. Yet the simplicity of the implementation makes it easily adoptable, and readily extensible; especially in a research context.

V. THEORETICAL BOUNDS ON PERFORMANCE

To illustrate the focus of previous benchmarking studies, and motivate our methodology – we can consider use cases within a broad domain according to message size and CPU cost of the map function. We can consider the theoretical performance of a 'ideal' stream processing framework which exhibits performance equal to the tightest bound, either network or CPU, with zero overhead; across this domain. The approach taken in this article is to investigate how close the frameworks under study perform across this parameter space. Figure 1 illustrates how we might expect such an 'ideal' framework to perform in different regimes of the parameter space.

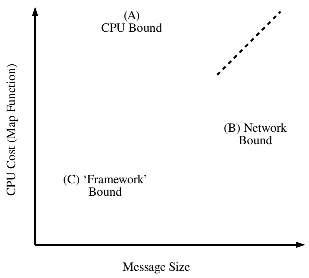

Fig. 1. Schematic of parameter space for this study, showing the processing cost of the map function, and the message size. Expected Performance Limiting Factors: (A) Expensive Map Function – CPU bound, (B) Large Files – network bound (C) Small Files, low-cost map function – in theory, frequency in this region should be high (unless constrained by the framework and integration).

A - Small message size, large processing cost - CPU Bound: For sufficiently large processing cost in relation to message size, performance will be CPU bound. Relative performance of the frameworks in this region will be determined by their ability to utilize CPU, and minimizing processing overheads. This regime would be typical of scientific computing applications involving e.g. a simulation step as part of the map stage.

B - Large message size, small processing cost - Network Bound: For sufficiently large message size, performance will be network bound. In this region, relative performance between frameworks will be determined by the network topology. P2P network topologies should perform well, whereas routing messages among the worker nodes will create additional network traffic which could impair performance. This regime would be typical for scientific computing applications involving relatively simple filtering operations on binary large objects (BLOBs), such as filtering of massive genomics datasets (Ausmees et al., 2018).

C - Small messages, small processing cost: In this regime processing frequency should be high. This region will expose any limitations on the absolute maximum message frequency for the particular framework configuration and thus may be 'framework bound'. Well-performing frameworks may be able to approach the network bounds; with very high frequencies. This regime would be typical for the type of enterprise applications studied in previous benchmarks such as analysis of social media streams.

VI. METHODOLOGY

We designed our study to evaluate performance with a simulator benchmarking application and streaming source, with tunable message size (100 bytes – 10Mb), and tunable CPU cost for processing each message (0 – 1 second per message). These two parameters allowed us to sample the performance of the studied streaming frameworks over a parameter space with settings ranging from highly CPU-intensive workloads to highly data-movement intensive use-cases. This domain captures both typical enterprise use cases, and scientific computing workloads.

The microscopy use case is a particular focus: message (image) sizes 1-10Mb, with a CPU cost profiled at around 100mS for very simple analysis (consistent with the previous study (Torruangwatthana et al., 2018)) – CPU cost would depend on the specific features being extracted, and could be significantly higher.

We measure maximum sustained frequency (i.e throughput) at each point (message size, CPU), for each of the three framework configurations explained below, so that we get an overview of performance limits of the various frameworks (and integrations) as a function of these input parameters.

A. Apache Spark Streaming Configurations

There are many ways to integrate input streams into Apache Spark Streaming – we discuss some of the commonly used approaches most likely to be adopted by new (or non-expert) users of Spark. We exclude more complex integration options such as the `RawInputDStream` – requiring more detailed knowledge of Spark. These input integrations were chosen:

Spark + TCP Socket: TCP sockets are a simple, universal mechanism, easy to integrate into existing applications, with minimal configuration. Being a low-level abstraction, it gives us more transparency when investigating how well ASS can utilize the network hardware.

Spark + Kakfa: Kafka is a stream processing framework in its own right: Kafka producers write messages onto one end of message queues, which are processed by Kafka *consumers*. These messages queues are organized by topic, and by partition – so that a single logical message queue can be distributed between nodes (offering resilience), with messages written to disk for durability. Kafka is commonly used in conjunction with Spark Streaming in enterprise contexts, to provide a resilient and durable buffer, allowing Spark applications to be restarted without interrupting the streaming source.

Spark + File Streaming: Under this approach, Spark will process new files in each batch interval, offering an ostensibly straightforward approach to integration. We configure an NFS share on the streaming source node. This should allow direct transfer of file contents from the streaming source node, to the processing machines – similar to the TCP socket approach used in HarmonicIO.

We do not use HDFS, it is an distributed filesystem intended replicated storage of very large (append only) files – many GBs, TBs, this makes it inappropriate for the files used in this study, which are at most 10Mb.

