Naiad: A Timely Dataflow System

Derek G. Murray, Frank McSherry, Rebecca Isaacs,
Michael Isard, Paul Barham, Martin Abadi

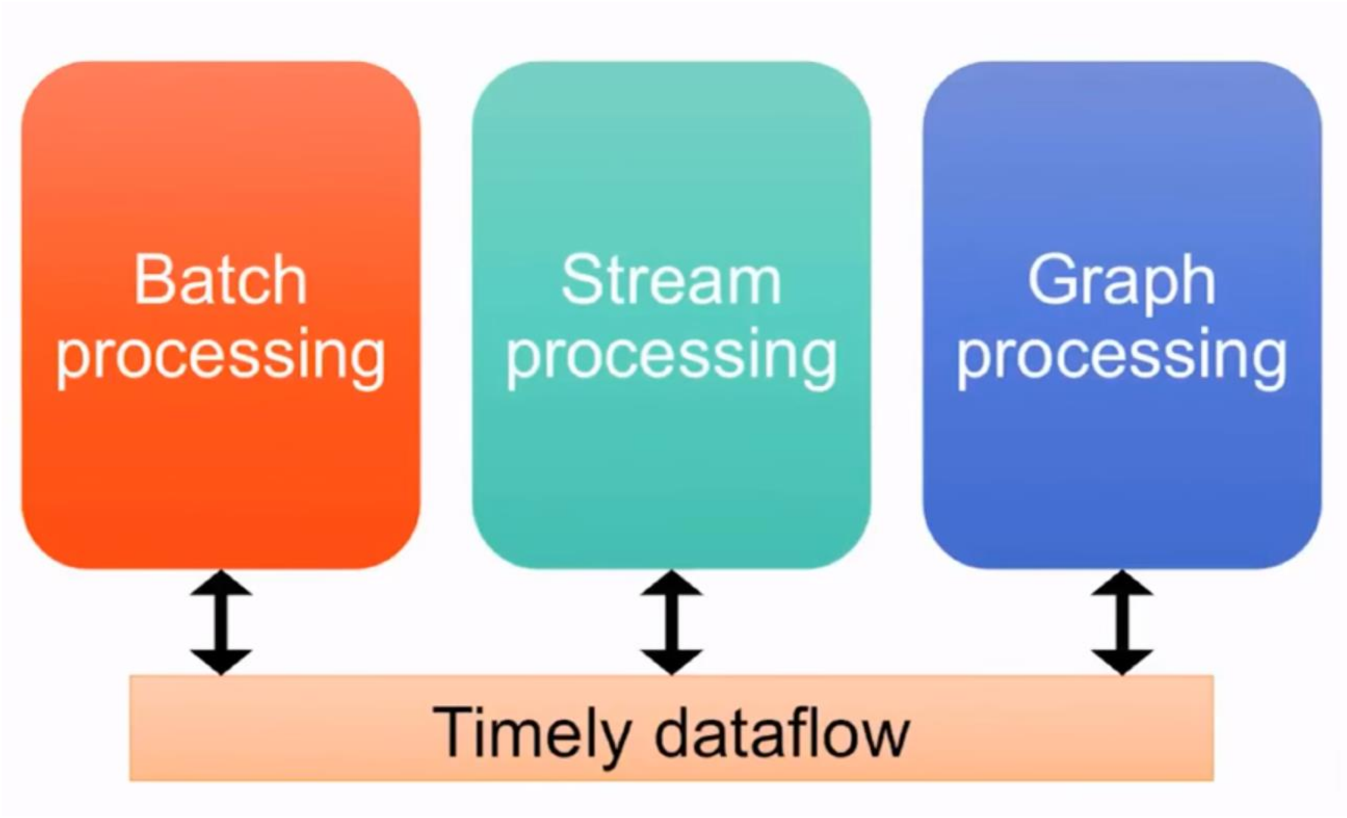
Presented by Stefan Ivanov for
R244: Large-Scale Data Processing and Optimization

