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

Paweł Walczak

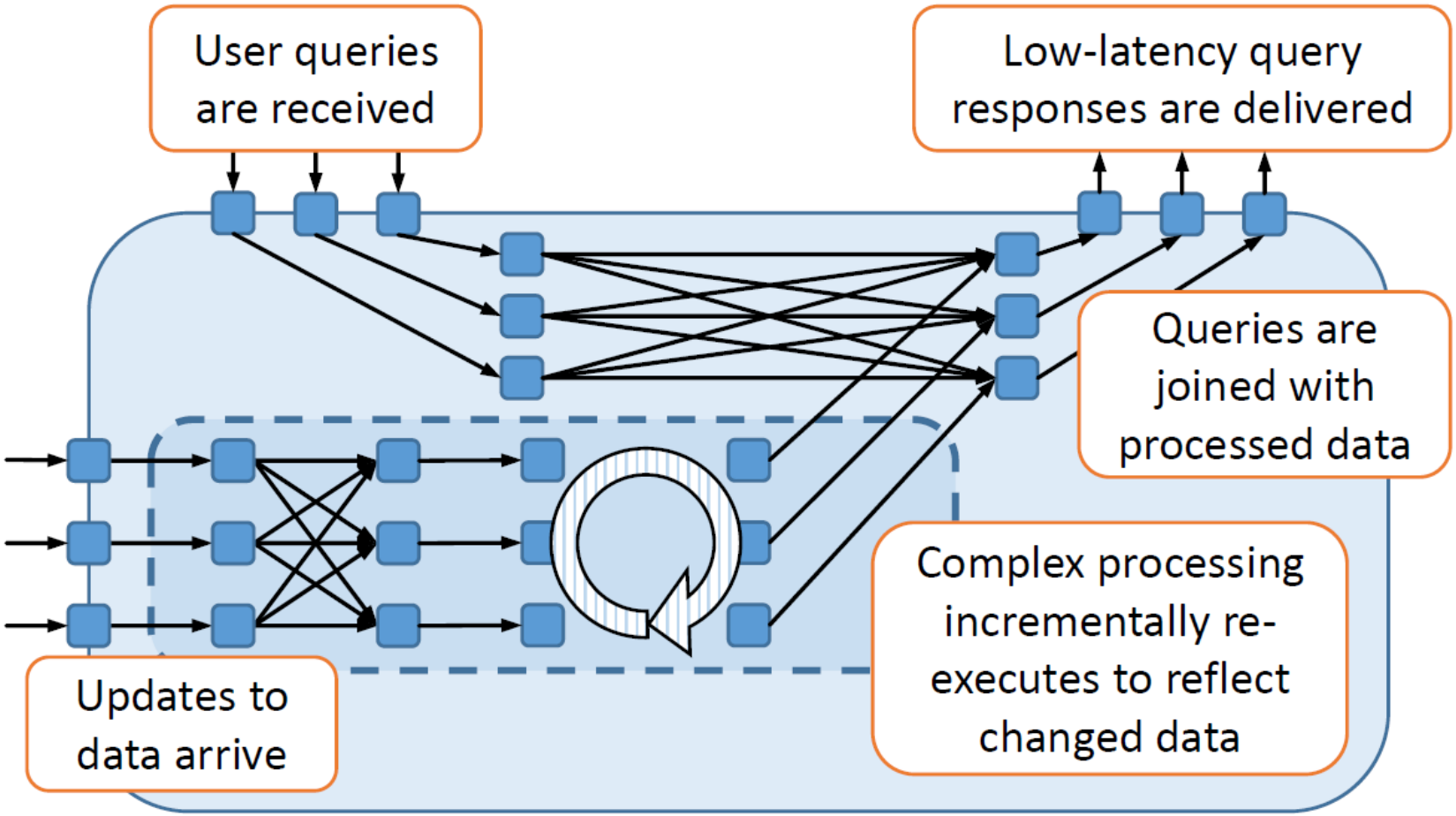
December 2, 2015

# Definition

**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

# Motivational example



# Outline

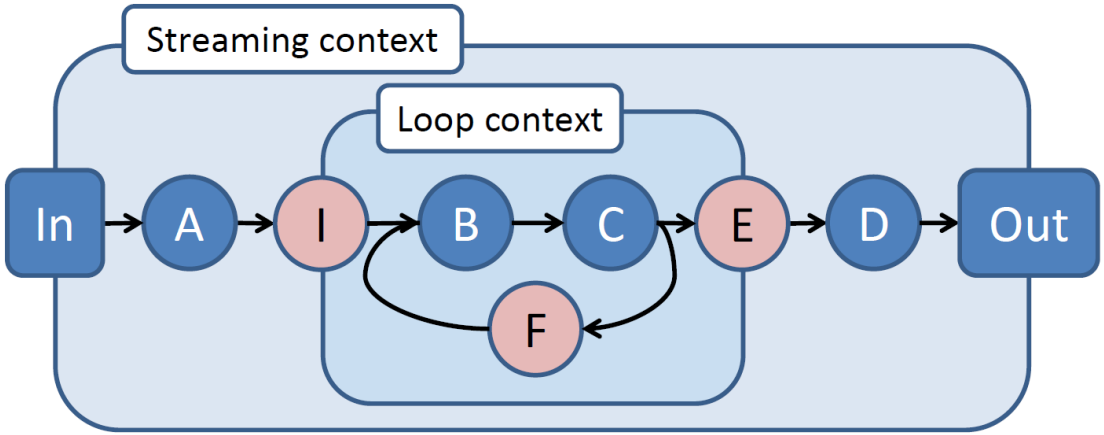
- Timely Dataflow computational model
- Single–threaded solution
- Extending solution to distributed system
- Performance evaluation
- Applications

# 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: $(e_1, \vec{x_1}) \leqslant (e_2, \vec{x_2}) \iff e_1 \leqslant e_2 \wedge \vec{x_1} \leqslant \vec{x_2}$

