FID3008: Approaches to Dataflow Processing

Max Meldrum - mmeldrum@kth.se

1 Introduction

The dataflow processing model is at the core of modern distributed data processing frameworks. This review looks at three system papers that utilise the aforementioned model in contrasting ways. The first paper is MillWheel: Fault-Tolerant Stream Processing at Internet Scale [1] which was published at VLDB 2013. The second paper is Naiad: A Timely Dataflow System [2], also published in 2013, but at SOSP. Finally, the last paper is Ray: A Distributed Framework for Emerging AI Applications [3] published at OSDI 18.

2 MillWheel

MillWheel is described as a distributed stream processor for large-scale real-time analytics at Google. Similar to the MapReduce [4] model, the aim of MillWheel's programming model is to allow users to create large-scale systems without being distributed systems experts. MillWheel adopts a static dataflow graph where each vertice represents a streaming operator that run transformations on incoming data before passing the data along the graph. The paper introduces the system requirements by going through an example pipeline at Google called Zeitgeist. The key requirements it touches upon are the following:

- Persistent state abstractions
- Exactly-once semantics
- Out-of-order processing

MillWheel enables users to reason about state and different consistency models within the programming interface. The state of the streaming pipelines is partitioned by key and is backed by a global store such as BigTable/Spanner. It is through different communication strategies to the global store that MillWheel is able to support various consistency guarantees (e.g., exactly-once). The MillWheel system utilises the Watermark notion to measure progress and to deal with out-of-order [5] processing.

Thoughts: The paper is well written and it is great to see that the system requirements were driven by a real-world application scenario. I like the simplicity of the system design. State is managed per key, thus making it easy to reason about. Progress is tracked through watermarks as it scales well for the target environment while also introducing low complexity compared to other coordination mechanisms. The experiments in the paper focus mostly on the latency aspect of streaming records and watermarks. The authors state that the one of the core contributions of their work is an efficient implementation of the MillWheel system, however, I do find it interesting that there is no comparison to other systems in the evaluation section. It is quite common to run experiments against the state-of-the-art. I would be curious to know if this is because there was no comparable system at that time or if it was simply a lack of motivation by Google. The other main contribution is the programming model allowing users to design large-scale streaming pipelines using state, timer, and key abstractions.

The work behind MillWheel has greatly influenced streaming systems/models today (e.g., Apache Flink [6], Apache Beam [7]).

3 Naiad

Naiad, is a distributed dataflow processing system for data-parallel applications. The aim of Naiad is to provide a general-purpose system that supports high-throughput and low-latency processing, while simultaneously supporting iterative and incremental com-

putation. The authors state that there is no such existing system capable of supporting all the aforementioned qualities. To that end, they introduce a new computational model called *timely dataflow*. The key features of the *timely dataflow* model are as following:

- Structured loops within the static dataflow graph, allowing for feedback within the dataflow.
- Stateful operators capable of ingesting and producing records without global coordination.

Timely dataflow graphs are directed but may contain cycles as depicted in figure 1. Logical timestamps are utilised to track global progress. The timestamps in timely dataflow are defined as a tuple containing an epoch and loop counter. The loop counter helps the system identify which iteration the loops are currently in. This form of progress tracking enables efficient execution of iterative workloads (e.g., graph algorithms).

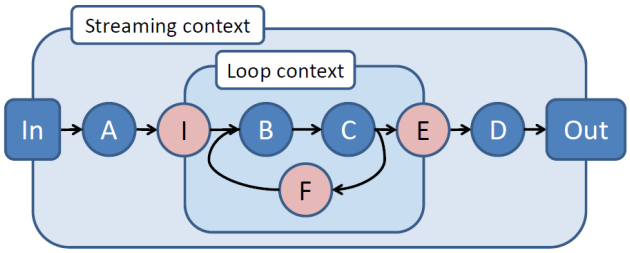

Figure 1: Structured loop in timely dataflow [2]

Naiad is the distributed implementation of the *timely dataflow* model and its system stack is illustrated in figure 2. Naiad constructs the same dataflow graph for each binary worker that is deployed in the cluster. Data-parallelism is achieved by partitioning the incoming data and coordination across the dataflow stages is done through a distributed progress tracking protocol.

