Naiad: A Timely Dataflow System

Derek G. Murray       Frank McSherry
Rebecca Isaacs        Michael Isard
Paul Barham           Martín Abadi
MSR Silicon Valley

Presented by Jesse Mu (jlm95)
Background: dataflow programming
Batch processing
Batch processing
Batch processing

Count most popular hashtags at a given time
Batch processing

Count most popular hashtags at a given time...
Batch processing
Batch processing

Must wait for all inputs to be completed (= latency)
Stream processing (asynchronous)
Stream processing (asynchronous)

Pick out key words/mentions/relevant topics
Stream processing (asynchronous)

Pick out key words/mentions/relevant topics

Real-time access
Background: types of data processing systems

- **Batch processing** (e.g. Pregel, CIEL)
  - High throughput, aggregate summaries of data
  - Waiting for batches introduces latency

- **Stream processing** (e.g. Storm, MillWheel)
  - Low-latency, near-realtime access to results
  - No synchronization/aggregate computation

- **Iterative (graph-centric) computation**
  - e.g. network data, ML
Background: types of data processing systems

- **Batch processing** (e.g. Pregel, CIEL)
  - High throughput, aggregate summaries of data
  - Waiting for batches introduces latency

- **Stream processing** (e.g. Storm, MillWheel)
  - Low-latency, near-realtime access to results
  - No synchronization/aggregate computation

- **Iterative (graph-centric) computation**
  - e.g. network data, ML
Background: types of data processing systems

Timely Dataflow

User queries are received
Low-latency query responses are delivered
Queries are joined with processed data
Complex processing incrementally re-executes to reflect changed data

One-size-fits-all
Contributions

1. **Timely dataflow**, a dataflow computing model which supports batch, stream, and graph-centric iterative processing
   a. Supports common high-level programming interfaces (e.g. LINQ)

2. **Naiad**, a high-performance distributed implementation of the model
   a. Faster than SOTA batch/streaming frameworks
Timely Dataflow supports Batch and Stream

Async event-based model
Nodes are always active.
Send and receive messages via

A. **SendBy** (edge, message, time)
B. **OnRecv** (edge, message, time)

Request and operate on **notifications** for batches

C. **NotifyAt** (time)
C. **OnNotify** (time)
Timely Dataflow supports Batch and Stream

Async event-based model

Nodes are always active.
Send and receive messages via

A. \texttt{SendBy}(edge, message, time)
B. \texttt{OnRecv}(edge, message, time)

Request and operate on \texttt{notifications} for batches

C. \texttt{NotifyAt}(time)
C. \texttt{OnNotify}(time)

Stream processing
Batch processing
A

\[ \text{a\_out} \rightarrow \text{rt\_out} \rightarrow \text{realtime output} \]

\[ \text{a\_out} \rightarrow \text{b\_out} \rightarrow \text{batched output} \]

B
<table>
<thead>
<tr>
<th>Input</th>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image.png)

- **A** → **B**
  - **a_out** to **B**
  - **rt_out** to realtime output
  - **b_out** to batched output
Pass through even numbers only

Input

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

A

A

B

A

B

\(a_{out}\)

\(b_{out}\)

\(rt_{out}\)

realtime output

batched output
Pass through even numbers only

Input
time numbers
1 9, 3, 2, 5, ...
2 3, 2, 7, 12, ...
...

Pass through all numbers; compute min of each time

realtime output

batched output

A

B

a_out

rt_out

b_out
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time);}

Pass through even numbers only

Input

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Pass through all numbers; compute min of each time

realtime output

batched output
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)}

Pass through even numbers only

Input

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

state = {}  // times -> running mins

Pass through all numbers; compute min of each time
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)}

Pass through even numbers only

Input

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

state = {}  // times -> running mins

function OnRecv(input_edge, msg, time) {
    Pass through all numbers; compute min of each time
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)
}

Input
time numbers
1 9, 3, 2, 5, ...
2 3, 2, 7, 12, ...
...

state = {}  // times -> running mins
function OnRecv(input_edge, msg, time) {
    this.SendBy(rt_out, msg, time)