B. HarmonicIO

We choose HarmonicIO because it has a simple and intrinsically P2P architecture, whose APIs are a simple abstraction around TCP sockets (in contrast to Apache Spark). Its container-based architecture provides a convenient way for scientists to encapsulate complex (and often fragile) software with a variety of dependent libraries, models and datasets. Docker containers are a useful 'unit' of scientific computing code – likely to be relevant as we look ahead to creating the HASTE SCaaS platform.

VII. EXPERIMENTAL SETUP & TOOLS

To explore the (message size, CPU cost per message) space we developed benchmarking applications to run on Spark and HarmonicIO, able to process synthetic messages, and generate synthetic CPU load. These tools are publicly available at: `https://github.com/HASTE-project`.

Figure 2 shows the various pipeline configurations showing of HarmonicIO, and ASS with file and TCP streaming respectively. The arrows show the busy network communication.

For each setup, 6 stream Processing VMs were used, each with 8 VCPUs and 16GB RAM (1 master, 5 workers). For the streaming source VM we used a 1 VCPU, 2GB RAM instance. These resources are similar to the experimental setup of (Xin, 2014), where 40 cores were used. We used a tenant-based private network to be consistent with throughput. The maximum network bandwidth monitored using `iperf` was 1.4Gb/sec. Below, we describe the details of the experimental setup for each framework, and the approach used for determining the maximum frequency throughput:

A. Apache Spark

We created a *streaming source* application, supporting TCP and file-based streaming modes in ASS. The synthetic messages contain metadata describing the CPU load, so that both parameters (CPU load and message size) can be tuned via the streaming source application. To determine the maximum throughput, we adopt the approach of gradually increasing the message frequency (for a fixed (message_size, CPU cost) pair) until a bottleneck is detected somewhere in the pipeline, then iterating to accurately find the maximum. A monitoring and throttling tool was developed for this purpose. Listing 3 shows a simplified view of the algorithm used to determine the maximum. It also monitors and controls our streaming source application and the spark application, through a combination of REST APIs and log file analysis. The benchmarking tool saves progress so that it can be restarted over several days.

This process is repeated for (message_size, CPU cost) in a parameter sweep. We used a micro-batch interval of 150mS. Experimenting with other values had little impact on throughput. For the Spark File Streaming investigation, files are shared on the streaming server with NFS.

Maximum throughput is reached when a bottleneck occurs somewhere in the system. The throttling tool is able to detect various bottlenecks:

- Spark is taking too long to process messages (i.e. 'Total Delay' exceeds the batch interval – the key spark performance metric for stream processing). This can be due to a combination of network and/or CPU bound performance.
- There is a network bottleneck at the stream source, the source application reporting it cannot stream messages sufficiently.
- For file streaming, Spark is taking too long to perform a directory listing to decide which files to include in the next batch (this depends on the performance of NFS, the HDFS drivers, caches, and the way the file system is queried) streaming case).

B. HarmonicIO

For HarmonicIO, the maximum throughput is easier to determine, since the HarmonicIO streaming client is synchronous for message transfer (multiple client threads are used to obviate negative performance impact). In HarmonicIO, the maximum throughput is determined by measuring the time to stream and process a predefined number of messages for the given parameters.

We created a separate benchmarking application for HarmonicIO, which reads messages in our format. As with Spark, metadata describing the amount of CPU load is embedded in each message. Each worker hosted 8 Processing Engines that in total contributed 40 processing slots (one for each core).

VIII. RESULTS

The maximum frequencies achieved by each framework (and stream integration setup), according to message size and per-message CPU load, are show in figure 4, color-coded according to the best performing framework. A subset of these results is presented again in figure 5, where results for particular CPU loads are shown in relation to CPU and network-theoretical bounds. In figure 6, some of these results are shown normalized as a fraction of the theoretical bound. We first summarize the results for each setup, before further discussion in relation to the intended use cases, architectures, and relation to theoretical bounds in the proceeding discussion section (IX).

A. Apache Spark Streaming with TCP

This configuration achieves very high frequency when message size and CPU load are small, consistent with previous studies. For 100 byte messages without CPU load; the pipeline was able to process messages at frequencies approaching 320KHz, meaning around 1.6M messages were processed by Spark in a 5 second batch. This can seen in the extreme lower-left of figure 4, and the results shown in figure 5.A. But performance degraded rapidly for larger message sizes, and

