

# Naiad: A Timely Dataflow System

Derek G Murray et al.

24<sup>th</sup> ACM Symposium on Operating System Principles, 2013

Presentation by:

SB Ramalingam Santhanakrishnan

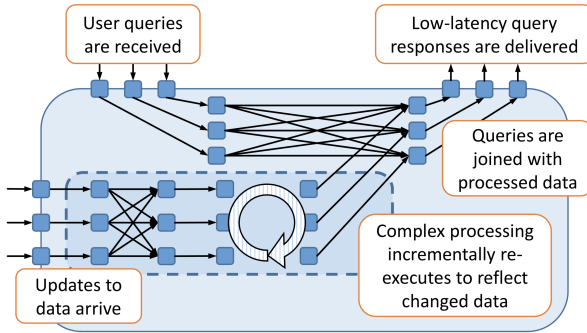
K Kleeberger

B Jain

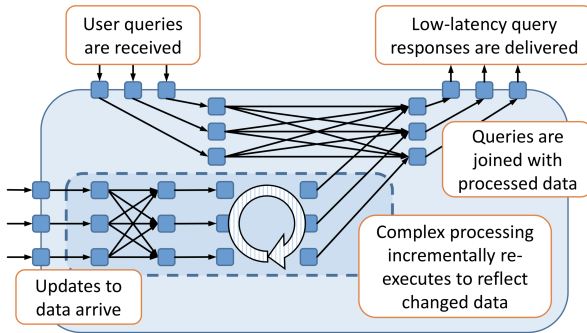
# Agenda

- Introduction
- Timely Data Flow
- Distributed Implementation
- Programming Model
- Performance Evaluation
- Real World Applications

# Introduction



# Introduction



## Batch

- Iteration
- Consistency
- Poor latency

## Stream

- No iteration
- No consistency
- Low latency

## Graph

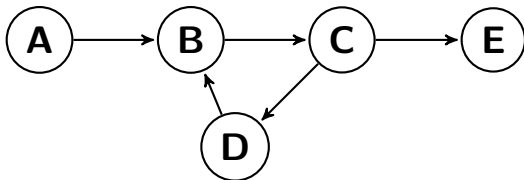
- Iteration
- Vertex-oriented Programming

# Introduction

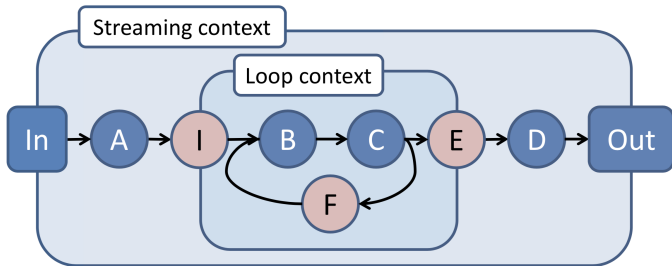
- General purpose system
- Support high-level programming models
- Iteration on real-time data
- Supports interactive queries on fresh, consistent view of results
- Low latency – specialized system performance

Dataflow programming primitives based on *time*

- Structured Loops
- Stateful vertices
- Notification for vertices on iteration completion

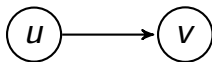


# Timely Dataflow - Graph Structure



- Time epoch on every input
- Streaming Context - Process data and pass
- Loop Context
  - Loop - Ingress (I)  $\Rightarrow$  Feedback (F)  $\Rightarrow$  Egress (E)
  - Monitors progress

## Timely Dataflow - Concurrency Primitives



Vertices register callbacks

$v.ONRECV(e: \text{Edge}, m: \text{Message}, t: \text{Timestamp})$

$v.ONNOTIFY(e: \text{Edge}, m: \text{Message}, t: \text{Timestamp})$

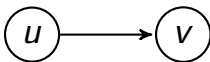
Vertices generate events

$this.SENDBY(e: \text{Edge}, m: \text{Message}, t: \text{Timestamp})$

$this.NOTIFYAT(t: \text{Timestamp})$



## Timely Dataflow - Concurrency Primitives



- `ONRECV` and `ONNOTIFY` are queued, no strict ordering
- Guarantee:  
 $v.ONNOTIFY(t)$  is invoked only after no further invocations of  $v.ONRECV(e, m, t')$ , for  $t' \leq t$
- Constraint:  
 $v.SENDBY(t')$  and  $v.NOTIFYAT(t')$  such that  $t' \geq t$