Summary

- The Context – Overall ideas
- The Problem – Main contributions
- Opinions – How good is the paper?
- Conclusion

The Context

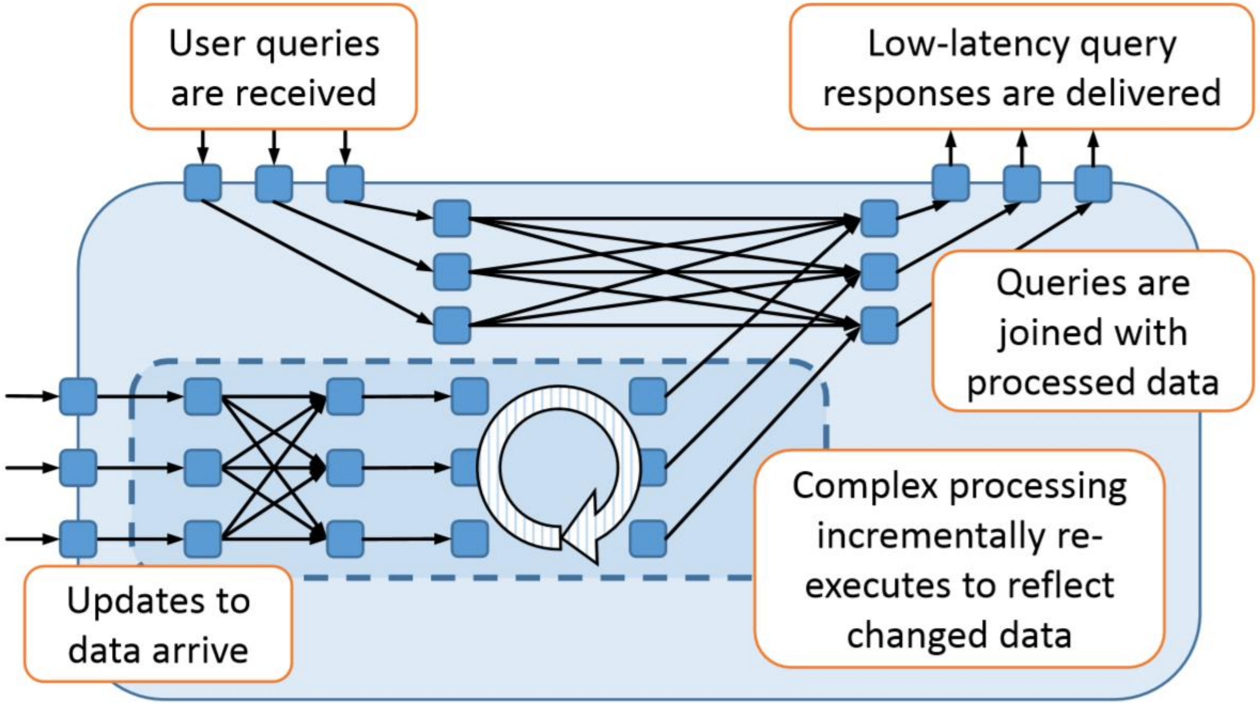
Distributed computation model



Source: [4]

Motivation for Naiad

- Data processing tasks are quite varied in terms of workload
- Architectural difficulty combining the various processing approaches



Source: [1]

What is Naiad?

A low-latency and high-throughput system for executing data parallel, cyclic dataflow programs.

A note on naming

An application written for Dryad is modeled as a directed acyclic graph (DAG) and Dryad is the "tree nymph" in Greek mythology. Naiad is a stream processing platform and Naiad is the "stream nymph" in Greek mythology.\

Authors: Who, where, when?

- Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, Martin Abadi

  → Worked for Microsoft Research Silicon Valley while writing the paper

  → Everyone (but Frank McSherry) moved to Google
- Further research on timely data flow → mostly refinements on their ideas
- Frank McSherry → also continued research on dataflow computations

Environment: Other frameworks

- Batch processing:
  - Dryad
  - MapReduce
  - Spark
- Stream processing:
  - Storm
  - MillWheel
- Graph processing:
  - Pregel
  - GraphLab
  - Giraffee

Environment: Authors' previous work

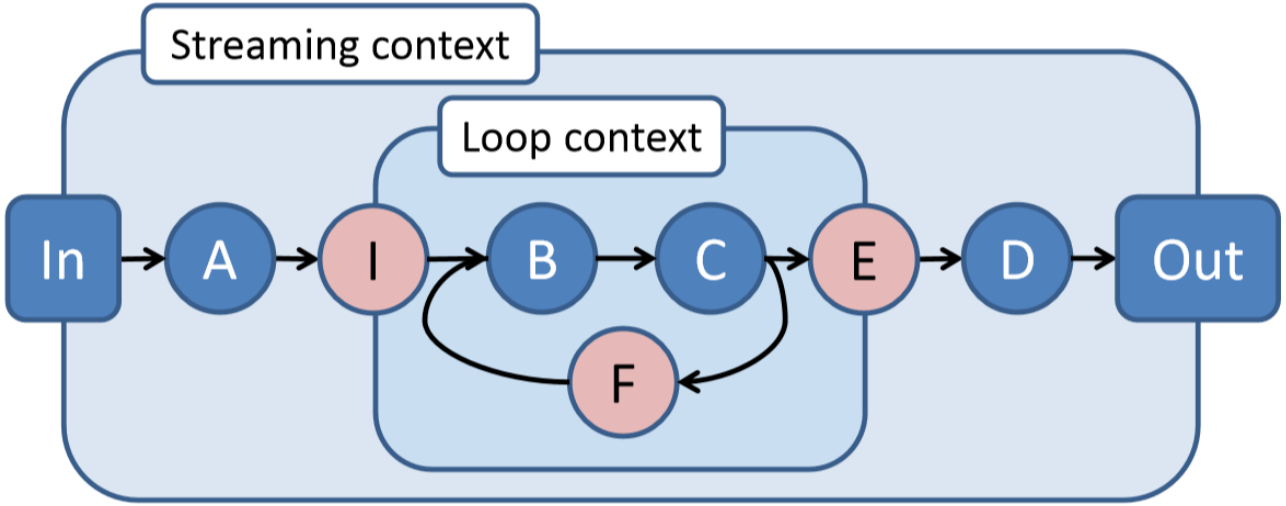
- Composable Incremental and Iterative Data-Parallel Computation with Naiad [2]
  - Verification of mathematical model and introduction to partially order relations (found in the discussed paper)
  - Precursor paper, developed from a focus on differential data flow to a more general framework