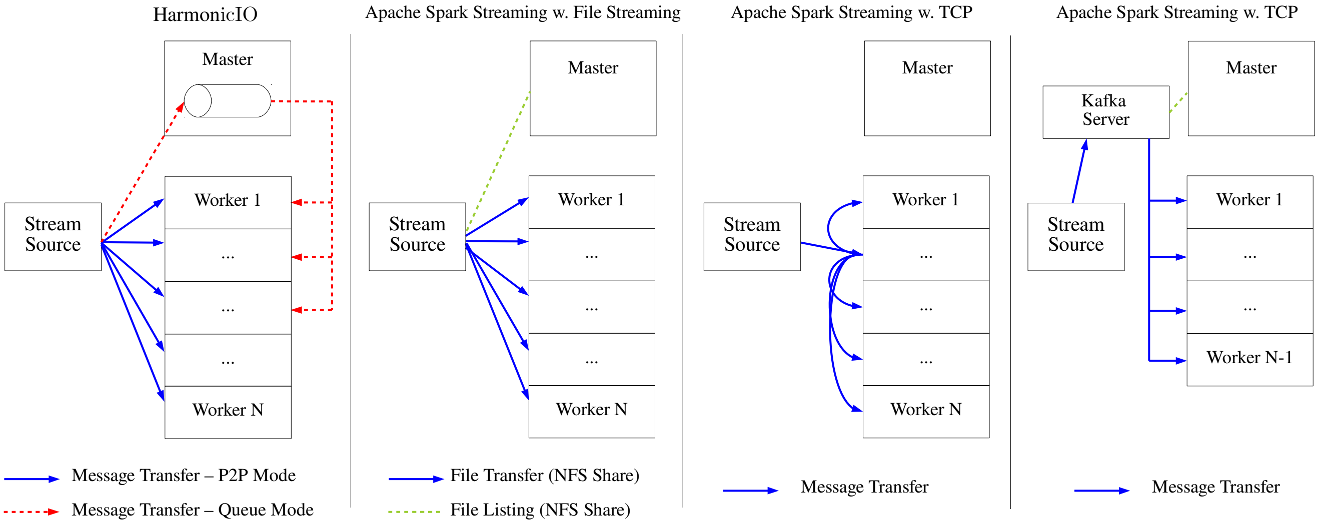

Fig. 2. Architecture (i.e. Network Topology) Comparison between the frameworks and stream intregratins – focusing on major network traffic flows. Note that under the Spark with TCP configuration, one of the worker nodes is selected as 'receiver' and manages the TCP connection to the streaming source. Note that neither the monitoring and throttling tool (which communicates with all components) is not shown, and nor is additional metadata traffic between nodes (for scheduling, etc.). Note the 2 colours showing adaptive queue- and P2P-based message processing in HarmonicIO.

under our benchmarks, it couldn't reliably handle messages larger than 10^5 bytes at any frequency.

B. Apache Spark Streaming with Kafka

With Kafka, the streaming pipeline performed well for messages less than 1MB, and CPU loads less than 0.1 second/message, as can be seen in region towards the bottom-left of figure 4. Outside this domain, performance degrades relative to other setups in this study.

C. Apache Spark Streaming with File Streaming

This configuration performed efficiently at low frequencies – in regions tightly constrained by network and CPU-theoretic bounds (figure 1 the top and right edges of figure 4. This is also shown in the plot in figure 5.C; showing 0.5 second/message CPU cost, and the results for higher message sizes in the other two plots.

D. HarmonicIO

Figure 4 shows that it was the best performing framework for a large region of our study domain, for medium-sized messages (larger than 1.0MB), and/or CPU loads higher than 0.05 seconds/message. It broadly matched the performance of file-based streaming with Apache Spark for larger messages and higher CPU loads (the top and right edges of Figure 4), and the results for larger message sizes shown in figure 5.

IX. Discussion

A. Performance: Message Size & CPU Load

Building on the results summary in the preceding section, this section compares performance among framework configurations, and with theoretical bounds, in the different regions of our study domain. These regions represent different use cases, and each framework setup shows good performance for their own intended use case/region.

There are both network- and CPU-bounds on the overall message throughput. Theoretically, maximum message throughput is always the lowest of network-bound and CPU-bound throughput, which are inversely proportional to message size and CPU cost size respectively (constant terms being network speed, and number of CPU cores; respectively). In practice, the relative performance of different frameworks (and stream integrations) depend on these same parameters, and we can see from figure 4 that all frameworks have specific, well-defined regions where they each perform the best. We discuss these regions in turn, moving outwards from the origin of figure 4.