## Timely Dataflow - Timestamp

$$(e \in \mathbb{N}, \langle c_1, \dots, c_k \rangle \in \mathbb{N}^k)$$

Example: (epoch, counter) -  $(1, \langle 0, 1, 2 \rangle)$

Vertex Behavior:

- Ingress -  $\langle c_1, \dots, c_k \rangle \Rightarrow \langle c_1, \dots, c_k, 0 \rangle$
- Egress -  $\langle c_1, \dots, c_k, c_{k+1} \rangle \Rightarrow \langle c_1, \dots, c_k \rangle$
- Feedback -  $\langle c_1, \dots, c_k \rangle \Rightarrow \langle c_1, \dots, c_k, c_{k+1} \rangle$

Ordering:

$$t_1 = (e_1, \vec{c}_1), t_2 = (e_2, \vec{c}_2)$$

$$t_1 < t_2 \iff e_1 < e_2 \text{ and } \vec{c}_1 < \vec{c}_2$$

# Timely Dataflow - Progress Tracking

Future timestamps constrained by,

- Unprocessed *events* (SEND<sub>BY</sub> and NOTIFY<sub>AT</sub>)
- Graph structure

# Timely Dataflow - Progress Tracking

Future timestamps constrained by,

- Unprocessed *events* (SEND<sub>BY</sub> and NOTIFY<sub>AT</sub>)
- Graph structure

**Pointstamp:**  $(t \in \mathbf{Timestamp}, l \in \mathbf{Edge} \cup \mathbf{Vertex})$

# Timely Dataflow - Progress Tracking

Future timestamps constrained by,

- Unprocessed *events* (SEND<sub>BY</sub> and NOTIFY<sub>AT</sub>)
- Graph structure

**Pointstamp:**  $(t \in \mathbf{Timestamp}, l \in \mathbf{Edge} \cup \mathbf{Vertex})$

for,  $v.\text{SEND}_{\text{BY}}(e, m, t)$ ,  $\text{pointstamp}(m) = (t, e)$

for,  $v.\text{NOTIFY}_{\text{AT}}(t)$ ,  $\text{pointstamp}(m) = (t, v)$

# Timely Dataflow - Progress Tracking

Future timestamps constrained by,

- Unprocessed *events* (SEND<sub>BY</sub> and NOTIFY<sub>AT</sub>)
- Graph structure

**Pointstamp:**  $(t \in \mathbf{Timestamp}, l \in \mathbf{Edge} \cup \mathbf{Vertex})$

for,  $v.\text{SEND}_{\text{BY}}(e, m, t)$ ,  $\text{pointstamp}(m) = (t, e)$

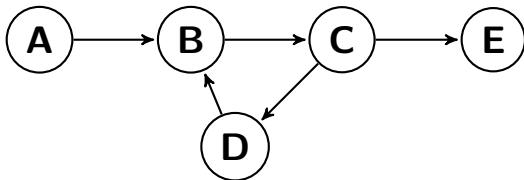
for,  $v.\text{NOTIFY}_{\text{AT}}(t)$ ,  $\text{pointstamp}(m) = (t, v)$

**Structure constraint induces ordering:**

$(t_1, l_1)$  *could-result-in*  $(t_2, l_2) \iff \exists \text{ path } \psi = \langle l_1, \dots, l_2 \rangle$

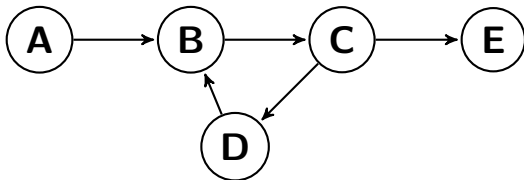
such that  $t_1$  is adjusted by each I, E or F, satisfies  $\psi(t_1) \leq t_2$ .

## Timely Dataflow - could-result-in



Is there a path between D and E?

