Scalability! But at What COST?
and
Naiad: A Timely Dataflow System

Presented by Scott Sanderson

Papers We Love - Boston, October 7th, 2019

Scalability! But at What COST?

and

Naiad: A Timely Dataflow System

Presented by Scott Sanderson

Papers We Love - Boston, October 7th, 2019

$ whoami

- Principal Engineer at Quantopian.
- Academic background in Math and Philosophy.
- My computing interests high-performance tools more accessible for mainstream use.

Scalability! But at what COST?

Frank McSherry
Unaffiliated

Michael Isard
Unaffiliated*

Derek G. Murray
Unaffiliated†

Scalability! But at What COST?

- Published in HotOS in 2015.
- Evaluates performance of “big data” graph processing systems.
  - In particular, evaluates performance **relative to** simpler alternatives.
- Content motivated by authors’ experience working on Naiad.
  - In particular, authors’ experience evaluating and tuning Naiad’s performance.

Reasons I Love this Paper

- Short and accessible.
  - It’s just 5 pages, but it clearly articulates a problem and proposes a solution.
- A bit snarky.
  - Parts of the paper are quite critical of work it discusses.
  - Criticism generally manages to be constructive though.
- Written from a systems implementer’s perspective.

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

Abstract

- New metric, COST, for evaluating performance of “big data” systems.
  - A system’s COST for a workload is, essentially: “How many cores does your system need before it’s faster than my laptop?”
- Purpose of COST is to measure **overheads** introduced by the system
  - Communication costs.
  - Fault-tolerance.
  - Restrictive programming models.
- Many recently published systems had worryingly high COSTs.
- Authors argue that other researchers have over-prioritized scalability, to the detriment of absolute performance.

1 Introduction

"You can have a second computer once you've shown you know how to use the first one."

–Paul Barham

The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system's impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?

cppcon
ANDREI ALEXANDRESCU
std::allocator Is to Allocation what std::vector Is to Vexation
www.CppCon.org

`operator new`: The Next Generation

Introduction

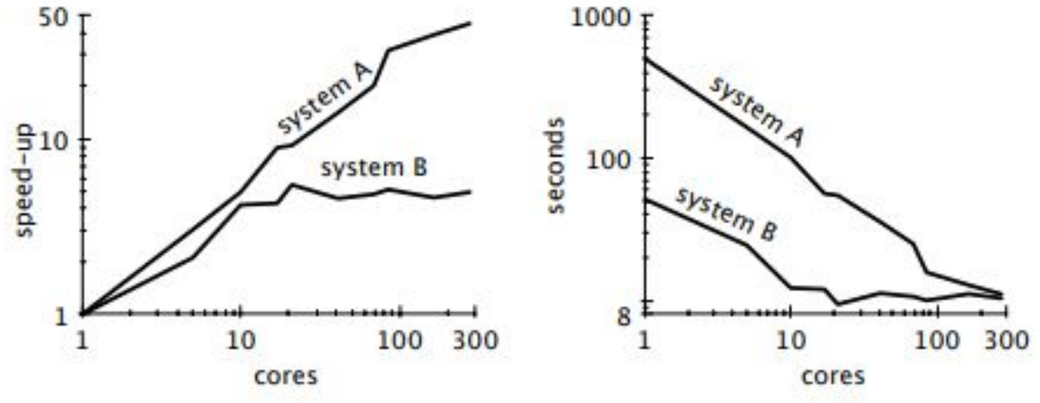


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

Introduction

- Recent work (circa 2014) focused on “scalability”, the ability of systems to improve runtime by adding more machines to a distributed computation.
- But scalability, in and of itself, isn’t valuable. Any system can be made arbitrarily “scalable” by introducing parallelizable overheads.
- Scalability is still useful. It’s valuable to be able to add machines to make a computation run faster.
- But a scalable system is only useful **insofar as it allows you to do things you couldn’t do with a non-scalable system.**

