HarmonicIO: Scalable Data Stream Processing for Scientific Datasets

Preechakorn Torruangwatthana*, Håkan Wieslander†, Ben Blamey‡, Andreas Hellander‡ and Salman Toor‡
* Department of Information Technology, Division of Computer Science, Uppsala University, Sweden
† Department of Information Technology, Division of Visual Information and Interaction, Uppsala University, Sweden
‡ Department of Information Technology, Division of Scientific Computing, Uppsala University, Sweden
Email: Preechakorn.Torruangwatthana.1715@student.uu.se,
{Hakan.Wieslander, Ben.Blamey, Andreas.Hellander, Salman.Toor}@it.uu.se

***Abstract*—Many streaming frameworks have been introduced to deal with the needs for online analysis of massive datasets. Scientific applications often require significant changes to make them compatible with these frameworks. Other issues include tight coupling with the underlying infrastructure, shared computing environment, static topology settings, and complex configuration. In this article we present HarmonicIO, a lightweight streaming framework specialized for scientific datasets. It boasts a smart dynamic architecture, is highly elastic, and enforces a clear separation between framework components and application execution environment using container technology.**

***Keywords*-Data stream processing; Streaming engines; Data analysis; Cloud platforms; Scalability; Efficiency.**

I. INTRODUCTION

Data processing frameworks have a history that starts with conventional Database Management Systems (DBMS) to more recent frameworks based on streaming engines (Gama and Rodrigues, 2007). The former were highly effective when the entire datasets could be stored in available, clustered resources, and exact answers to queries are possible with that technology. Massive datasets need very large amounts of storage and memory resources for their processing, calling for new strategies that provide approximate but fast answers using commodity hardware. Adopting existing frameworks (Apache Spark (Zaharia et al., 2016), Heron (Kulkarni et al., 2015), StreamFlex (Spring et al., 2007)) in a scientific computing context can present challenges of configuration complexity; especially in relation to performance, messages being routed through multiple nodes (impacting performance of network-bound applications), and close integration between the user's application and the framework APIs (creating difficulties migrating scientific applications - often with complex and fragile dependencies). Dynamic data rates present a further challenge, the typical strategy of using a (say, Kafka, Kreps et al., 2011) queue upstream of the processing cluster requires additional hardware (reducing overall utility), can introduce performance bottlenecks (especially in scientific use cases involving large datasets), and creates additional deployment and configuration complexity. Elastic scaling according to the workload and shared processing engines with predefined software libraries and tools is also noted as a challenge (Kulkarni et al., 2015)

A wide range of scientific applications need similar streaming mechanisms but the datasets have different characteristics. For example, applications dealing with images and files with specialized formats have defined internal structures. The individual objects in the scientific datasets are autonomous and often have significant sizes (10s, 100s of KBs or MBs). Whereas most of the studies based on conventional streaming frameworks present large homogeneous datasets base on very small messages (10s, 100s of bytes). We have designed and implemented a prototype framework that capitalizes on those characteristics to achieve highly scalable, parallel streaming by considering each object as complete and autonomous in the dataset.

In this article, we present HarmonicIO, a framework designed to address the challenges mentioned above for data stream processing on scientific datasets. The proposed framework manages dynamic data rates. It is a lightweight framework, designed on several key principles: 1. simplicity, 2. a smart, dynamic architecture favoring peer-to-peer data transfer between data sources and processing engines – buffering in a queue only when necessary, 3. a clear separation between the streaming framework and the application execution environment by using Docker containers (Merkel, 2014) to provide a custom execution environment.

II. HARMONICIO FRAMEWORK

A. Framework Design

HarmonicIO is a smart P2P stream processing framework: data is streamed directly from the sources to processing engines (running inside containers on worker nodes) - if there is too much data to process, messages are queued at the *Master* node (to avoid back pressure). Adopting a P2P architecture allows for high throughput, especially for medium and large messages. HarmonicIO is also a container manager, the *Master* maintains a list of processing engine containers; and their status, and is responsible for message distribution (in both the P2P and queue case), and so that new processing engine containers can be started or terminated on the worker nodes as necessary.

Worker nodes host multiple Docker containers, application-specific data processing engines, with a listening socket daemons for HarmonicIO. The encapsulation offered by containerization is important for scientific applications, since it allows convenient and framework-free application development. Figure 1 illustrates the overall design of the framework. The capabilities of each component are discussed in section II-B.