The Problem

Arbitrary Graph Execution Model

- Structured loops
- Stateful dataflow
- Notifications

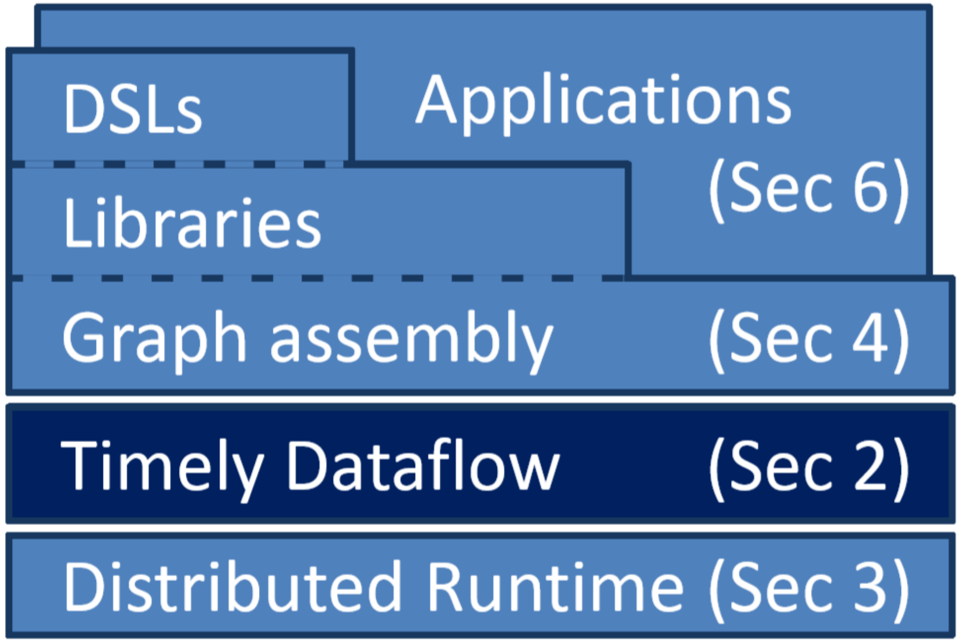
Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$



Source: [1]

Generalization for dataflow programming

- Runtime, graph construction and the timely dataflow modules are **completely** separate.
- Enables, a “mix-a-match” concentrated



Source: [1]

Timely dataflow: Timestamps

- Partial order based on lexicographical comparison
- Optimization opportunities due to formal verification of out the progress tracking code [3]

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

Source: [1]

Timely dataflow: Loop Contexts

- Necessary to impose a partial order of the notes
- Fundamental for any iterative algorithm
- Could-result-in metric

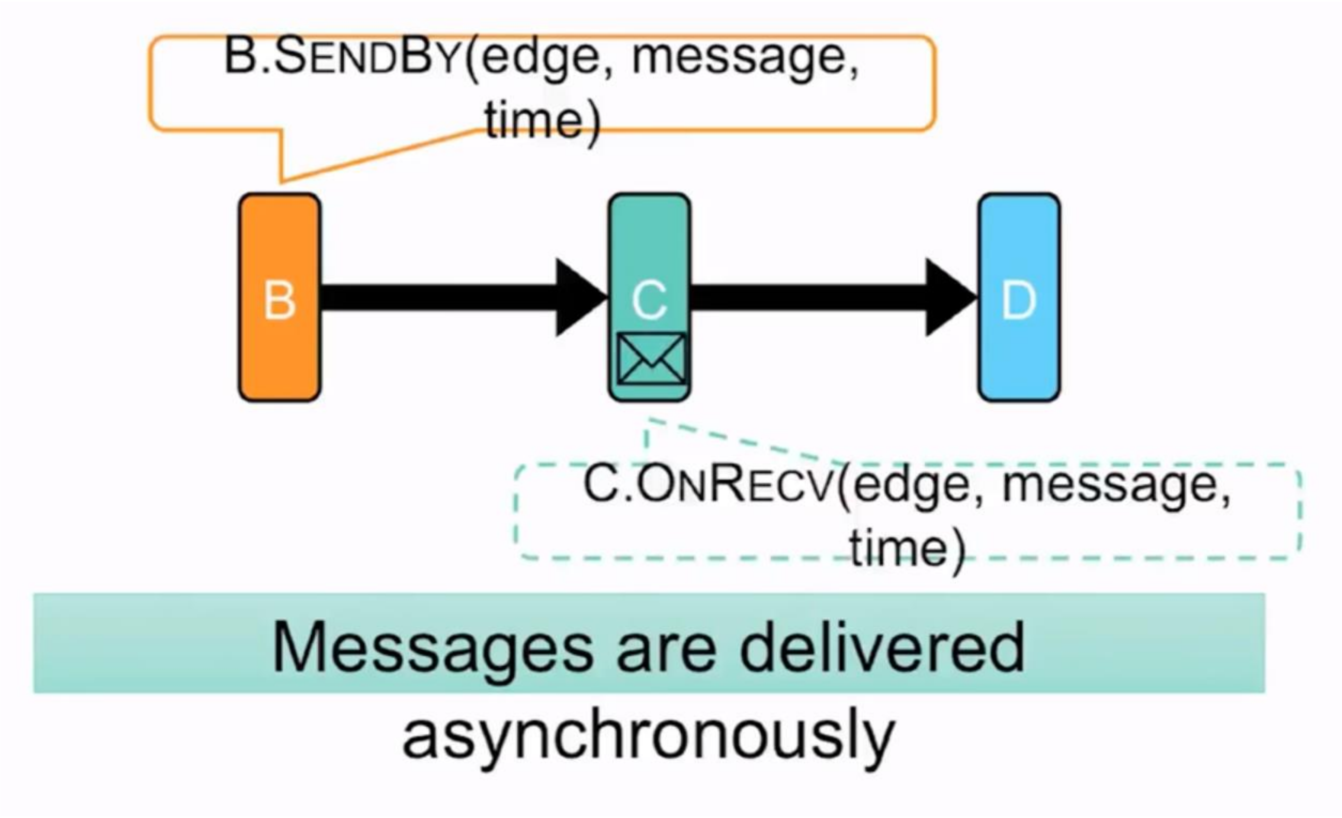
Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