# 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'.

# 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);
  }
}
```

# Predicting the future

- *event* – unprocessed message or notification waiting on a queue
- Pointstamp: $(t \in Timestamp, l \in Edges \cup Vertices)$
- relation on pointstamps: $(t_1, l_1)$ *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 $\Psi[l_1, l_2]$
- $(t_1, l_1)$ *could-result-in* $(t_2, l_2) \iff \Psi[l_1, l_2](t_1) \leqslant t_2$.

# Predicting the future

Scheduler maintains set of active pointstamps. For each of them:

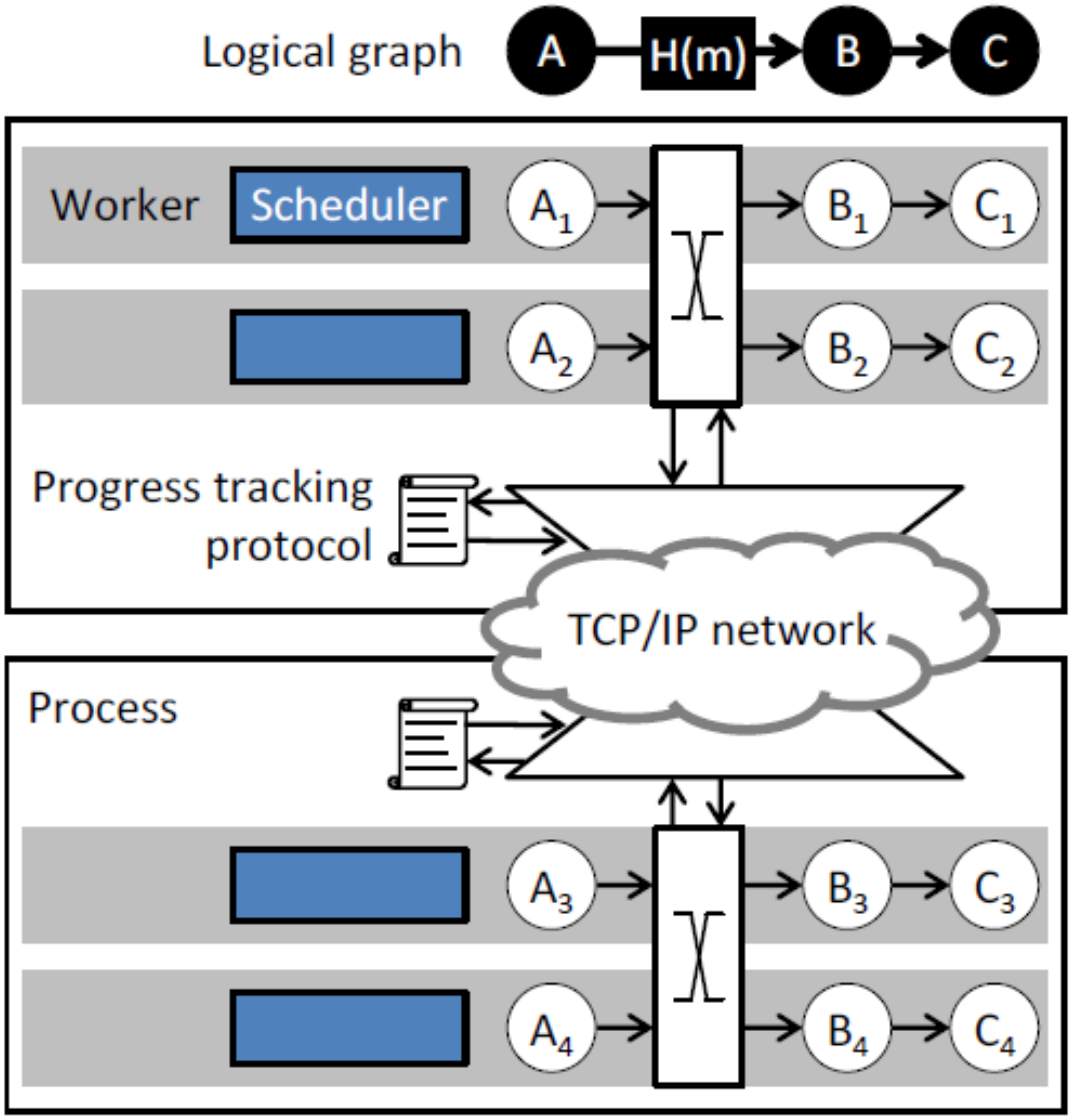
- *occurence count* – how many times it occurs
- *precursor count* – how many active pointstamp *could-result-in* this one

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

# Predicting the future – distributed

- 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 occurence count*, *local precursor count*, *local frontier*