2 Basic Graph Computations

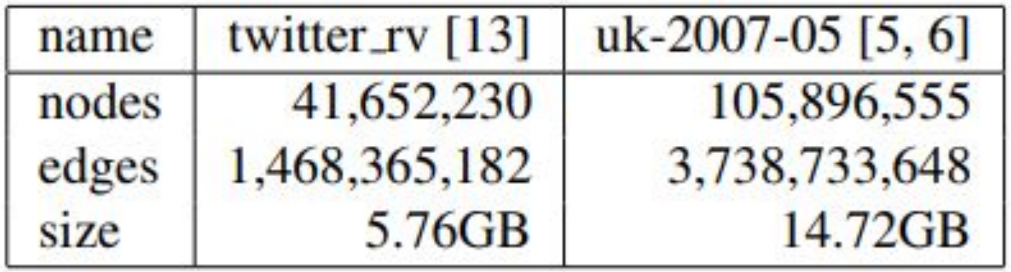
Graph computation has featured prominently in recent SOSP and OSDI conferences, and represents one of the simplest classes of data-parallel computation that is not trivially parallelized. Conveniently, Gonzalez et al. [10] evaluated the latest versions of several graph-processing systems in 2014. We implement each of their tasks using single-threaded C# code, and evaluate the implementations on the same datasets they use (see Table 1).[1]

Basic Graph Computations

- Paper evaluates peers on two problems:
  - PageRank
  - Connected Components

Demo (Graph Algorithms)