Pass through even numbers only
Pass through all numbers; compute min of each time
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)
}

state = {} // times -> running mins
function OnRecv(input_edge, msg, time) {
    this.SendBy(rt_out, msg, time) // Streaming
    if (time not in state) // New time
        state[time] = msg
        this.NotifyAt(time)
function `OnRecv` (input_edge, msg, time) {
    if (msg % 2 == 0) {
        this.SendBy(a_out, msg, time);
    }
}

Pass through even numbers only

Input

time  numbers
1     9, 3, 2, 5, ...
2     3, 2, 7, 12, ...
...

state = {} // times -> running mins
function `OnRecv` (input_edge, msg, time) {
    this.SendBy(rt_out, msg, time) // Streaming

    if (time not in state) // New time
        state[time] = msg
        this.NotifyAt(time)

    if (msg < state[time]) // New min
        state[time] = msg
}

Pass through all numbers; compute min of each time

realtime output
batched output

Input time numbers
1 9, 3, 2, 5, ...
2 3, 2, 7, 12, ...
...
function `OnRecv(input_edge, msg, time)` {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)
}  

function `OnNotify(time)` {
    this.SendBy(batch_out, state[time], time)
}  

Pass through even numbers only

Pass through all numbers; compute min of each time
```javascript
function OnRecv(input_edge, msg, time) {
    if (msg % 2 == 0)
        this.SendBy(a_out, msg, time)
}

function OnNotify(time) {
    this.SendBy(batch_out, state[time], time)
}
```

**Input**

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**State**

```
state = {} // times -> running mins
```

**Node B, you’ve seen all messages for time 1**

Pass through even numbers only

Pass through all numbers; compute min of each time

**Realtime output**

**Batched output**
Input

time | numbers
---|---
1 | 9, 3, 2, 5, ...
2 | 3, 2, 7, 12, ...
...

A → B

A → B

realtime output

batched output
Input

<table>
<thead>
<tr>
<th>time</th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9, 3, 2, 5, ...</td>
</tr>
<tr>
<td>2</td>
<td>3, 2, 7, 12, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

All messages for time 1 delivered

```
Input time numbers
1 9, 3, 2, 5, ...
2 3, 2, 7, 12, ...
```

A

B

a_out

rt_out

b_out

realtime output

batched output
Input

time | numbers
-----|--------
1    | 9, 3, 2, 5, ...
2    | 3, 2, 7, 12, ...
...  | ...

All messages for time 1 delivered

A 

A_out 

B 

rt_out 

b_out 

realtime output 

batched output 

???
Progress tracking
Progress tracking

SendBy(_, _, 1)
Progress tracking

SendBy(_, _, 1)  NotifyAt(1)
Progress tracking

SendBy(_, _, (1, 1))

SendBy(_, _, 1)

NotifyAt(1)
Progress tracking

SendBy(_, _, (1, 1))

NotifyAt(1)

SendBy(_, _, 1)

SendBy(_, _, (1, 2))

NotifyAt((1, 2))
Progress tracking

SendBy(_, _, (1, 1))

SendBy(_, _, 1)  NotifyAt(1)

SendBy(_, _, (1, 2))

NotifyAt((1, 2))
Sort by *could-result-in* order

- SendBy(_, _, (1, 1))
- SendBy(_, _, 1)
- NotifyAt(1)
- SendBy(_, _, (1, 2))
- NotifyAt((1, 2))
Sort by *could-result-in* order

\[
\text{NotifyAt}(1) \quad \text{SendBy}(\_, \_, (1, 1)) \quad \text{SendBy}(\_, \_, (1, 2))
\]

\[
\text{NotifyAt}((1, 2)) \quad \text{NotifyAt}(1)
\]
Sort by *could-result-in* order

SendBy(_, _, (1, 1))

NotifyAt(1)

NotifyAt((1, 2))