- This information is delayed

# Predicting the future – distributed

- When occurence count changes, we broadcast to all workers $progress\ update = (p \in Pointstamps, \delta \in \mathbb{Z})$
- All workers who receive this, update the local occurence 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

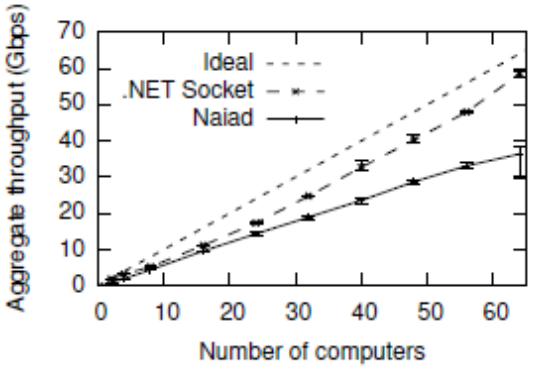
1. Computation is sensitive to latency at many points
2. Probability of latency problem increases with system size
3. Naiad copes with those problems:
   - Turning off Nagle's algorithm on TCP connection reduces small packet exchange from 200ms to 10ms
   - Naiad uses 20ms lost packet retransmission 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

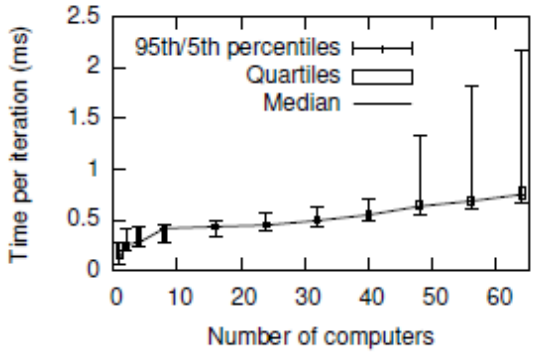
- Many higher level interfaces to use, examples: SQL, MapReduce, LINQ
- It is possible to directly construct timely dataflow graph