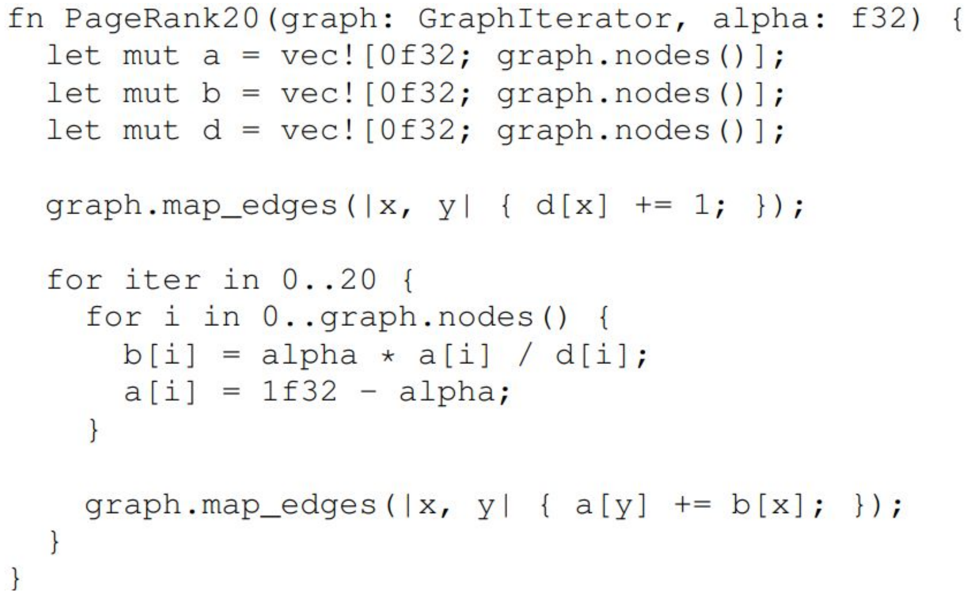
Basic Graph Computations



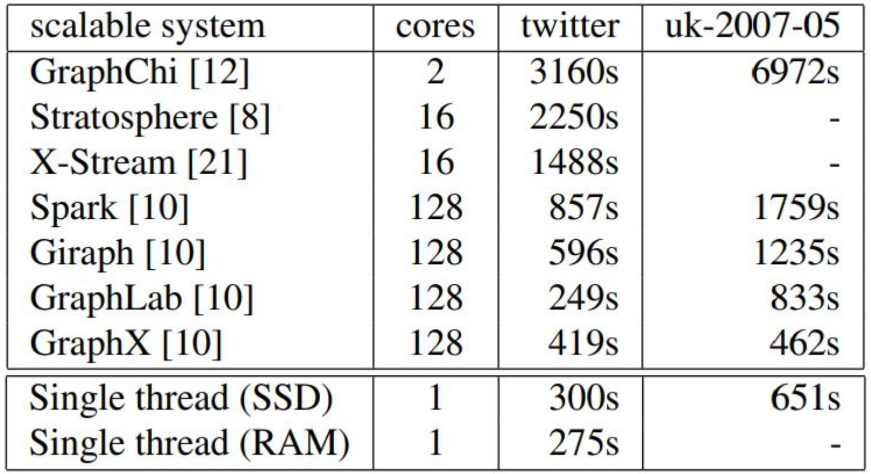
name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Table 1: The "twitter_rv" and "uk-2007-05" graphs.

Basic Graph Computations - PageRank



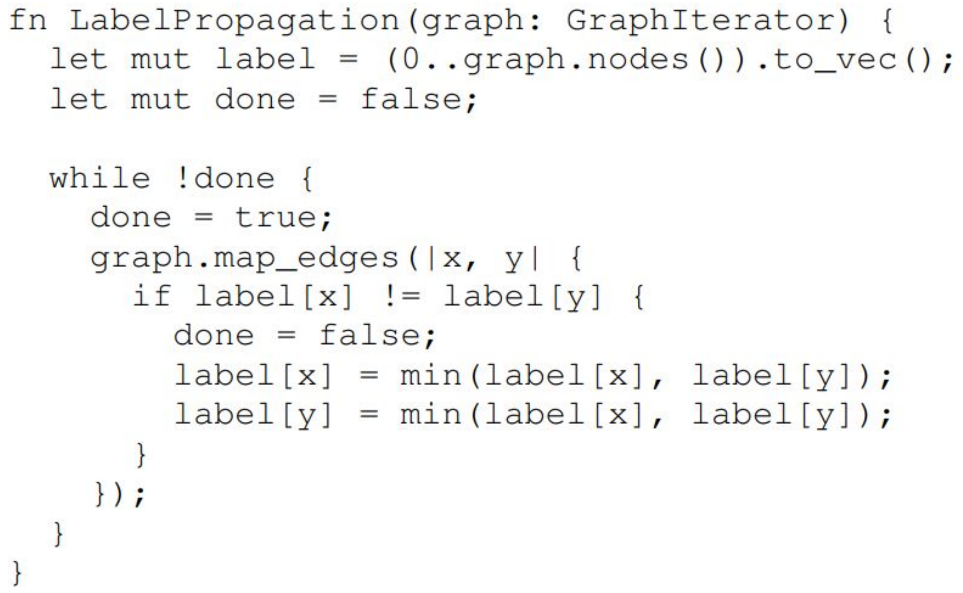
```
fn PageRank20(graph: GraphIterator, alpha: f32) {
  let mut a = vec![0f32; graph.nodes()];
  let mut b = vec![0f32; graph.nodes()];
  let mut d = vec![0f32; graph.nodes()];

  graph.map_edges(|x, y| { d[x] += 1; });

  for iter in 0..20 {
    for i in 0..graph.nodes() {
      b[i] = alpha * a[i] / d[i];
      a[i] = 1f32 - alpha;
    }

    graph.map_edges(|x, y| { a[y] += b[x]; });
  }
}
```



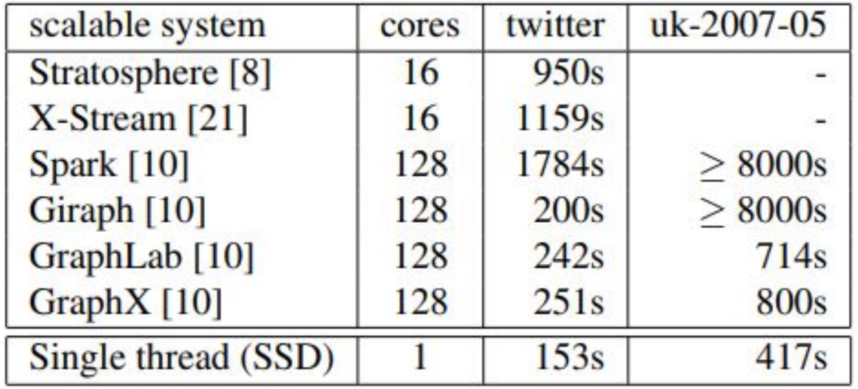
scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