Close to the origin, theoretical maximum message throughput is high, figure 5.A shows that for the smallest of messages, Spark with TCP streaming is able to outperform Kafka. It seems that for extremely small messages, the overheads of Kafka message handling mean it can be outperformed by more lightweight (and less resilient) handling of batches of messages in Spark. However, as figure 5.A shows, for slightly larger message sizes (but less than 1MB), Kafka slightly outperforms Spark with TCP (it is better optimized for handling messages in this size range - its intended use case). For Kafka, Under the direct DStream integration with Spark, messages are transferred directly from the Kafka server to Spark workers. Yet, the Kafka server itself; having been deployed on its own machine, has theoretical network bounded throughput at half the network link speed (half the bandwidth for incoming messages, half for outgoing). With Apache Spark, we see a

```
def find_max_f(msize, cpu_cost):
    max_known_ok_f <- 0
    min_known_not_ok_f <- null
    f <- f_last_run or default(msize, cpu_cost)
    while true:
        metrics = [spark_metrics(), ssrc_metrics()]
        switch heuristics(metrics):
        case sustained_load_ok:
            f <- throttle_up(metrics, f)
        case too_much_load:
            f <- throttle_down(f)
        case wait_and_see:
            pass
     sleep

def throttle_up(metrics, f):
    max_known_ok_f <- f
    if min_known_not_ok_f == null:
        load = estimate_fraction_max_load(metrics)
        if load < 0.01: new_f <- f * 10
        elif load < 0.1: new_f <- f * 5
        elif load < 0.5: new_f <- int(f * 1.10)
        elif load < * 0.8: new_f <- int(f * 1.05)
        else: new_f <- int(f * 1.05)
        if f == new_f:
            new_f <- f + 1
        return new_f
    else:
        return find_midpoint_or_done()

def throttle_down(f):
    min_known_not_ok_f <- f
    return find_midpoint_or_done()

def find_midpoint_or_done():
    if max_known_ok_f + 1 >= min_known_not_ok_f:
        done(max_known_ok_f);
    else:
        return int(mean(max_known_ok_f,
                        min_known_not_ok_f)
```

Fig. 3. Listing: Monitoring and Throttling Algorithm (Simplified)

similar effect, with the spark worker nominated as receiver performing an analogous role. Consequently, figure 5.A shows that neither Kafka nor Spark with TCP can approach the theoretical network bound, as *is* the case with other frameworks in some regions. This is consistent with the overview of the network topology shown in figure 2.

Moving further from the origin, into the region with CPU cost of 0.2-0.5 secs/message and/or medium size (1-10MB) – HarmonicIO performs better than the Spark configurations under this study. It exhibits good performance in most scenarios – transferring messages over 'raw TCP makes means it is able to make very good use of bandwidth (when the messages are larger than 1MB or so); similarly, when there is considerable CPU load, its simplicity obviates spending CPU time on serialization, and passing messages multiple times among nodes. For large messages, and heavy CPU loads, this configuration is able to approach closely to the network and CPU bounds (when the associated message frequency is low) – its able to make good, cost-effective use of the hardware.

For the very largest messages, and the highest per-message CPU loads, the network and CPU bounds are very tight, and overall frequencies are very low (double digits). In these

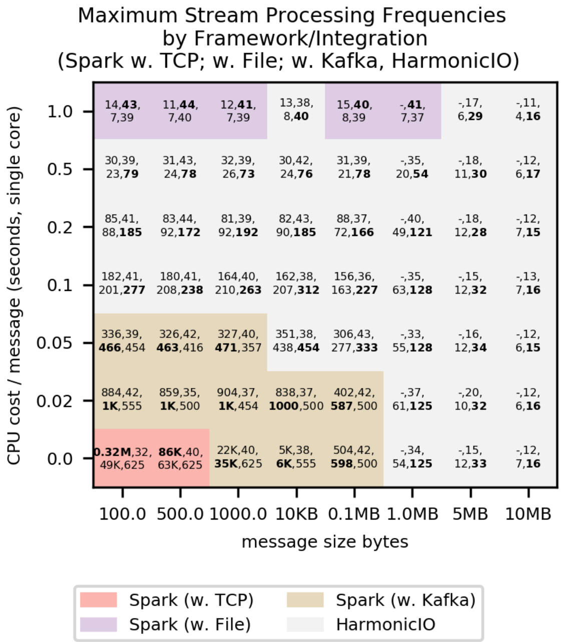

Fig. 4. Performance of Apache Spark (under TCP, File streaming, and Kafka integrations), and HarmonicIO, by maximum message throughput frequency over the domain under study. Under each setting, the highest frequency is shown in bold, and the figure is color-coded according to which framework/integration was able to achieve the best frequency.

regions HarmonicIO performance is matched (or exceeded) by Spark with File Streaming. Both approaches are able to tightly approach the network and CPU theoretic bounds – as shown in figure 5, especially the network bound with larger messages. Under these conditions, Spark with file streaming performs slightly better in CPU-bound cases, with HarmonicIO performing slightly better in network-bound use cases (c.f. figure 1).

B. *Performance, Frameworks & Architecture*

This section draws together previous discussions, to summarize of the strengths and weaknesses of each framework configuration, their applicability to different use cases, with an emphasis on internals, and the network topology/architecture in each case.