Thoughts: The *timely dataflow* model is expressive and very well suited for iterative algorithms whether if it is bounded or unbounded data. The Naiad system does support fault-tolerance through a checkpointing protocol, however I would not say it is a strong point of the system. This is indicated by the lack of experiments showing how well Naiad recover from failures. Naiad is described as a general-purpose system and although it may express streaming computation, it does not support event-time windowing, an essential feature for any production-grade stream processor. The timely dataflow computational model is powerful, but I think the reason it didn't take off

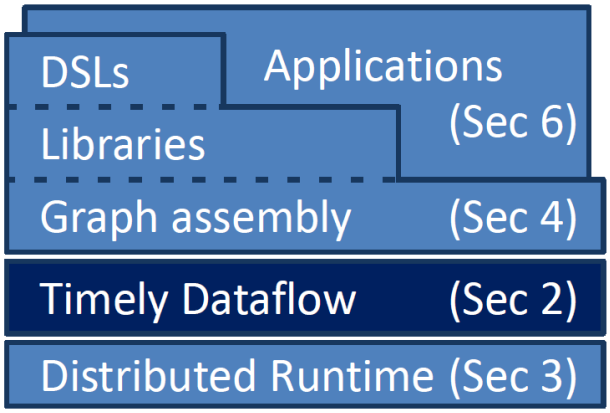

Figure 2: Naiad's System Stack [2]

in the industry is due to its high complexity. Also, its progress tracking mechanism does not scale as well compared to the Watermarks approach when you start adding hundreds of machines to the cluster.

There is still ongoing work happening today with a Rust implementation [8] of the *timely dataflow* model.

4 Ray

Ray is a distributed dataflow processing system mainly targeting AI applications (e.g., Reinforcement Learning). It does not assume a static dataflow graph as the two previous systems. Ray utilises a dynamic task graph model and combines the task-parallel and actor model into a single API. The authors started working on the Ray system after attempting to build a distributed training library in Spark [9]. They state that the inflexibility of the BSP model and lack of actor abstraction led them to develop Ray. Ray provides a flexible computational model that supports both stateful (Actor) and stateless (Futures) tasks. Stateful actors are useful for things like parameter servers, while dynamically spawned stateless tasks are well suited for RL simulations as one does not know a priori when to end a simulation.

A core focus of the system architecture was the ability to scale. Ray has to be able to execute millions of tasks per second in order to satisfy the demands of RL algorithms. Ray does not employ a centralised scheduler as in Spark, as this would end up becoming a bottleneck. They instead adopt a hybrid scheduling approach that favours scheduling

tasks on a local worker rather than going to the global scheduler. Figure 3 illustrates the system architecture of Ray. Another key component of their design is the GCS (Global Control Store). Similar to Spark, Ray utilises the lineage to re-execute computation in order to recover from failed tasks. Other than storing the lineage, the GCS is also used to coordinate the execution from node to node.

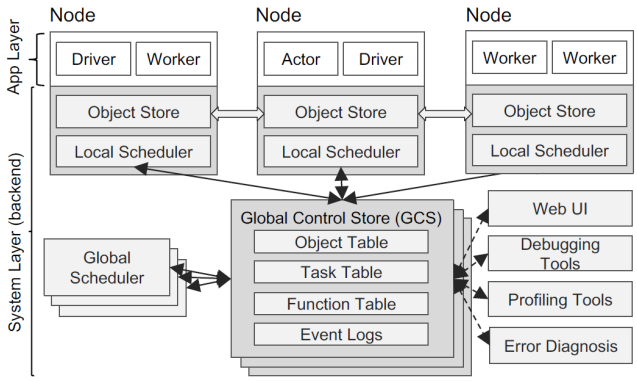

Figure 3: Ray's System Architecture [3]

Thoughts: Ray's dynamic task graph approach is inspired by the CIEL [10] system. However, in contrast to CIEL, Ray avoids the centralised scheduler and merges the actor model into the programming model. I like the idea of building the computation on top of a dynamically growing graph. Not much focus was put onto the consistency guarantees provided by underlying system. From what I understand, it could happen in worst case that a stateful actor recovers into a non-consistant state. However, this might not matter that much for the system's targeted use cases. Their evaluations focus mainly on scalability and how well their system handles different tasks of a Reinforcement Learning application (e.g., training, serving, and simulation). They run comparisons against OpenMPI using an allreduce example. As OpenMPI is not a distributed dataflow processing engine, it would have been more interesting in my opinion to show comparisons to Spark. To conclude, I do agree with the authors choice of having a centralised store at the core of the system. While their contributions are not drastically novel, Ray addresses important system demands for an increasingly hot topic in that of AI.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[2] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[5] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288, August 2008.

[6] Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink tm : Stream and batch processing in a single engine paris. 2016.

[7] Apache beam. `https://beam.apache.org/`. (Accessed on 04/05/2020).

[8] Timelydataflow/timely-dataflow: A modular implementation of timely dataflow in rust. `https://github.com/TimelyDataflow/timely-dataflow`. (Accessed on 04/06/2020).

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A

fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[10] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 113–126, USA, 2011. USENIX Association.