Basic Graph Computations - Label Propagation



```
fn LabelPropagation(graph: GraphIterator) {
  let mut label = (0..graph.nodes()).to_vec();
  let mut done = false;

  while !done {
    done = true;
    graph.map_edges(|x, y| {
      if label[x] != label[y] {
        done = false;
        label[x] = min(label[x], label[y]);
        label[y] = min(label[x], label[y]);
      }
    });
  }
}
```



scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

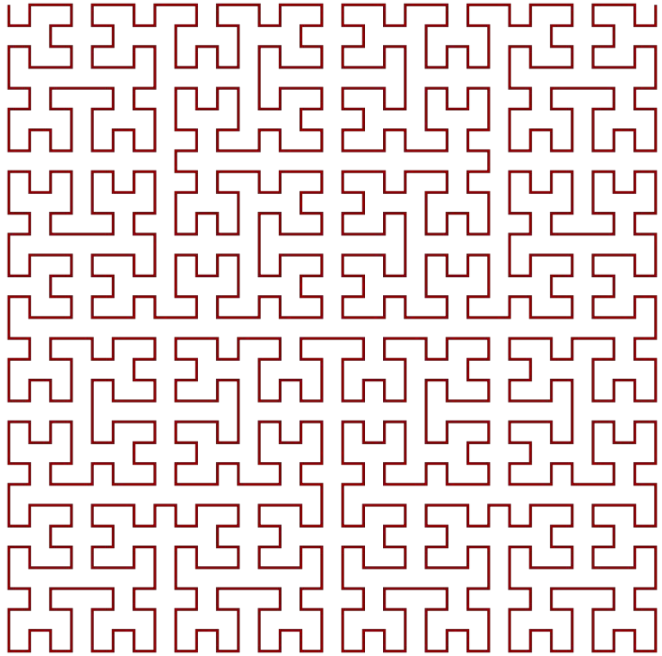
Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.

3 Better Baselines

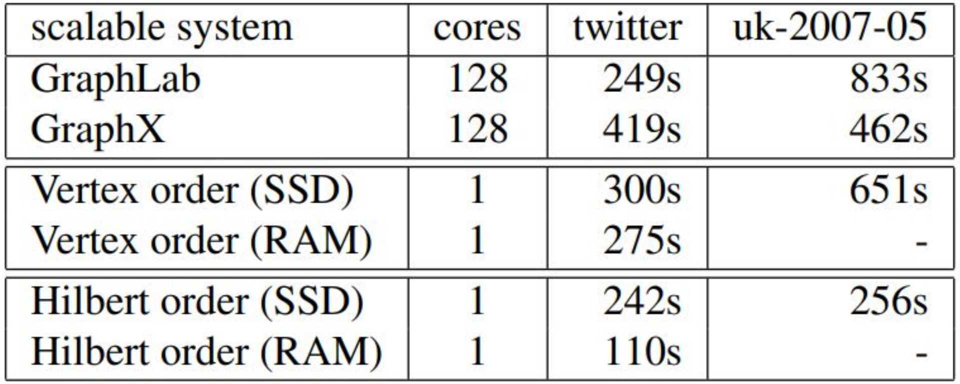
The single-threaded implementations we have presented were chosen to be the simplest, most direct implementations we could think of. There are several standard ways to improve them, yielding single-threaded implementations which strictly dominate the reported performance of the systems we have considered, in some cases by an additional order of magnitude.

Better Baselines - Improving Graph Layout

- Initial baselines stored graph edges in vertex order: edges touching v0 come first, then edges touching v1, etc..
- Can improve performance on real-world graphs by first sorting edges into Hilbert Curve order.
- This helps because it improves locality, resulting in better cache behavior.



