Summary of “Naiad : A Timely Dataflow System”

Matthew Furlong (mfurlon), Ayush Goel (goelayu), HyunJong Joseph Lee (hyunjong)

Problem and Motivation

There are currently two main systems for processing data: batch and stream. These processing methods have advantages and disadvantages because of their inherent tradeoffs. Batch systems can use iterative algorithms synchronously and have high throughput but suffer from latency issues. Stream systems provide low-latency results but require non-iterative algorithms and have lower throughput. For example, Spark provides users with a framework for batch processing, while Apache Storm is specifically designed for low-latency responses by processing a single datum as it arrives. As an interesting note, Spark Streaming attempts to perform streaming by using batch processing where the batch size (measured in time) is as low as 500ms, but this naturally incurs a higher latency in comparison to systems like Storm.

While distributed systems exist that specialize in either batch or stream, there is not currently a single framework that allows for either processing method to be used within one system. Applications that require high throughput, low latency, and iterative computation must therefore rely on multiple systems which dramatically increases the complexity for the user.

Hypothesis

The authors propose that a single framework can provide the benefits of high throughput and iterative algorithms (batch processing) and low-latency results (stream processing). In other words, they propose a general-purpose system that can outperform specialized systems in all three characteristics mentioned above.

Solution Overview

Naiad use a timely dataflow to represent the computation throughout a distributed system. A timely dataflow is a directed graph but it can contain cycles known as loop contexts. A system entering a loop context must pass through a node to enter (ingress vertex) and leave through a different node (egress vertex). Within the loop context, there is a single node (feedback vertex) that counts the number of cycles completed within in the loop. This feedback vertex allows for messages to be logically timestamped using the counters and the counters are able to distinguish between consecutive iterations in a cycle.

Vertexes within a timely dataflow communicate via four methods. For the following functions, *e* is an Edge, *m* is a Message, and *t* is a timestamp. A vertex, u, can invoke two functions, u.SendBy(*e*, *m*, *t*) and u.NotifyAt(*t*). The vertex receiving the

output of these functions, as determined by the edge u to v specified in the SendBy method, implements two callback functions, v.OnRecv(*e, m, t*) and v.NotifyOn(*t*). v.OnNotify() indicates that all v.OnRecv() invocations have been delivered to v, and thus the system must guarantee that v.OnNotify(*t*) is invoked only after no further calls to v.OnRecv(*t'*) for *t'* <= *t* will occur (see limitations).

Naiad also include support for distributed progress tracking to ensure that there are no events at any worker throughout the distributed system that could potentially result in an identical pointstamp of the delivered notification from another worker. In this system, each worker maintains a local occurrence counter, a local precursor count, and a local frontier. When a worker sends an event, it does not update its local occurrence counter until after it has broadcasted a progress update. Progress updates are pairs of (Pointstamps, update rules), with the simple update rules defined in Section 2.3 of the paper. The key insight of this system is the local frontier can never move ahead of the global frontier, and thus if a worker has a pending notification that is in the local frontier, it must also be in the global frontier and thus can be safely delivered to the next vertex. The paper discusses two optimizations for broadcasting updates for reducing communication between nodes in the system: projected pointstamps and a local buffer to accumulate progress pairs before sending updates.

Limitations and Possible Improvements

Simulating synchronous batching on top of asynchronous streaming improves job-completion latency in exchange of fault-tolerance. Naiad takes full checkpoints, which means for every checkpoint, worker and message delivery threads need to be paused to ensure correct stateful vertex in restoring time. This heavy checkpoint mechanism can be a substantial bottleneck of asynchronous streaming, by reducing parallelism. Furthermore, due to progress tracking protocol's '*could-result-in*' relationship, deterministic recovery is not guaranteed.

Progress tracking with pointstamp relies on monotonically increasing epoch t completion to determine tasks' sequence (i.e., task with time t can happen after time t' where t' <= t). Because asynchronous streaming allows processing on-the-fly, external entity (e.g., user, application) to notify the system the termination of a tasks. That means, Naiad needs to hold stateful information of a task till an external entity explicitly declares a termination of task.

Distributed workers orchestrated by individual machine's scheduler requires a global barrier to ensure consistency. The paper denotes that as number of machines increase, micro-stragglers adversely affect global-barrier's tail latency. However, programs requesting complete notification in a stage of subset of vertices are rare, and therefore, high tail latency due to global barrier is a corner case.

Summary of Class Discussion

In the class, we first discussed how Naiad simulates batch processing on top of stream processing and the tradeoff between low latency and fault tolerance. Among many optimizations Naiad made, we exchanged our thoughts about tracking stages and connectors as logical graph instead of actual graph vertices and edges in order to reduce metadata communication between workers. One of the conclusions we reached was that fault-tolerance in streaming can be very complex and the paper does not cover much in about this availability aspect, leaving it as a future work.