## Timely Dataflow - could-result-in

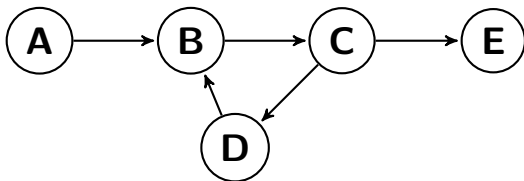


Is there a path between D and E?

$(1, A)$  *could-result-in*  $(1, E)$  ?



## Timely Dataflow - could-result-in

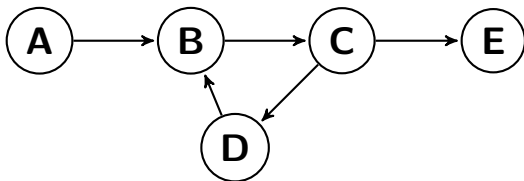


Is there a path between D and E?

$(1, A)$  *could-result-in*  $(1, E)$  ?

$((1, 2), D)$  *could-result-in*  $((1, 4), C)$  ?

## Timely Dataflow - could-result-in



Is there a path between D and E?

$(1, A)$  *could-result-in*  $(1, E)$  ?

$((1, 2), D)$  *could-result-in*  $((1, 4), C)$  ?

$((3, 4), D)$  *could-result-in*  $(2, E)$  ?

## Timely Dataflow - Single-Threaded Scheduler

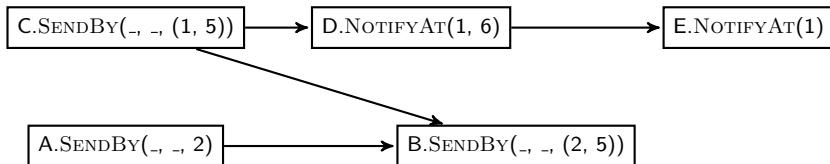
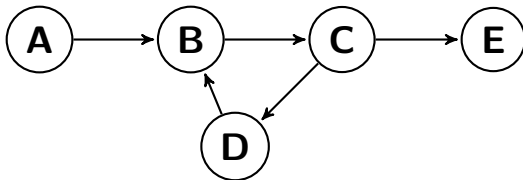
- A set of *active pointstamps* (at least 1 unprocessed event).
- For each active pointstamp, maintains,
  - *occurrence count* - outstanding events.
  - *precursor count* - how many active pointstamps precede.
- Update *occurrence count* for each event.

# Timely Dataflow - Single-Threaded Scheduler

- A set of *active pointstamps* (at least 1 unprocessed event).
- For each active pointstamp, maintains,
  - *occurrence count* - outstanding events.
  - *precursor count* - how many active pointstamps precede.
- Update *occurrence count* for each event.

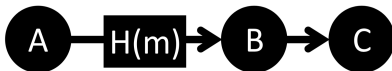
Scheduler is a simple message sorting function,  
to deliver notifications

## Visualizing the scheduler



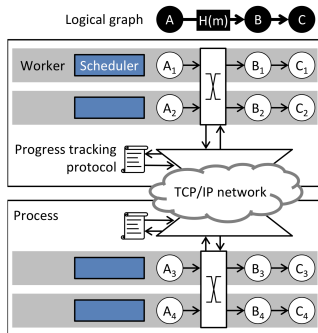
# Distributed Implementation

*Naiad* is the distributed implementation of timely dataflow



- A program consists of logical stages (A, B, C)
- $H(m)$  controls exchange of data between stages

# Data Parallelism



- Physical graph represents the chosen amount of workers and distributed connected hosts
- Programmer can select which way a message should flow in the system stages
- Naiad always uses the logical graph as a decision base where data has to flow

## Worker

Delivers messages (data) and notifications to vertices

- Tie-breaker – Always deliver messages before notifications
- Responsible for multiple vertices
- Loop data does not pass through the workers queue



## Worker

Delivers messages (data) and notifications to vertices

- Tie-breaker – Always deliver messages before notifications
- Responsible for multiple vertices
- Loop data does not pass through the workers queue

Synchronization

- Communicate through shared queue
- Queue not necessary if `SEND/RECV` are under same worker
- Re-entrancy due to loops – enqueue for later, coalesce incoming messages in `ONRECV` to reduce memory.