Beter Baselines - Improving Graph Layout



scalable system	cores	twitter	uk-2007-05
GraphLab	128	249s	833s
GraphX	128	419s	462s
Vertex order (SSD)	1	300s	651s
Vertex order (RAM)	1	275s	-
Hilbert order (SSD)	1	242s	256s
Hilbert order (RAM)	1	110s	-

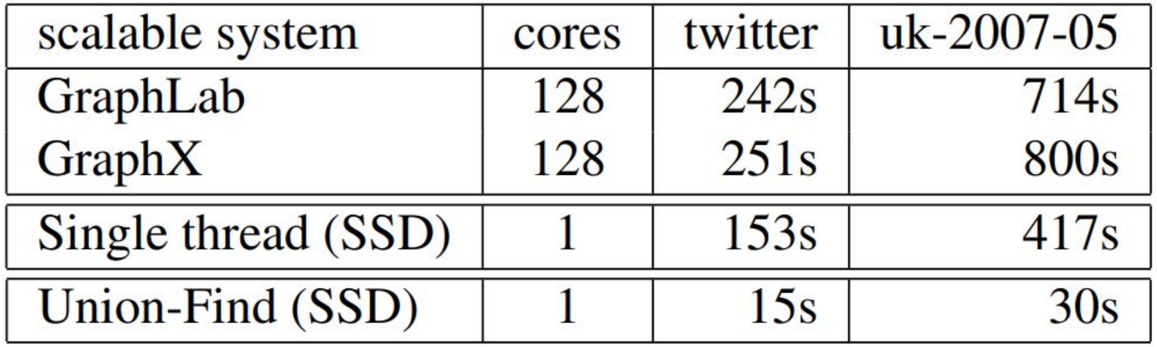
Table 4: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. The single-threaded times use identical algorithms, but with different edge orders.

Better Baselines - Improving Algorithms

- Label propagation is very naive.
  - Spends a lot of time comparing already-propagated labels.
  - Commonly used in distributed contexts because it’s easy to parallelize across nodes.
- Union-Find is a better algorithm.
  - Basic idea of Union-Find is to iterate graph edges and build a spanning-tree for each component.

```
fn UnionFind(graph: GraphIterator) {
  let mut root = (0..graph.nodes()).to_vec();
  let mut rank = [0u8; graph.nodes()];

  graph.map_edges(|mut x, mut y| {
    while (x != root[x]) { x = root[x]; }
    while (y != root[y]) { y = root[y]; }
    if x != y {
      match rank[x].cmp(&rank[y]) {
        Less    => { root[x] = y; },
        Greater => { root[y] = x; },
        Equal   => { root[y] = x; rank[x] += 1; },
      }
    }
  });
}
```

Better Baselines - Improving Algorithms



scalable system	cores	twitter	uk-2007-05
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread (SSD)	1	153s	417s
Union-Find (SSD)	1	15s	30s

4 Applying COST to prior work

Having developed single-threaded implementations, we now have a basis for evaluating the COST of systems. As an exercise, we retrospectively apply these baselines to the published numbers for existing scalable systems.