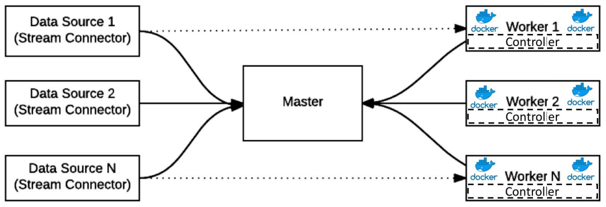
Data Source 1 (Stream Connector)
Data Source 2 (Stream Connector)
Data Source N (Stream Connector)
Master
Worker 1
Controller
Worker 2
Controller
Worker N
Controller
docker

Fig. 1. The design consists of three main components: Stream Connectors (client nodes), a Master node, and Workers (hosting processing engines within docker containers). The solid arrows show the framework communication whereas dotted arrows represent the actual peer-to-peer data transfers.

B. System Components

1) Worker Nodes: Each worker node runs a single *controller*, which manages multiple *Processing Engines* running inside docker containers. *Processing Engines* are user-defined applications for processing messages, built around a lightweight socket listening daemon (*HarmonicPE*) which receives messages. The HarmonicPE socket listening daemon is a very lightweight dependency; and containers built around it can be used for 'plug and play' sharing of scientific code, decoupled from the streaming source application and the other framework components. Internally, the daemon operates continuously according to a loop: it requests the next message (if any) from the *master* queue, or else waits to receive a P2P message via its TCP socket. On receiving this message, the message is processed by the user's code.

The worker node *Controller* manages the processing engine containers running on that node. It aggregates information about the availability of processing engine containers and reports to the *master* (over a REST API); and can start and stop processing engine containers as requested. Figure 2 shows the architecture of the controller and *Processing Engine* on a worker node.

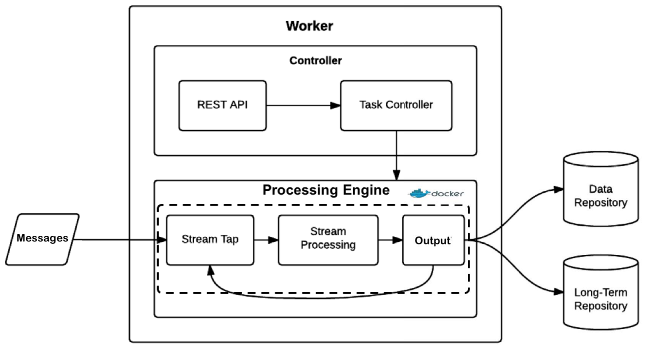

Fig. 2. The Design of worker node. The controller is responsible to initiate and monitor the status of *Processing Engines*.

2) Master Node: The master node performs a number of functions: it maintains a list of processing engine containers and their availability status (and responds to queries from the stream connector accordingly), it maintains a message queue – used only in the case of backlogs (and responds to requests from processing engines to retrieve messages), and handles requests from clients to instantiate additional processing engine containers, according to a heuristic described in Section II-C.

3) Stream Connector: The stream connector resides at the stream sources, and is the entry point into the framework. Typical operation is as follows: The stream connector requests the socket address of an available processing engine container from the *Master* (via a REST API), and the master responds with a socket address either for an available processing engine container, or the message queue. The message is sent to a *processing engine* container, or the message queue as appropriate. The *Stream Connector* is a Python package, and is incorporated directly into the user application generating the stream.

C. Task and Resource Allocation

Incoming messages are assigned to worker nodes with the least number of idle processing engines, subject to a per-worker defined maximum – this is to maximize the node utilization, so that idle worker nodes can be removed from the cluster, consequently reducing the overall cost.

III. System Features

- **Smart Architecture:** A message backlog queue is used as a buffer to absorb fluctuations in source and/or processing rates, without blocking streaming sources. This queue is only used when necessary, favoring P2P message transfer.
- **High Throughput:** HarmonicIO uses P2P transfer whenever possible, and uses data-parallelism by having multiple processing engines on multiple worker nodes.
- **Elasticity:** Processing engines are very lightweight, and workers hold minimum state. The message queue can act as a buffer while scaling up the system.
- **Custom Environment:** The user's application is built within a custom environment, encapsulated inside a Docker container, with very minimal dependency on the framework. This makes it easier to migrate existing scientific applications (often with complex dependencies) to the cloud.
- **Simplicity:** HarmonicIO requires almost zero configuration, and whilst it naturally lacks many optimizations of mature frameworks, it is robust; and is readily extensible – making it an ideal platform for research prototyping.

IV. Results and Discussion

We evaluate HarmonicIO in two use cases. First, we study the framework's performance and scalability for generic workloads. Second, we consider a case-study using HarmonicIO for a conventional scientific application. In all our experiments, we used the SNIC Science Cloud (SSC) OpenStack-based IaaS

(Toor et al., 2017), for Swedish academia. The experimental setup was based on Ubuntu 16.04 virtual machines (VMs); each with 8 virtual cores, 16GB memory. The maximum bandwidth measured between two VMs using *iperf* was 4.53Gb/sec. We deployed HarmonicIO using Ansible.