SendBy(_, _, (1, 2))
Sort by *could-result-in* order

\[
\text{NotifyAt}(1) \\
\text{SendBy}(\_, \_, (1, 1)) \\
\text{SendBy}(\_, \_, (1, 2)) \\
\text{NotifyAt}((1, 2))
\]
Sort by *could-result-in* order

```
NotifyAt(1)
```

```
SendBy(_, _, (1, 2))
```

```
NotifyAt((1, 2))
```
Sort by *could-result-in* order

```
NotifyAt(1)
```

```
SendBy(_, _, (1, 2))
```

```
NotifyAt((1, 2))
```
Sort by *could-result-in* order

NotifyAt((1, 2))

NotifyAt(1)
Sort by *could-result-in* order

NotifyAt((1, 2))

NotifyAt(1)
Sort by *could-result-in* order

NotifyAt(1)

NotifyAt((1, 2))

Send notification!
Sort by *could-result-in* order

NotifyAt(1)
Sort by *could-result-in* order

Send notification!

NotifyAt(1)
Sort by could-result-in order
Sort by *could-result-in* order

...a notification can be delivered *only* when no possible predecessors of a timestamp exist
Sort by *could-result-in* order

...a notification can be delivered *only* when no possible predecessors of a timestamp exist

(based on timestamps + graph structure)
Low vs High Level Interfaces
Low vs High Level Interfaces

Event-based system

SendBy(edge, message, time)

OnRecv(edge, message, time)

NotifyAt(time)

OnNotify(time)
Low vs High Level Interfaces

Event-based system

SendBy(edge, message, time)

OnRecv(edge, message, time)

NotifyAt(time)

OnNotify(time)

Common dataflow interfaces (LINQ, Pregel)

// 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.OnCompleted();
Contributions

1. **Timely dataflow**, a dataflow computing model which supports batch, stream, and graph-centric iterative processing
   a. Supports common high-level programming interfaces (e.g. LINQ)

2. **Naiad**, a high-performance distributed implementation of the model
   a. Faster than SOTA batch/streaming frameworks
Implementation: Naiad
Implementation: Naiad
Distributed Progress Tracking
Distributed Progress Tracking

Each node has its own local progress tracker, must be conservative.

Updates other nodes over network as events finish.
Optimizations
Optimizations

Reduce small delays \textit{micro-stragglers}

Tweak TCP configuration

GC less often

Reduce backoff time to 1ms after concurrent access to shared memory
Fault Tolerance
Fault Tolerance

Since vertices have dynamic state, one failure -> all nodes have to reset from checkpoint

System-wide synchronized checkpoints

Tradeoff between how often to log checkpoints and performance
Evaluation

(a) PageRank on Twitter follower graph (§6.1)  
(b) Logistic regression speedup (§6.2)
Evaluation
Contributions

1. **Timely dataflow**, a dataflow computing model which supports batch, stream, and graph-centric iterative processing
   a. Supports common high-level programming interfaces (e.g. LINQ)

2. **Naiad**, a high-performance distributed implementation of the model
   a. Faster than SOTA batch/streaming frameworks
My opinion
My opinion

- Computational model is theoretically sound
  - Iterative computation without modifying graph in e.g. CIEL (which has overhead)
- Evaluation good too, though dramatic speedups likely better than real-world applications
- Fine-grained control over logging for fault tolerance/throughput tradeoff seems annoying
- But…
What problem does Naiad solve?
What problem does Naiad solve?

“While it might be possible to assemble the application in Figure 1 by combining multiple existing systems, applications built on a single platform are typically more efficient, succinct, and maintainable.”
What problem does Naiad solve?

“While it might be possible to assemble the application in Figure 1 by combining multiple existing systems, applications built on a single platform are typically more efficient, succinct, and maintainable.”
My opinion

- Computational model is theoretically sound
  - Iterative computation without modifying graph in e.g. CIEL (which has overhead)
- Evaluation good too, though dramatic speedups likely better than real-world applications
- Fine-grained control over logging for fault tolerance/throughput tradeoff seems annoying
- But…
  - For all but especially complex systems requiring graph + stream + batch, existing systems probably work just fine + have better infrastructure
Naiad: A Timely Dataflow System

Derek G. Murray  Frank McSherry
Rebecca Isaacs  Michael Isard
Paul Barham  Martín Abadi
MSR Silicon Valley

Presented by Jesse Mu (jlm95)