Spark with TCP has is able to process messages at very high frequencies in a narrow region; yet this performance is highly sensitive to message size and CPU load. Forwarding messages between workers yields allows message replication (and hence resilience), at a cost overall message throughput. With heavy CPU loads, we see reduced performance of Spark with TCP relative to the other configurations – effectively fewer cores being utilized for processing. We speculate that this is due to processing overheads associated with the message forwarding. Figure 5 shows a performance impact of consistent proportion (relative to the CPU theoretic bound), in all CPU-bound cases. As discussed, this configuration is completely unsuitable for

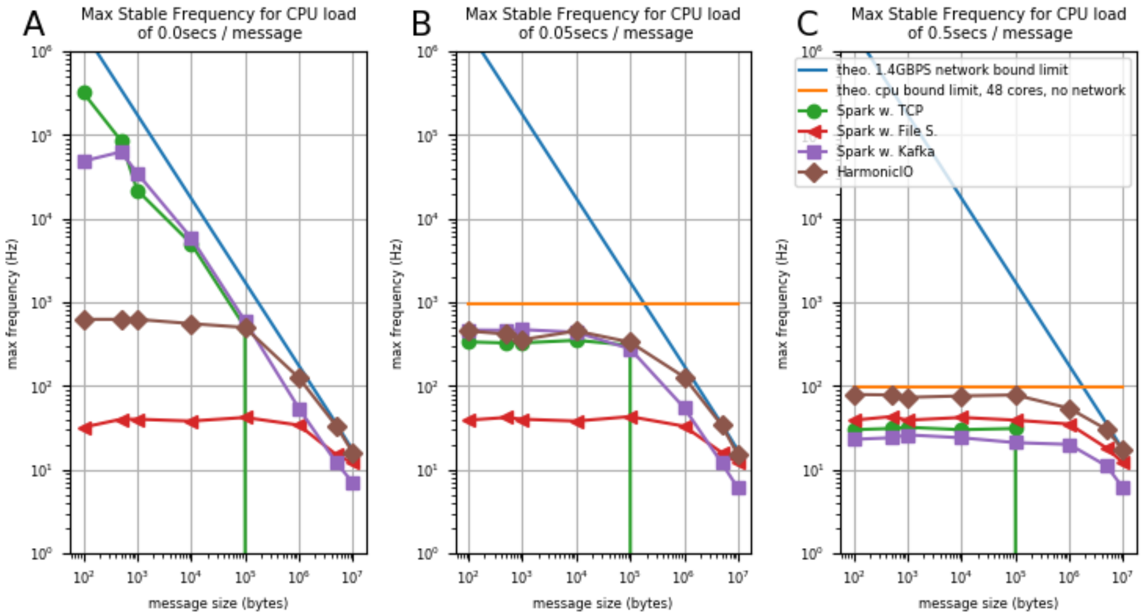

Fig. 5. Maximum stream processing frequencies for Spark (with TCP, File Streaming, Kafka) and HarmonicIO under varying message size; for selected choices of CPU cost/message.

message sizes larger than a few 100KB; being outside its intended use case.

Kafka is the best performing framework for slightly larger messages and CPU loads of less than 0.05secs/message. For these low CPU loads, using a few cores to run Kafka makes for better overall throughput, because Kafka is highly optimized for handling these small messages - the Direct DStream integration with Spark means that messages are being transferred directly from the Kafka server to the Spark workers (see figure 2).

However, for larger messages (1-10MB), Kafka performs poorly (this is unsurprising, this region being outside the intended use case - Kafka is not intended as a file server). When the CPU load is more considerable, for our small cluster (48 cores), the overheads of Kafka (and Spark) reduce the overall CPU core utility – there are simply less cores available for message processing (which becomes the bottleneck at higher load, see figure 1). In these cases, performance is met or exceeded by HarmonicIO – messages are transferred directly from the streaming source, so more cores are available more of the time for processing message content.

The results for Spark with File Streaming are quite different. The implementation polls the filesystem for new files, which no doubt works well for quasi-batch jobs at low polling frequencies – order of minutes, with a small number of large files (order GBs, TBs, intended use cases for HDFS). However, for large numbers of much smaller files (MBs, KBs), this mechanism performs poorly. For these smaller messages, network-bound throughput corresponds to message frequencies of, say, several KHz (see figure 5) - at frequencies, (outside the intended use case of this integration) a filesystem polling-based implementation is cumbersome. Especially at low polling intervals required for low-latency applications (like quasi-real-time control systems). This makes it a bottleneck at high frequencies, especially over NFS, in our configuration. There are a variety of discussions and issues related to (distributed) file system integration (e.g. HDFS, object stores) and Spark under small file use cases, vaguely discussed as the ‘small file(s) problem’ (Pointer, 2015). For example, implementation of the `FileInputDStream` is not intended to handle the deletion of files during streaming, requiring workarounds [3]. Message frequency is bounded at around 40Hz, so for our 48 core cluster, if the CPU cost per message is lower than 0.5seconds/message, this becomes the bottleneck – figure 5.A shows performance for File Streaming well below the theoretical maximum.