A. *Performance and Scalability*

1) Dynamic scaling: Figure 3 illustrates the capability of HarmonicIO to scale dynamically. The experiment had three stages. First (time points 1-50), the framework was deployed with 10 processing engines. As the number of messages increased, we added more processing engines. Eventually, the framework managed to meet the workload with 30 processing engines and the message queue length went down to almost zero (around time point 60).

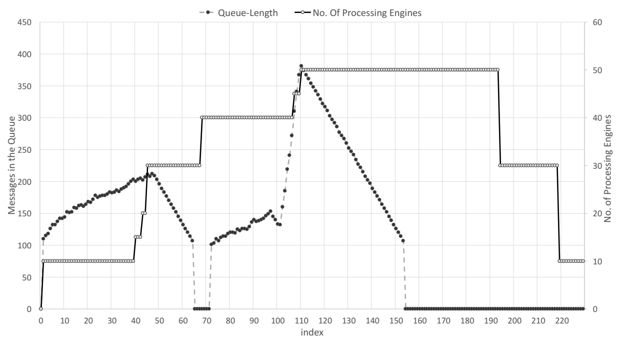

Fig. 3. Dynamic load management using different number of workers. The x-axis shows the time index and the two y-axes presents the messaging queue and the number of running containers in the framework.

In the second stage (time points 70 - 160), an even higher amount of requests were received in the framework. The load on the system was again managed by adding more processing engines and around time point 160, the queue length was again close to zero. The final stage (time points 160-220) illustrates the scenario where resources are currently over-provisioned in the framework and the flexibility to release unused resources at runtime. HarmonicIO can manage workloads using limited resources with the help of the message queue.

2) Performance for small messages: HarmonicIO is specialized for scientific datasets where each data object is self-contained and often has a significant size (100s of KBs, MBs, GBs). This is in contrast to conventional streaming use-cases where the object sizes are typically small (10s or 100s of bytes). Communication overhead impact performance while working with very small messages. Figure 4 presents the framework's response to relativity small data objects. For this experiment, each worker had 5 processing engines each. We transfered 100,000 data objects for each data point. We see that for small message sizes (< 100KBs) metadata communication dominates, and adding additional processing engines does little to improve performance. In contrast, with 500KB and 1000KB objects, we see improvements in throughput when adding more processing engines.

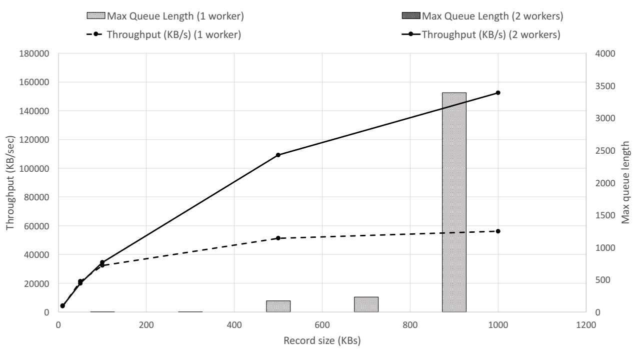

Fig. 4. Processing throughput for small data objects, for 1 and 2 workers.

B. *Florescent Cell Imaging Experiment*

HarmonicIO was evaluated in a real scientific computing use case: the online analysis of a microscopy image stream. A new class of drugs, based on genes coding for a therapeutic protein, can be delivered using lipid nano particles (LNPs). A range of different LNP formulations have to be investigated (Kauffman et al., 2015) - the delivery process is visualized by time-lapse microscopy, and successful protein production is modeled using the green florescent protein (GFP). The experiments are conducted on cells cultured in multi-well plates with a different LNP formulation added to each well. With successful delivery and uptake, the cell starts to produce GFP, which can be observed by fluorescence microscopy, with use of a filter (Ettinger and Wittmann, 2014). Time-lapse generates large amounts of data, so it is desirable to retain only interesting images, where GFP expression can be observed.

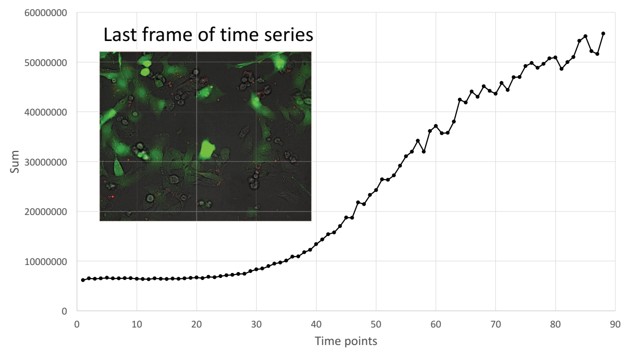

Fig. 5. Measurement of the amount of GFP at different time points.