Applying COST to Prior Work - PageRank

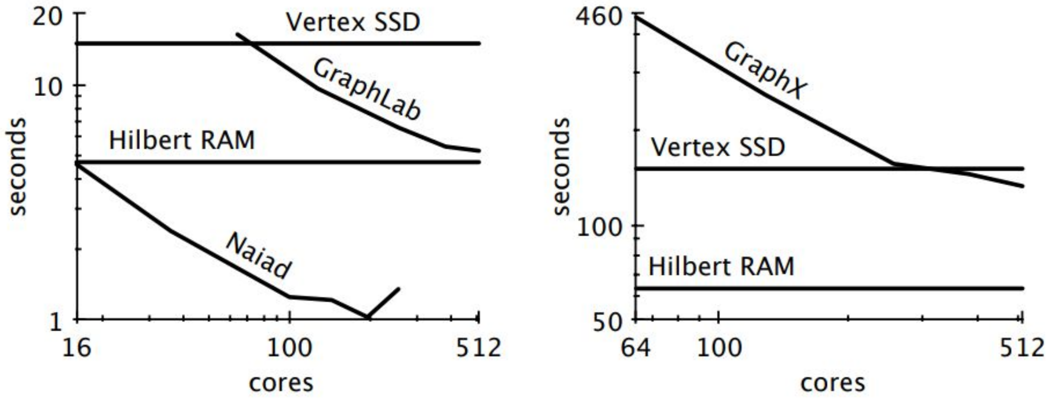


Figure 5: Published scaling measurements for PageRank on twitter_rv. The first plot is the time per warm iteration. The second plot is the time for ten iterations from a cold start. Horizontal lines are single-threaded measurements.

Applying COST to Prior Work - Naiad Union-Find

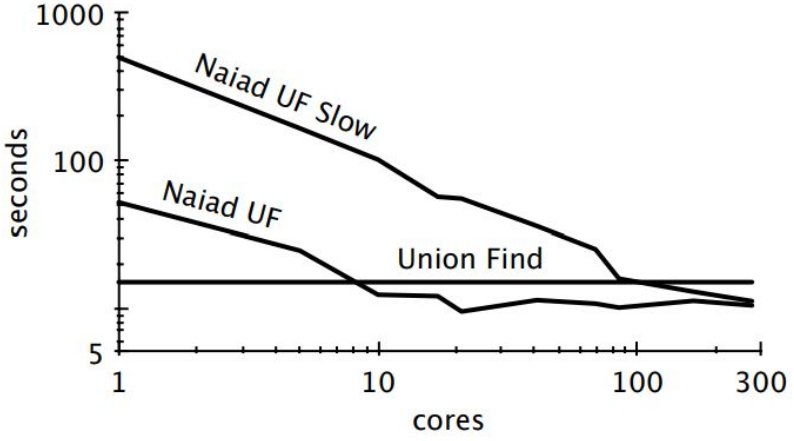


Figure 6: Two Naiad implementations of union find.

