Naiad: A Timely Dataflow System,
by D. Murray, F. McSherry, et al.

Paweł Walczak

December 2, 2015
**Naiad** – distributed system for executing data parallel, cyclic dataflow programs.

- High throughput of batch processors
- Low latency of stream processors
- Support for iterative, incremental computations
Outline

- Timely Dataflow computational model
- Single-threaded solution
- Extending solution to distributed system
- Performance evaluation
- Applications
Timely Dataflow

- Directed graph
- Stateful vertices performing computations
- Timestamped messages passed along edges
- We allow cycles (possibly nested)
- Specified input and output vertices
- Each record at input is labeled with *epoch* number to distinguish between different batches of data
- Each input is informed when given *epoch* is finished
Graph structure

- Vertices organized in *loop contexts*
- Each cycle must be contained in loop context
- Edges entering context pass *ingress vertex*, and leaving, *egress vertex*
- Each cycle must pass through a *feedback vertex*
Timestamps

- Timestamp : $(e \in \mathbb{N}, \langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k)$ – epoch number, and loop counters
- Passing ingress vertex: $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_k, 0 \rangle)$
- Passing egress vertex: $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_{k-1} \rangle)$
- Passing feedback vertex:
  $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_k + 1 \rangle)$
- Timestamp ordering:
  $(e_1, \vec{x}_1) \leq (e_2, \vec{x}_2) \iff e_1 \leq e_2 \land \vec{x}_1 \leq \vec{x}_2$
Vertex implementation

Callbacks:

- $v$.OnRecv($e$: Edge, $m$: Message, $t$: Timestamp)
- $v$.OnNotify($t$: Timestamp)

System provided methods:

- $this$.SendBy($e$: Edge, $m$: Message, $t$: Timestamp)
- $this$.NotifyAt($t$: Timestamp)

It is forbidden to call methods with timestamp $t' < t$, to prevent ‘going backwards in time’.
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);
        }
    }
}
Predicting the future

- **event** – unprocessed message or notification waiting on a queue

- Pointstamp: \((t \in Timestamp, l \in Edges \cup Vertices)\)

- relation on pointstamps: \((t_1, l_1) \text{ could-result-in } (t_2, l_2)\)

- For each pair of locations \(l_1, l_2\) we can determine the path with minimal increment to timestamps

- We can precompute those values for all pairs of locations

- We call it \(\Psi[l_1, l_2]\)

- \((t_1, l_1) \text{ could-result-in } (t_2, l_2) \iff \Psi[l_1, l_2](t_1) \leq t_2.\)
Scheduler maintains set of active pointstamps. For each of them:

- *occurrence count* – how many times it occurs
- *precursor count* – how many active pointstamp *could-result-in* this one

Scheduler modifies them as needed.

*frontier* – set of pointstamps with zero precursor count

We can deliver notifications to vertices in the frontier.
Distributed version
Partitioning function

- Maps messages to integers
- System automatically routes messages so that same labels go to same workers
- Can be used for example to implement *group by* or *reduce*
- If no partitioning function, the message is processed immediately by sending worker
We extend previous solution with finding frontier.

Before delivering notification, we must ensure no other worker has data that may-result-in given pointstamp.

Each worker maintains local occurrence count, local precursor count, local frontier.

This information is delayed.
When occurrence count changes, we broadcast to all workers
\( \text{progress update} = (p \in \text{Pointstamps}, \delta \in \mathbb{Z}) \)

All workers who receive this, update the local occurrence count

No local frontier will move ahead of factual frontier
Predicting the future – distributed Optimization

- Use projected pointstamps, that is identify all physical vertices derived from a single logical vertex
- Accumulate updates before sending
- Accumulation done on many levels of system
- Accumulate as long as possible – until no pointstamp could-result-in buffered pointstamp or net update is zero
- Technical optimization: optimistically broadcast update using UDP before actual TCP communication
Latency delays

1. Computation is sensitive to latency at many points
2. Probability of latency problem increases with system size
3. Naiad copes with those problems:
   - Turning off Nagle’s algorithm on TCP connection reduces small packet exchange from 200ms to 10ms
   - Naiad uses 20ms lost packet retransmission threshold instead of 300ms
   - Naiad reduces time granularity on communication queues to 1ms
   - By managing memory in a smart way they reduce garbage collection to minimum
Writing Naiad programs

- Many higher level interfaces to use, examples: SQL, MapReduce, LINQ
- It is possible to directly construct timely dataflow graph

```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 */);
inpuIt.onNext(/* 2nd epoch data */);
inpuIt.onNext(/* 3rd epoch data */);
inpuIt.onNextCompleted();
```
Performance results overview

(a) All-to-all exchange throughput (§5.1)
(b) Global barrier latency (§5.2)
(c) Progress tracking optimizations (§5.3)

(d) Strong scaling (§5.4)
(e) Weak scaling (§5.4)
Performance results overview

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>PDW</th>
<th>DryadLINQ</th>
<th>SHS</th>
<th>Naiad</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>156,982</td>
<td>68,791</td>
<td>836,455</td>
<td>4,656</td>
</tr>
<tr>
<td>SCC</td>
<td>7,306</td>
<td>6,294</td>
<td>15,903</td>
<td>729</td>
</tr>
<tr>
<td>WCC</td>
<td>214,479</td>
<td>160,168</td>
<td>26,210</td>
<td>268</td>
</tr>
<tr>
<td>ASP</td>
<td>671,142</td>
<td>749,016</td>
<td>2,381,278</td>
<td>1,131</td>
</tr>
</tbody>
</table>

(a) PageRank on Twitter follower graph (§6.1)  
(b) Logistic regression speedup (§6.2)  
(c) k-Exposure response time (§6.3)
Solving motivating example

- Stream of continuously incoming tweets
- Extract hashtags and mentions of other users
- Find connected components in the graph of users mentioning other users
- For each connected component, find the most popular hashtag
- Quickly respond to queries for most popular tag in given user’s component
- This problem is solved in 27 lines of code
Solving motivating example