However, in terms of data throughput, the integration is lightweight, and using an NFS share on the streaming source machine allows message content to be transferred directly to where its processed on the Spark workers, see figure 2. Consequently, under loads where the theoretical bound on message frequency is tight (both network bound and CPU bound - see figure 1), message frequency is low double-digits. Under these conditions, integration is very efficient, and is the best performing framework at very high CPU loads (near the top of figure 4). Each worker directly fetches the files it needs

[3] This is a known issue: `https://issues.apache.org/jira/browse/SPARK-20568`

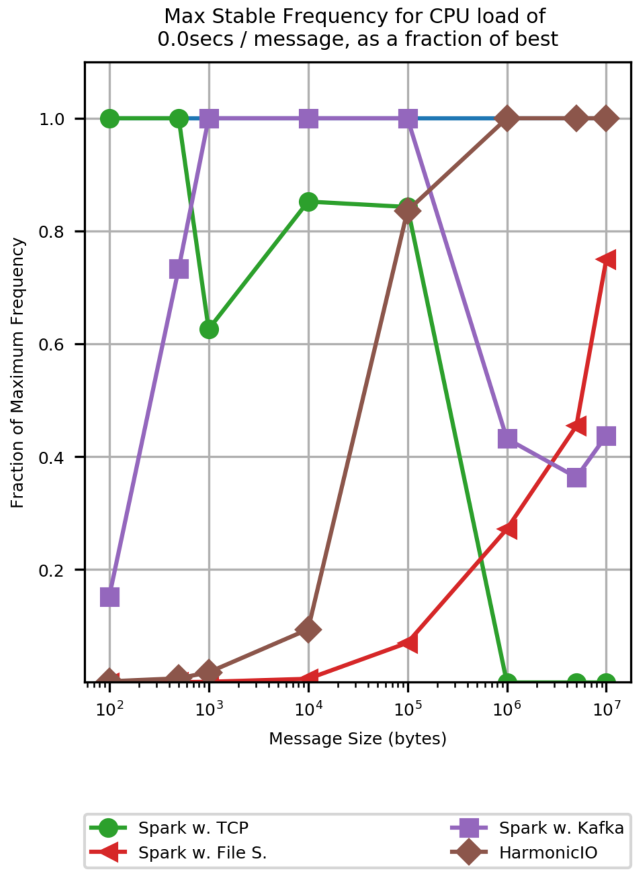

Fig. 6. Maximum frequency by message size, for Spark (with TCP, File Streaming, Kafka) and HarmonicIO under varying message size; normalized as a fraction of the best performing framework for the particular parameter values.

from the streaming source machine, this makes for a good use of network bandwidth, and robustly handles messages of arbitrary size. CPU cores (and network bandwidth) are not used for forwarding (and serializing) messages, and are free to process message content.

HarmonicIO is the best performing framework (in terms of maximum frequency) over much of the parameter space under study, as shown in figure 4. It's peer-to-peer network topology (see figure 2) explains its ability to make good use of the network infrastructure, it has excellent performance in the network bound case, and its simplicity explains how this generalizes well over the rest of the domain. Consequently, in scenarios handling message sizes in 1-10MB range, or computationally intensive message analysis on medium clusters – i.e. scientific computing applications, including low-latency microscopy image analysis – overall throughput can be better using HarmonicIO. Plots in figure 5 show HarmonicIO is able to make excellent use of available network bandwidth for larger message sizes especially.

As is the case with Spark integrated with File Streaming, the bottleneck is not message transfer, but matching workers to messages (or files on disk): in the case of HarmonicIO, the master must be queried for each message to find an available processing slot. In practice, HarmonicIO achieved an absolute message transfer rate of 625Hz, for the smallest, lightest messages (see figure 4) – meaning in this domain it was easily beaten by Kafka and Spark with TCP stream integration. Figure 5.A clearly shows this frequency bound – making it unsuitable for traditional enterprise use cases with high message frequencies.

The implementation of HarmonicIO is well optimized for network transfer for larger messages, but has limited compute optimizations (lacking any asynchronous IO optimizations, for example). Not being constrained by a filesystem polling implementation, it is able to match the performance of Spark with File Streaming at low frequencies in tightly-bounded regions, yet also out perform it at higher message frequencies in less-bounded regions.

HarmonicIO's dynamic P2P architecture switching between P2P and queuing mode is a simple approach to making effective use of network bandwidth (at message frequencies up to 625Hz): if all workers are busy, messages are sent to the queue, keeping the link to the streaming source fully utilized.

In summary, for very small messages (and lightweight processing), Spark (and Kafka) perform well (as does Spark with TCP streaming), and similarly for quasi-batch processing of very large files (at large polling intervals), Spark with file streaming integration performs well. These are the two typical enterprise stream analytics contexts: low-latency, high-frequency processing of small messages, and high-latency quasi-batch file-based processing of large message batches, log files, archives, etc.

This leaves a middle region – medium sized messages, and CPU intensive processing of small messages: conditions typical of many scientific computing use cases, including our motivating microscopy image processing use case. In this region, HarmonicIO is able to outperform the Spark streaming approaches benchmarked in this study. There are clearly caveats for this claim, discussed in the conclusions (section X).

C. Features, Challenges & Recommendations

This section briefly discusses the experiences with each framework setup, implementation issues, and challenges of measuring performance.