Figure 5 illustrates GFP expression in the successful case. Summation of all pixel values in the green color channel of each image provides a basis for filtering. HarmonicIO is well suited to this case, since the images can be evaluated independently, and the object (image) size is relatively large. We developed the image analysis component (and its dependencies) separately and distributed it via a Docker container, convenient for code re-use.

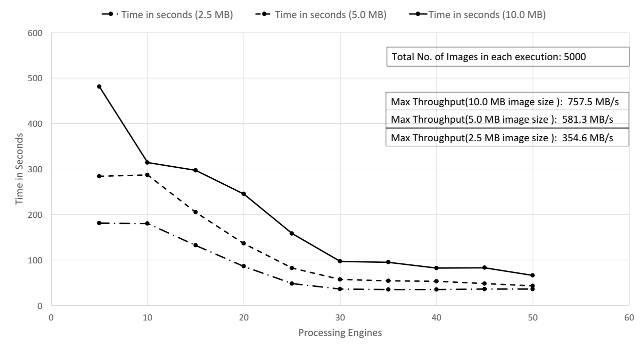

Fig. 6. Strong scaling, fixed data size (5000 images) with varying processing engines (5 - 50). The horizontal and vertical axes shows the *number of processing engines* and *total execution time (transfer + green channel analysis* respectively.

We conducted both strong- and weak-scaling analysis for this use-case. Figure 6 shows the strong-scaling results: with 30 processing engines, the framework reached its maximum performance for this specific workload and adding further resources contributed marginally to the throughput.

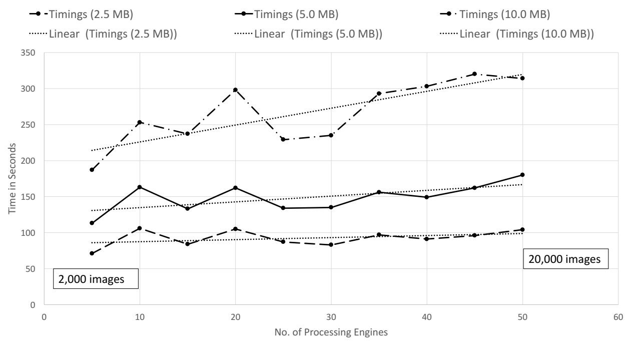

Fig. 7. Weak scaling, varying data size (2,000 - 20,000 images) and number of processing units (5-50).

Figure 7 shows the weak-scaling results: the number of images in each case were increased to maintain constant number of images per processing engine.

The framework scales well for 2.5MB and 5.0MB files, whereas for 10MB, there was an increase in overall transfer and processing time with a 10x increase in workload causing approximately a 50% increase in processing time – as the system throughput becomes constrained by the underlying maximum network throughput. In the best case scenario we processed 200GBs of data in less than 5 minutes.

V. Conclusion and Future Directions

We have proposed HarmonicIO as a smart, scalable and cost-efficient stream-based processing framework for large-scale datasets. The framework design targets scientific datasets (and applications) but the framework can also be used for general purpose streaming applications. Our results show the performance and effectiveness of the framework for a realistic image processing use case. HarmonicIO is used in HASTE[1] to provide containerized stream processing in the cloud. Future work includes intelligent placement of computations/containers, and more efficient data processing for small objects. HarmonicIO is open-source software[2].

VI. Acknowledgements

This work has been funded by the Swedish Foundation for Strategic Research (SSF), the Göran Gustafsson foundation and the Swedish strategic initiative for eScience, eSSENCE.

References

Ettinger, A. and Wittmann, T. (2014). Chapter 5 - Fluorescence Live Cell Imaging. In Waters, J. C. and Wittman, T., editors, *Quantitative Imaging in Cell Biology*, volume 123 of *Methods in Cell Biology*, pages 77 – 94. Academic Press.

Gama, J. and Rodrigues, P. P. (2007). Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer.

Kauffman, K. J., Dorkin, J. R., Yang, J. H., Heartlein, M. W., DeRosa, F., Mir, F. F., Fenton, O. S., and Anderson, D. G. (2015). Optimization of lipid nanoparticle formulations for mrna delivery in vivo with fractional factorial and definitive screening designs. *Nano Letters*, 15(11):7300–7306. PMID: 26469188.

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

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA. ACM.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239).

Spring, J. H., Polytechnique, E., Lausanne, F. D., and Privat, J. (2007). Streamflex: high-throughput stream programming in java. In *In OOPSLA*, pages 211–228. ACM.

Toor, S., Lindberg, M., Falman, I., Vallin, A., Mohill, O., Freyhult, P., Nilsson, L., Agback, M., Viklund, L., Zazzik, H., et al. (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.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.

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

[2]`https://github.com/HASTE-project/HarmonicIO`, `https://github.com/HASTE-project/HarmonicPE`