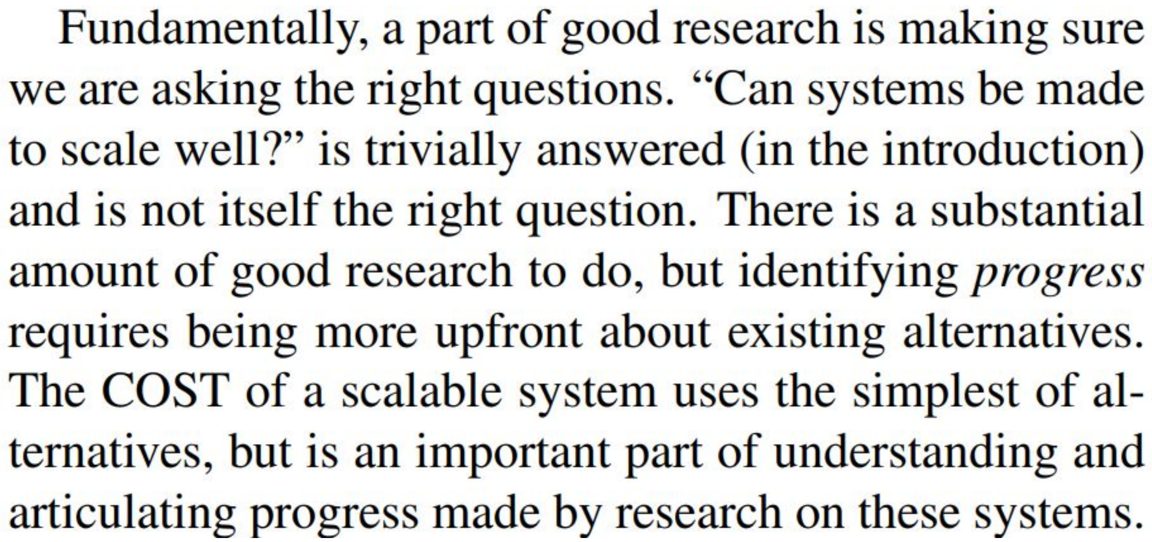
Lessons Learned

- Lots of factors can contribute to the COST of a system:
  - Restricted programming models.
  - Different hardware (laptop vs. cloud instances).
  - System implementation overheads.
- Having high COST doesn’t necessarily make a system bad:
  - System may solve a different problem, or integrate well with a particular target ecosystem, making it easier to use than alternatives for practitioners.
- Nevertheless, COST is a useful metric:
  - It provides a useful baseline for performance expectations.
  - It draws attention to potentially-avoidable inefficiencies.

6 Future directions (for the area)

While this note may appear critical of research in distributed systems, we believe there is still good work to do, and our goal is to provide a framework for measuring and making the best forward progress.

Future Directions



Fundamentally, a part of good research is making sure we are asking the right questions. “Can systems be made to scale well?” is trivially answered (in the introduction) and is not itself the right question. There is a substantial amount of good research to do, but identifying *progress* requires being more upfront about existing alternatives. The COST of a scalable system uses the simplest of alternatives, but is an important part of understanding and articulating progress made by research on these systems.

My Takeaways

- Don’t reach for a fancy hammer when a simple one will do.
  - This idea extends beyond just distributed computation. Any time you’re doing something “fancy”, it’s important to understand how much benefit you’re getting form that fanciness.
  - Examples in other domains: single vs. multiple threads, distributed systems vs. local systems, simple regression vs. machine learning.
- Good baselines are important.
  - Baselines help us figure out if the fancy hammer is needed.
- Distributed graph processing seems hard.
  - Many natural graph algorithms involve some form of (potentially unbounded) “search”.
  - Makes graph algorithms harder to parallelize, because it’s harder to know ahead of time what parts of your computation will need access to which parts of your data.
- Naiad seems neat!

Questions?

Naiad: A Timely Dataflow System

Derek G. Murray Frank McSherry Rebecca Isaacs
Michael Isard Paul Barham Martín Abadi
Microsoft Research Silicon Valley
{derekmur,mcsherry,risaacs,misard,pbar,abadi}@microsoft.com

Abstract

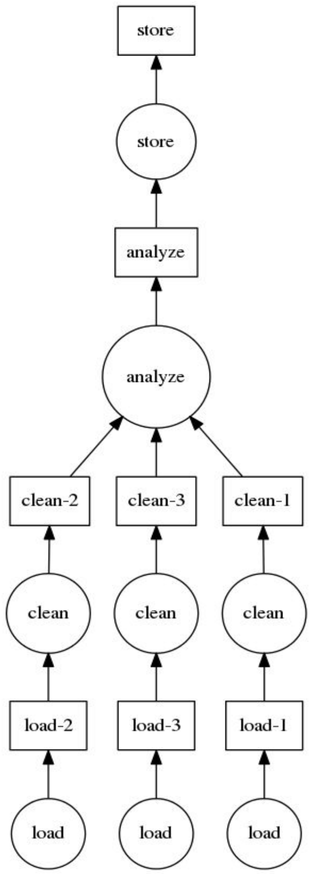
Naiad is a distributed system for executing data parallel, cyclic dataflow programs. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform iterative and incremental computations. Although existing systems offer some of these features, applications that require all three have relied on multiple platforms, at the expense of efficiency, maintainability, and simplicity. Naiad resolves the complexities of combining these features in one framework.

Naiad Overview - Dataflow

- Naiad is a **parallel dataflow** system.
    - Naiad programs are expressed as **graphs**.
    - Nodes in the graph receive messages on incoming edges, perform (possibly stateful) computations, and send results along their output edges.
    - Naiad automatically runs independent portions of the computation in parallel.
        - Parallelism happens within individual nodes (via partitioning).
        - Parallelism happens across multiple nodes (via pipelining, or if they don’t have dependencies).