Whilst HarmonicIO offers container isolation, and a simple and extensible code base, it lacks functionality for replication and fault handling with no delivery guarantees – messages are lost in the case of node failure. The current implementation has no support for reduce operations. The simplicity of HarmonicIO means it has only a handful of configuration options.

Apache Spark's features are well-documented in existing literature, and include rich Map/Reduce APIs, a variety of stream operators, and various stream integration approaches. The resilience and fault-handling of Spark make it attractive for online analysis of data streams, in scientific computing contexts, especially valuable, sensor-based data (as opposed to simulation output): the core RDD (resilient distributed dataset) records the functions applied to the input data, in the case of error or data loss (due to node failure), the same results can be deterministically re-computed, transparently from the user's perspective. Such functionality also allows transparent caching of intermediate results (without concerns about managing the validity of cached data).

But, these features create their own complexity, especially regarding configuration. In academic research environments, where scientists are developing their own software, the associated learning curve could be an issue. A key limitation of Spark, also relevant to scientific computing, is that the Python API is a restricted subset of the underlying Java/Scala API. HarmonicIO by contrast is implemented entirely in Python.

X. CONCLUSIONS

This study has confirmed Spark's excellent performance, consistent with earlier studies, albeit for use cases with small message size, and low message processing cost – and quasi-batch loads at low frequencies. But, we also find that these 'islands' of excellent performance do not generalize well across the wider domain of use cases we studied. In particular, it was difficult for Spark to achieve good performance in the 1MB to 10MB message size range (typical of microscopy image analysis, for example), using the stream integration approaches we studied.

By contrast, HarmonicIO performing well in this region, with good hardware utilization at low frequencies – whilst not matching Spark for maximum frequency for the small message/cheap map function use case. Arguably, its simplicity makes its performance less sensitive to configuration settings, and parameters of the application load.

Regarding features, as discussed, Spark boasts with enterprise-grade resilience and durability, and a host of stream processing operators – whilst HarmonicIO lacks all but basic support in these areas. Naturally, the disparity in features and performance characteristics is reflected in the respective implementations, and architectures (i.e. network topology).

Indeed, replicating data to secure against loss in the case of node failure requires copying it between nodes, limiting maximum flow in cluster computing contexts which are potentially network bound. Features like these also require additional processing, similarly impacting performance in CPU-bound cases. HarmonicIO, lacking these safeguards and features, is much simpler, and can consequently approach closer to the theoretical performance bounds, across much of the study domain. Hence, robust performance is traded off against the non-functional features provided by each framework.

The lack of fault tolerance and associated safeguards in HarmonicIO restrict the suitability of the frameworks to particular contexts – and affect the complexity of configuration. There is a similar trade-off with the APIs and more functional requirements for each framework: Apache Spark with a rich array of stream operators, whilst HarmonicIO providing a low-level API, thinly wrapping a TCP socket – making migration of existing applications much easier.

In some scientific computing contexts, CPU utility is more important than resilience and fault-tolerance. Considering a simulation running over several hours or days: hardware utility is important, as it has a direct bearing on financial cost. In many cases, the entire experiment can be repeated in the case of node failure, assuming the simulation is deterministic (yet not without cost). However, considering use cases where data is collected from sensors, and analyzed, the data itself is valuable (the physical experiments could be much more costly to reproduce) – Apache Spark's replication of the data, and resilient handling of errors make it attractive in these use cases, aside from performance. In the latter cases, some analysis at the cloud edge may be needed, depending on the parameters of the input stream, necessitating a fog computing approach (Bonomi et al., 2012).

But, performance only needs to be adequate for the intended application – even without any optimization, Sparks file streaming seemingly poor 60Hz maximum file ingress rate (under our configuration) will be adequate for many use cases (and when files represented data aggregated over minutes or hours, it is no issue at all). HarmonicIO's maximum throughput of around 700Hz would be adequate for many microscopy use cases, for example.

Overall, message size, processing cost, required processing frequency, and streaming source *reliability*[4] are all crucial consideration when selecting frameworks (and stream integration approaches).

The complexity of Apache Spark means it can arguably be viewed as a SDK for the development of stream processing pipelines – requiring expert knowledge (and some experimentation) to optimize performance for particular use cases. As discussed only a subset of the Spark stream integrations have been studied in this paper, and even for those, a variety of different configurations and topologies are possible. It is highly likely that with sufficient tuning, configuration, and selection of other stream integrations (and their configuration, tuning, and integration with streaming applications) Spark could match (or exceed) performance of HarmonicIO over more of the domain than it does in this study.