Source

Timely dataflow: Callback model

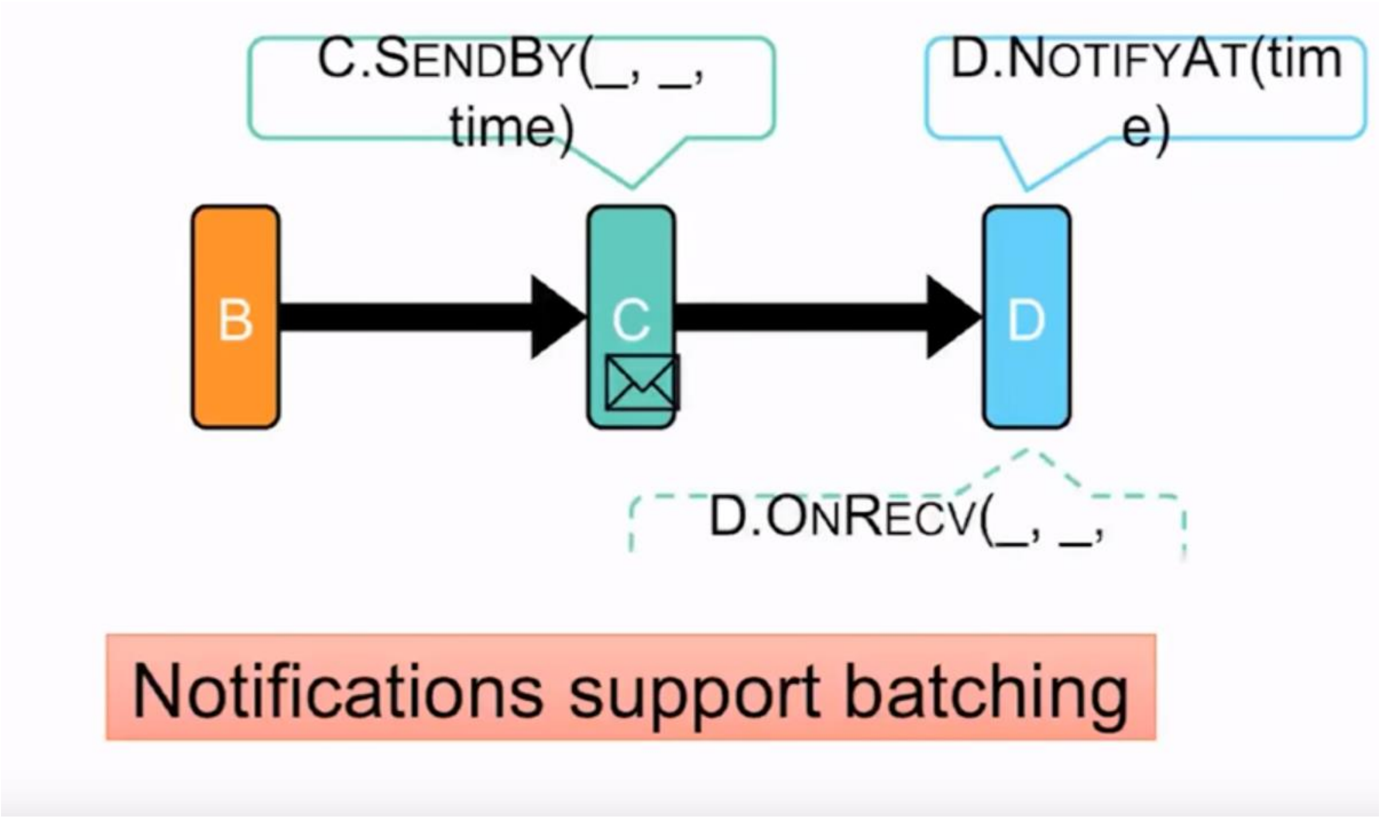
- Based on event passing (callbacks etc.)
- Interface methods
  - v.ONRECV(e : Edge, m : Message, t : Timestamp)\
  - v.ONNOTIFY(t : Timestamp)
  - this.SENDBY(e : Edge, m : Message, t : Timestamp)
  - this.NOTIFYAT(t : Timestamp).