Dataflow

- Programming model in which you specify the data dependencies of your program “up front”.
    - In exchange, your program usually runs more efficiently.
        - In parallel across a cluster.
        - Incrementally-updating computations.
    - Specifying execution graph ahead of time gives systems more information to use for optimization.
- Dataflow system generally provides a core collection of “built-in” operators.
    - Often SQL-ish vocabulary: map, filter, join, groupby, reduce, etc.
    - May also support custom user-defined operators.
- Many systems impose restrictions on dataflow graph:
    - Acyclic graphs are a common restriction.



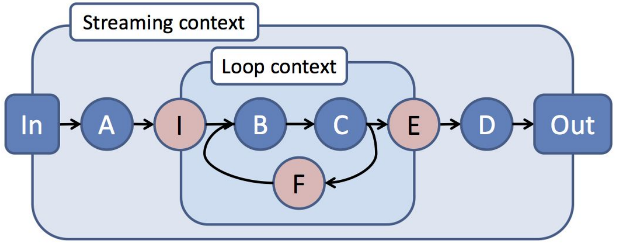
http://matthewrocklin.com/blog/work/2015/07/23/Imperative

Naiad Overview - Timely Dataflow

- Naiad is based on an underlying formalism: **timely dataflow**.
    - Each value flowing through a timely dataflow graph has a **logical timestamp**.
        - Timestamp is initially an integer (the value’s “epoch”) provided by input vertex.
        - Epoch is used to distinguish separate “batches” of input.
        - Timestamp can become more complex in loops (see next slide).
    - Vertices in dataflow graph must implement two methods:
        - `v.OnRecv(e: Edge, m: Message, t: Timestamp)`
            - “Process message from input edge.”
        - `v.OnNotify(t : Timestamp)`
            - “All future messages will have timestamp > t.”
    - Vertices provided with two methods they can call:
        - `this.SendBy(e : Edge, m: Message, t: Timestamp)`
            - “Send message to output edge.”
        - `this.NotifyAt(t : Timestamp)`
            - “I promise I will not emit any more messages <= t”.
            - This is generally what drives computation forward.

Naiad Overview - Timely Dataflow (cont'd)

- Naiad is based on an underlying formalism: **timely dataflow**.
  - Timely dataflow supports **iterative** computations, which feed values back into themselves until some termination condition (e.g., N iterations, fixed point).
    - These computations are expressed as cycles in the dataflow graph.
    - Cycles must be confined to a "loop context".
    - Record timestamp inside a loop context becomes a tuple of (epoch, iteration).
    - Upon exiting the loop context, loop counter gets stripped back off.
  - Structured timestamps mean that Naiad only defines **partial order** on messages. This enables new kinds of parallelism in ways that I don't totally understand.
    - **Example:** For an incremental iterative algorithm, we might be able to run step (1, 0) in parallel with step (0, 1).



Naiad Overview

- Naiad is **low-level**.
    - End-users aren’t expected to use Naiad’s core abstractions directly.
    - Instead, Naiad provides a generalized runtime on top of which many higher-level dataflow systems can be built:
        - SQL/LINQ
        - Incremental Computation
        - Differential Dataflow

Bad News, Good News

- Original Naiad / Timely Dataflow system is no longer under development:
  - MSFT cancelled the project when they closed the Silicon Valley research office.
  - Code can be found here: https://github.com/MicrosoftResearch/Naiad.
- Timely Dataflow has since been rewritten in Rust:
  - https://github.com/TimelyDataflow/timely-dataflow
- Differential Dataflow has also been rewritten:
  - https://github.com/TimelyDataflow/differential-dataflow

More Resources

Papers and Articles

- Naiad: A Timely Dataflow System
- Timely Dataflow MDBook
- Differential Dataflow
- Foundations of Differential Dataflow
- Frank McSherry's Blog

Talks and Videos

- Naiad: a timely dataflow system
- Timely DataFlow in Rust
- Introducing Project Naiad and Differential Dataflow

Questions?