The key argument is not about absolute performance: it is that *out of the box* HarmonicIO outperforms many typical (or easily adopted) stream integrations with Spark in the 'middle' region - atypical of enterprise use cases for which Spark (and common stream integrations) are intended, and that this performance is robust over a greater domain (of message size, processing cost). This is a contrast to highly-tuned Spark deployments, which are likely to give high performance in perhaps narrower domains. HarmonicIO is sometimes able to achieve better (and more robust) performance because of its simplicity by sacrificing extensive functionality, making it

[4]To use the Spark terminology: `https://spark.apache.org/docs/2.3.0/streaming-programming-guide.html#receiver-reliability`

much more lightweight – and able to achieve better hardware utilization in some regions.

This paper has quantified the use cases where each framework configuration can perform at or near theoretical bounds, and where each configuration may perform poorly. Our findings should be applicable to other stream processing use cases with large messages and non-trivial map operations, including other IoT applications where the size of each message is large – anything involving sensor arrays (for example infrared, acoustic, RF – and of course, cameras). Both HarmonicIO and Spark allow rapid development of stream processing pipelines for such applications.

XI. FUTURE WORK

There are some 'quick wins' for both frameworks to make them more suitable for scientific computing. HarmonicIO would benefit from more of Spark's resilience, and fault tolerance, for example, some basic functionality to make the cluster more highly-available, and improve performance for small messages through simple batching strategies. This should be possible without compromising HarmonicIO's key strengths: robust performance, minimal configuration, and overall simplicity.

Ideally the performance of the integrations could be made a little more robust across a wider range of message sizes, for those that we investigated: TCP integration could support larger messages, and some optimizations could be made to improve file streaming performance at high ingress frequencies. Whilst it might be possible to widen the margins of good performance for these integration strategies there is clearly no one-size-fits-all approach for either framework.

Our goal is to develop HASTE into a SCaaS platform suitable for both edge- and cluster- (and indeed fog-) based scientific computing deployments. The challenge is to combine the best features of Spark (and related frameworks) into a single platform, or indeed investigate whether this is feasible.

Under such a system, the user would express the trade-offs between say, performance, durability and economic cost that they desire. Our key challenge is to engineer a cloud system which is able to adapt to these requirements in in a smart and dynamic way, going beyond, say, the merely tuning of replication policies, so that instead the architecture itself is radically different depending on use case and the desired trade offs. For example, we could approach the problem 'from above' and build meta-frameworks, and stream processing middleware, routing messages to different frameworks, or 'from below' – selecting integration with different underlying systems (and reconfiguring them) depending on the characteristics of the application load and the desired trade offs for the non-functional requirements.

We feel this represents an open problem in cloud computing, and a key focus for our future research.

XII. ACKNOWLEDGEMENTS

This work is funded by the Swedish Foundation for Strategic Research (SSF) under award no. BD15-0008, and the eSSENCE strategic collaboration on eScience. Computational resources were provided by the Swedish National Infrastructure for Computing via the SNIC Science Cloud (SSC) (Toor et al., 2017), an OpenStack-based community cloud for Swedish academia.

REFERENCES

(2011). Apache Hadoop. http://hadoop.apache.org/.

Apache Flume (2016). Apache Flume. https://flume.apache.org/.

Ausmees, K., John, A., Toor, S. Z., Hellander, A., and Nettelblad, C. (2018). BAMSI: A multi-cloud service for scalable distributed filtering of massive genome data. *BMC Bioinformatics*, 19(1):240.

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA. ACM.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., and Peng, B. J. (2016). Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE.

Grier, J. (2016). Extending the Yahoo! Streaming Benchmark.

Kreps, J., Narkhede, N., Rao, J., and others (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7.

Lugnegård, L. (2018). *Building a High Throughput Microscope Simulator Using the Apache Kafka Streaming Framework*. PhD thesis, Uppsala University.

Marcu, O. C., Costan, A., Antoniu, G., and Pérez-Hernández, M. S. (2016). Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

Pointer, I. (2015). 5 things we hate about Spark. https://www.infoworld.com/article/3004460.

Qian, S., Wu, G., Huang, J., and Das, T. (2016). Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, pages 592–598.

Salvatore Sanfilippo (2009). Redis. https://redis.io/.

Toor, S., Lindberg, M., Falman, I., Vallin, A., Mohill, O., Freyhult, P., Nilsson, L., Agback, M., Viklund, L., Zazzik, H., and others (2017). SNIC Science Cloud (SSC): A National-Scale Cloud Infrastructure for

Swedish Academia. In *E-Science (e-Science), 2017 IEEE 13th International Conference On*, pages 219–227. IEEE.

Torruangwatthana, P., Wieslander, H., Blamey, B., Hellander, A., and Toor, S. (2018). HarmonicIO: Scalable data stream processing for scientific datasets. In {*IEEE*} *Services*.

Wollman, R. and Stuurman, N. (2007). High throughput microscopy: From raw images to discoveries. *Journal of Cell Science*, 120(21):3715–3722.

Xin, R. (2014). Apache Spark the fastest open source engine for sorting a petabyte.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). 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*, NSDI'12, pages 2–2, San Jose, CA. USENIX Association.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65.