```
// 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();
```

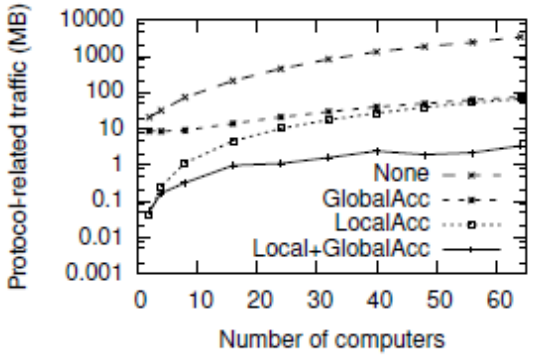
# Performance results overview



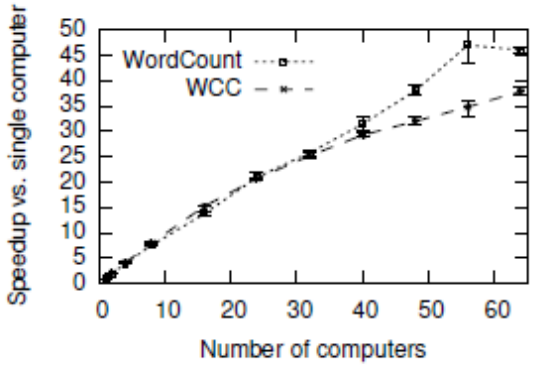

(a) All-to-all exchange throughput (§5.1)



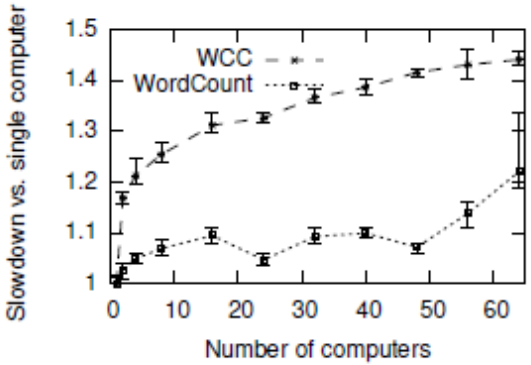

(b) Global barrier latency (§5.2)




(c) Progress tracking optimizations (§5.3)



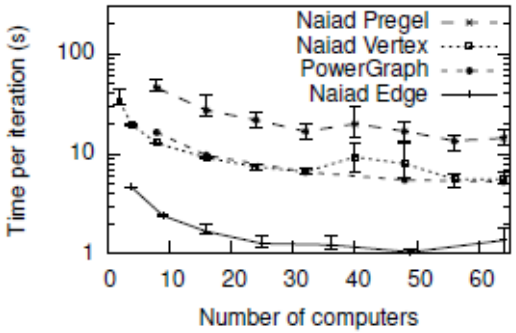

(d) Strong scaling (§5.4)



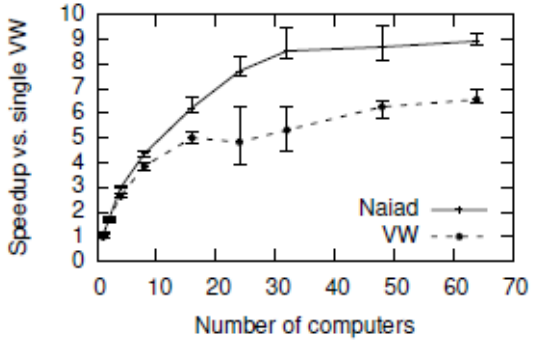

(e) Weak scaling (§5.4)

# Performance results overview

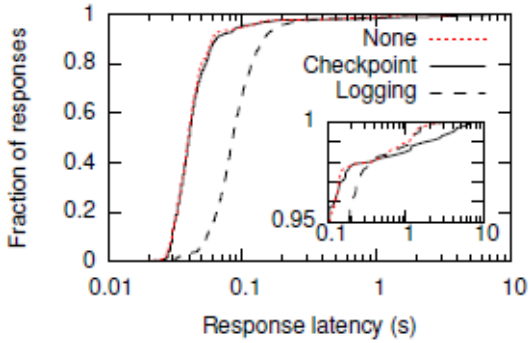
| Algorithm | PDW | DryadLINQ | SHS | Naiad |
|---|---|---|---|---|
| PageRank | 156,982 | 68,791 | 836,455 | **4,656** |
| SCC | 7,306 | 6,294 | 15,903 | **729** |
| WCC | 214,479 | 160,168 | 26,210 | **268** |
| ASP | 671,142 | 749,016 | 2,381,278 | **1,131** |




(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