# Progress tracking

## **Problem:**

Notification can only be sent if there are no outstanding messages

## **Solution:** Own progress tracking protocol

- Occurrence Counters gets updated after a Broadcast to all workers
- Local counter never moves ahead of global counter
- Allows safe delivery of notifications

## Optimizing broadcast updates

- Rely on the logical graph, not the physical.
- Buffer before broadcast.
- Optimistically first broadcast via UDP to reduce latency.
- Wake up threads with either broadcast or unicast with programming primitives.

# Fault tolerance and availability

## CHECKPOINT and RESTORE interface

- Vertex
  - either log data or
  - full checkpoint when requested
- Progress Tracking Protocol
  - full checkpoint

## Tradeoff between Performance and Durability

- paper favours performance over durability
- Relies on durable input and output

## Micro Stragglers

Naiad is sensitive to latency but tiny interruptions can decrease overall performance

Iterative Computation             $\leq 1$  MS

GC, Package loss            10th of MS

Network

- Disable Nagle's algorithms
- Set smaller retry timeout for package loss
- Use different network protocols in datacenters for computing

Garbage Collection

- avoid object allocation
- use buffer pools
- use value types

# Naiad Program

- Provides public API with primitives
- Higher Level APIs
  - LINQ
  - MapReduce
  - Pregel
- Examples do not support coordination
  - to improve performance
  - concat, distinct, select
- Generic API for vertex programming
  - First define the behavior dataflow vertices
  - Second define the topology