Timely dataflow: Callback model



Source: [4]

Timely dataflow: Callback model



Source: [4]

Distributed implementation: Runtime

- Naiad "Core" → about 22700 lines of code
- Controls the "physical graph" (what runs where)
- Use of intrinsic for common operations with known semantics (i.e. join, select, count)
- Workers communicate through message queues

Distributed implementation: Low-level API

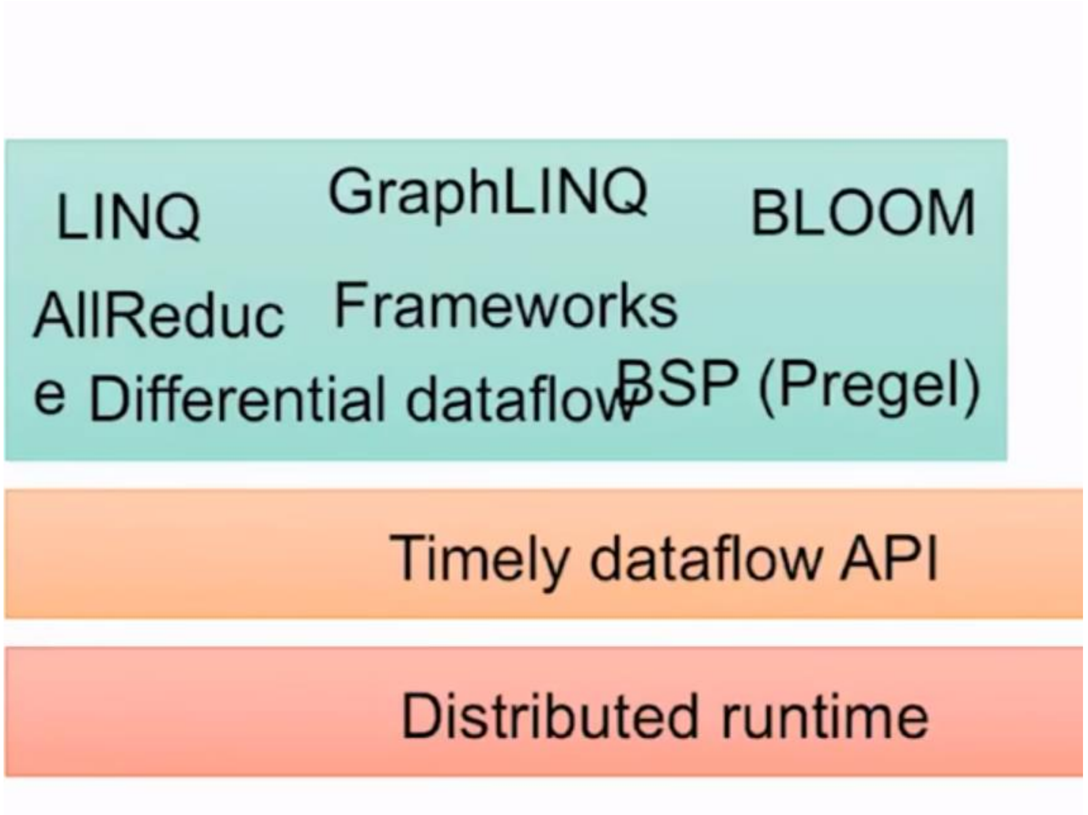
- The C# interface discussed before
- Relatively simple to use, yet verbose and error prone
- High performance applications can drop to this level if necessary

