Naiad

James Thomas
Goals

- High-throughput batch processing
- Low-latency processing
- Iterative computation with streaming updates (novel contribution)
- For 100% in-memory workloads
Novel Application, CIDR 2013 paper

- Maintaining connected components of graph formed by @username mentions on Twitter
- Connected components is iterative algorithm
- Batches of updates with new @username mentions coming in from Twitter, need to maintain connected components in real time
- First system that can do this
Solution: Lower-Level API, Vertex Model

- Philosophy: hack at lower level if performance needed, otherwise use higher-level library

```java
v.ONRECV(e : Edge, m : Message, t : Timestamp)
v.ONNOTIFY(t : Timestamp).

this.SENDBY(e : Edge, m : Message, t : Timestamp)
this.NOTIFYAT(t : Timestamp).
```
Low-level API Example

class DistinctCount<S,T> : Vertex<T>
{
    Dictionary<T, Dictionary<S,int>> counts;
    void OnRecv(Edge e, S msg, T time)
    {
        if (!counts.ContainsKey(time)) {
            counts[time] = new Dictionary<S,int>();
            this.NotifyAt(time);
        }

        if (!counts[time].ContainsKey(msg)) {
            counts[time][msg] = 0;
            this.SendBy(output1, msg, time);
        }

        counts[time][msg]++;
    }

    void OnNotify(T time)
    {
        foreach (var pair in counts[time])
        {
            this.SendBy(output2, pair, time);
            counts.Remove(time);
        }
    }
}
High-level Library Example

```csharp
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
// Here, we use LINQ to implement MapReduce.
var result = input.SelectMany(y => map(y))
    .GroupBy(y => key(y),
             (k, vs) => reduce(k, vs));

// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// 2. Supply input data to the query.
input.onNext(/* 1st epoch data */);
input.onNext(/* 2nd epoch data */);
input.onNext(/* 3rd epoch data */);
input.onNextCompleted();
```
Distributed Implementation

Logical graph: A → H(m) → B → C

Worker: Scheduler

Progress tracking protocol

TCP/IP network

Process
Distributed Progress Tracking -- Timestamps

\[
\text{Timestamp} : (e \in \mathbb{N}, \langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k)
\]
Distributed Progress Tracking -- Pointstamps

Pointstamp: \((t \in \text{Timestamp}, l \in \text{Edge} \cup \text{Vertex})\)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_SEND_BY(e, m, t)</td>
<td>( OC[(t, e)] \leftarrow OC[(t, e)] + 1 )</td>
</tr>
<tr>
<td>v_ON_RECV(e, m, t)</td>
<td>( OC[(t, e)] \leftarrow OC[(t, e)] - 1 )</td>
</tr>
<tr>
<td>v_NOTIFY_AT(t)</td>
<td>( OC[(t, v)] \leftarrow OC[(t, v)] + 1 )</td>
</tr>
<tr>
<td>v_ON_NOTIFY(t)</td>
<td>( OC[(t, v)] \leftarrow OC[(t, v)] - 1 )</td>
</tr>
</tbody>
</table>
Distributed Progress Tracking -- Putting it Together

- Can deliver OnNotify at a vertex if OC for all lower or equal timestamps at predecessor vertices or edges is 0
  - This OnNotify is in the “frontier”
- In distributed setting node’s local frontier is conservative and assumes that other nodes haven’t made progress until it explicitly hears from them
Fault Tolerance

- System calls user-defined Checkpoint() on vertices during a system-wide checkpoint, can Restore() them on failure
- Vertices can continuously log for better fault recovery at the expense of some throughput
- Higher burden on developer
Fault Tolerance -- Comparison with Spark/MR

- Since Spark/MR work with stateless tasks, on the failure of a node only the failed tasks need to be re-executed, reading from persisted barrier output.
- Since vertices are continuously sending data to one another and updating mutable state and there is no system-imposed barrier like in Spark/MR, on the failure of ANY node Naiad must stop all nodes and restore them from the last system-wide checkpoint.
- But scheduler needs to be on the path of every job to achieve this property (store lineage of ops), making Spark/MR less suitable for low-latency work.
Optimizations -- Prevent Micro-Stragglers

- Tune TCP for this workload (e.g. reduce retransmission timeouts)
- Tune GC so there are fewer stop-the-worlds
- Shared memory contention
- Keep message queues small
- Can’t solve stragglers if they still happen!