## Prototype Program

```
// 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 => { ... });

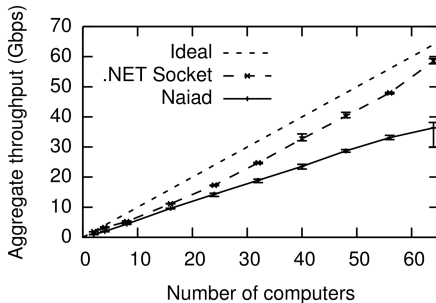
// 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```



# Performance Evaluation

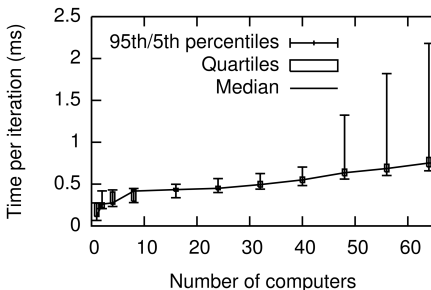
# Performance Evaluation - Throughput

- Cyclic dataflow that repeatedly performs all-to-all data exchange of a fixed number of records.
- Ideal aggregate throughput based on Ethernet Bandwidth.
- .NET Socket demonstrates achievable throughput given network topology, TCP overheads and .NET API costs.
- Naiad throughput scales linearly.



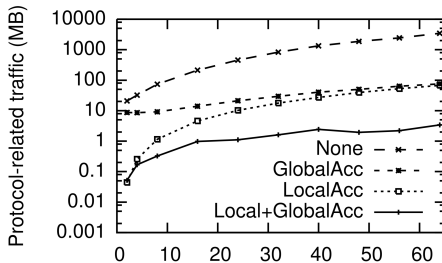
# Performance Evaluation - Latency

- Minimal time required for Global Coordination.
- Request and receive completeness notifications.
- Median time per iteration small at  $753\mu\text{s}$  for 64 computers.
- 95th percentile results - adverse impact of micro-stragglers.



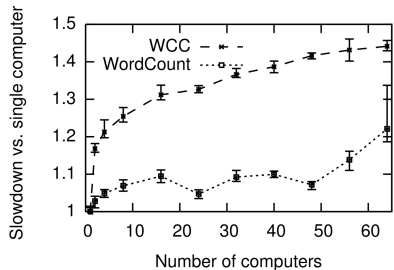
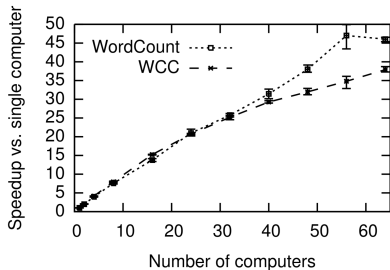
# Performance Evaluation - Protocol Optimizations

- Weakly Connected Components Computation (WCC) on random graph of 300 edges ( 2.2GB of raw input).
- Reduction in volume of protocol traffic depending on level of accumulation.
- In practice, reduction in messages from local accumulation is sufficient.



# Performance Evaluation - Scaling

- Strong scaling Adding compute resources with fix input size.
- Weak scaling Adding compute resources and increasing input.
- Word Count 128GB uncompressed corpus.
- WCC Random Graph of 200M edges.



# Real World Applications

## Real World Applications - Batch Iterative Graph Computation

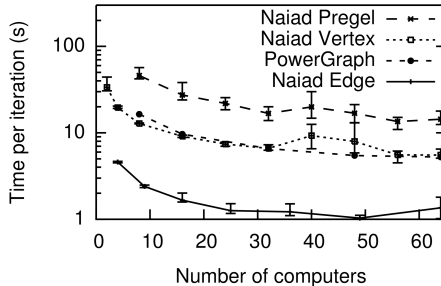
- Graph Computation on Large scale real world datasets.
- PDW – Distributed Database
- DryadLINQ – General Purpose Batch Processor
- SHS – Purpose Built Distributed Graph Store
- Speedups demonstrate the power of being able to maintain application-specific state in memory between iterations.

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

Running time in seconds

# Batch Iterative Graph Computation

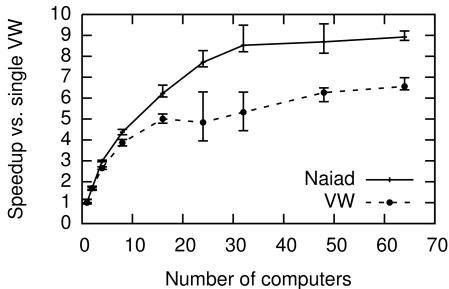
- Several systems have adopted the computation of PageRank on a Twitter follower graph as a standard benchmark.
- Naiad Vertex – Partitions edges by source vertex.
- Naiad Edge – Partitions edges using a space-filling curve.





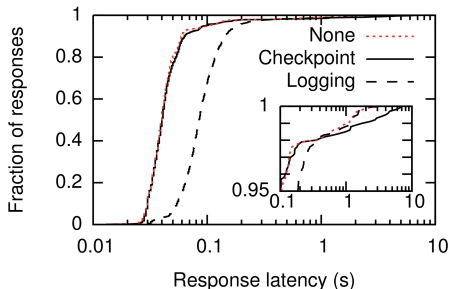
# Batch Iterative Machine Learning

- Vowpal Wabbit (VW) – open-source distributed machine learning library
- Modified VW – first and second phases run inside a Naiad vertex, third phase uses Naiad implementation of the AllReduce.



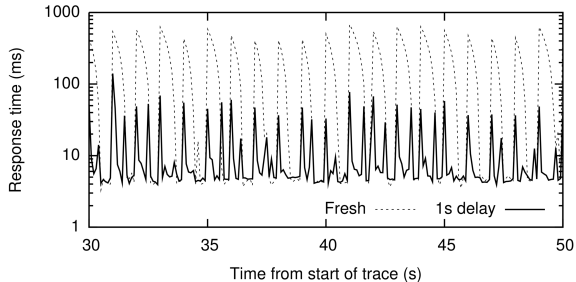
# Streaming Acyclic Computation

- Kineograph ingests arriving graph data, takes regular snapshots for data parallel computation, and produces consistent results.
- The system is partitioned into ingest nodes and compute nodes.
- k-exposure metric for identifying controversial topics on Twitter.



# Streaming Iterative Graph Analytics

- There are two input stages: stream of tweets and requests, specified by a user name and query identifier.
- "Fresh" – queries being delayed behind tweet processing
- "1s delay" – the benefit of querying stale but consistent data.
- Naiads support for overlapped computation by trading off responsiveness for staleness.



Thank you

SB Ramalingam Santhanakrishnan

K Kleeberger

B Jain