MapReduce Implementation

```
// 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();
```

Source: [1]

Distributed implementation: High-level programming models



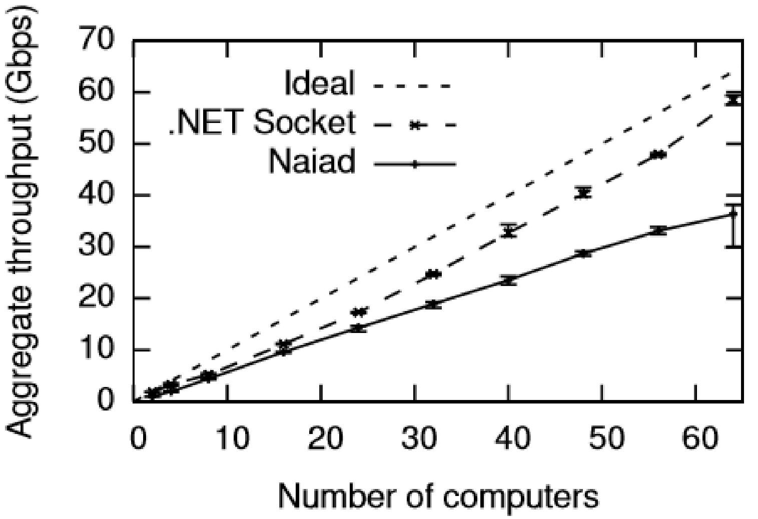
- Typical usage of Naiad is through other computational models and libraries build upon the low-level API

Mathematical formalization and optimizations

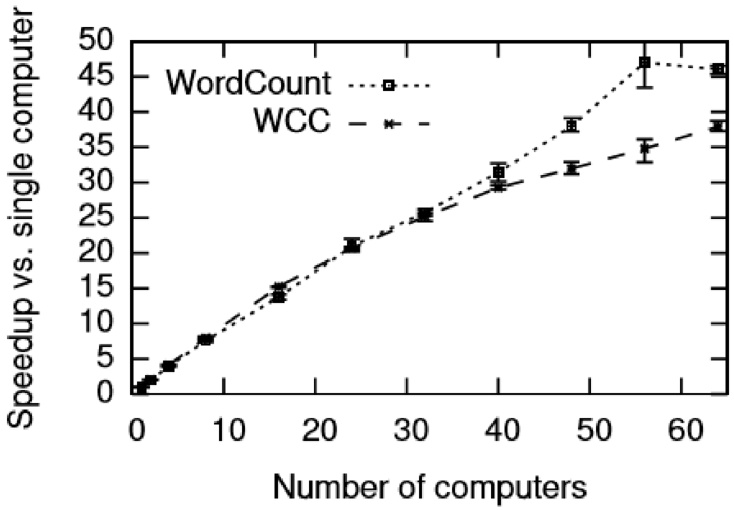
- In a separate paper [3]

"Formal analysis of a distributed algorithm for tracking progress. In Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems, June 2013"

- The previous Naiad paper [2] also contains mathematical formalism but for differential dataflow



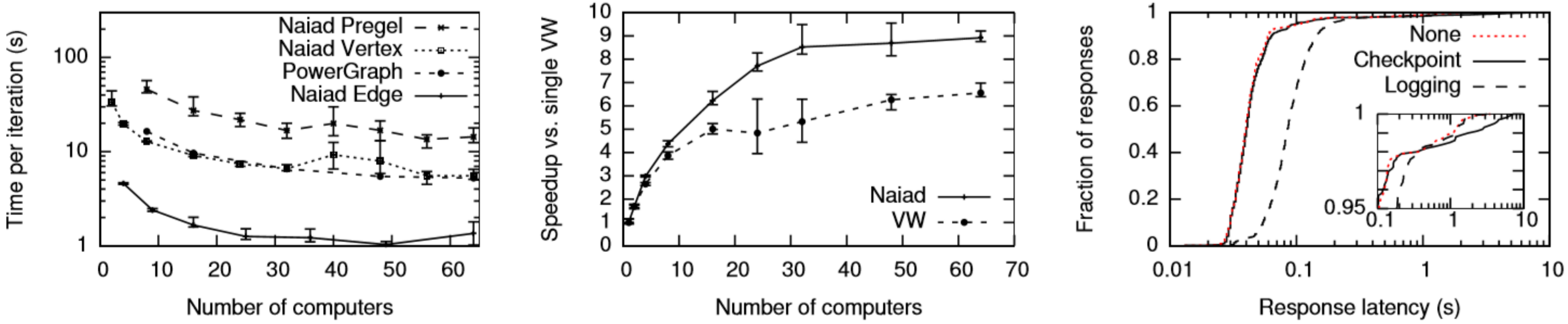
(a) All-to-all exchange throughput (§5.1)



(d) Strong scaling (§5.4)

Results: Microbenchmark results

Source: [1]



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

Results: Real world applications

Source: [1]

Fault tolerance

- Not a primary concern of Naiad
- Implemented through a Checkpoint and Restore mechanic
- Using continuous checkpoints reduces performance significantly

Opinions

Agreement and disagreements

- **Agreements**
- The API is cleaner and more extensible
- Generic API allowing for various parallel models
- Flexible execution model

- **Disagreements**
- Choice of implementation language
- Little focus on optimizations among subset of workers

Strengths and weaknesses

- **Strengths**
- Easy to implement a relatively performant distributed system in no time
- Consistency algorithms and the communication protocol is verified explicitly

- **Weaknesses**
- (Personal opinion) Not quite trivial to set up
- High memory usage which limits general applicability
- Naiad as a system is not as popular as I would expect

Key takeaways

- Timely dataflow is a unique model with convenient properties enabling high throughput and low latency
- Decoupling high-level programming model from the implementation detail of the runtime
- Providing an efficient base for complex systems enables requiring batch, stream and graph processing techniques

Impact

- Best paper of Symposium on Operating Systems Principles (SOSP) 2013
- More than 100 citations (after a quick research)
- Affected distributed data flow programming systems
- Timely dataflow programming is still in development

References

[1] Murray, McSherry, et al., Naiad: A Timely Dataflow System

[2] McSherry, Isaacs, et al., Composable Incremental and Iterative Data-Parallel Computation with Naiad

[3] Abadi, McSherry, et al., Formal Analysis of a Distributed Algorithm for Tracking Progress

[4] Naiad: A Timely Dataflow System: https://www.youtube.com/watch?v=yyhMI9r0A9E

Q&A

Thank you for your attention