# Next generation data-parallel dataflow

Frank McSherry

http://github.com/frankmcsherry

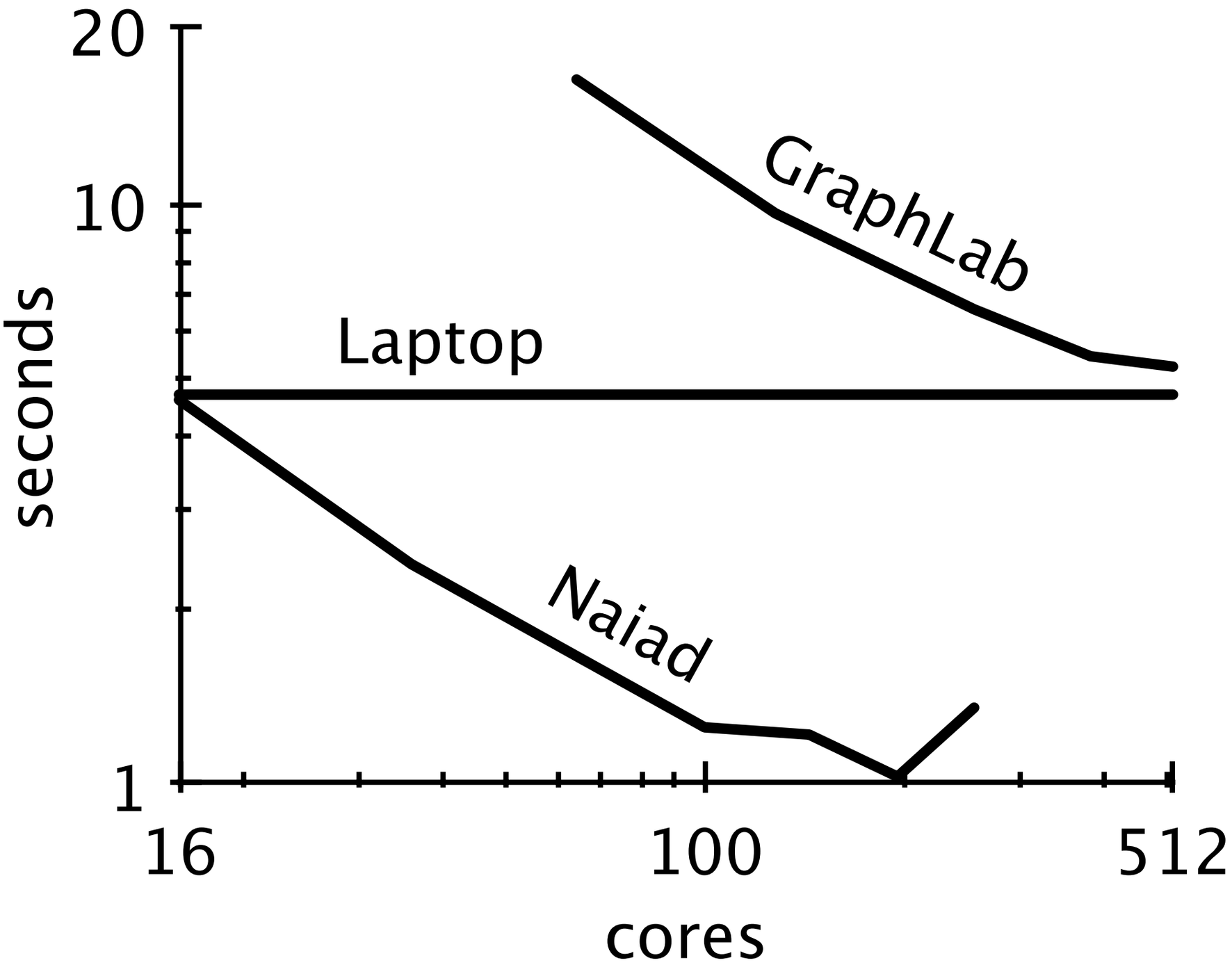
# What’s wrong with the current generation?

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| **Spark** | 128 | 857s | 1759s |
| **Giraph** | 128 | 596s | 1235s |
| **GraphLab** | 128 | 249s | 833s |
| **GraphX** | 128 | 419s | 462s |

from Gonzalez *et al*., OSDI 2014

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |
| Laptop | 1 | 300s | 651s |

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |
| Laptop | 1 | ~~300s~~ 110s | ~~651s~~ 256s |


20
10
1
seconds
Laptop
GraphLab
Naiad
16
100
512
cores

| Connectivity | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| **Spark** | 128 | 1784s | 8000s+ |
| **Giraph** | 128 | 200s | 8000s+ |
| **GraphLab** | 128 | 242s | 714s |
| **GraphX** | 128 | 251s | 800s |

from Gonzalez *et al*., OSDI 2014

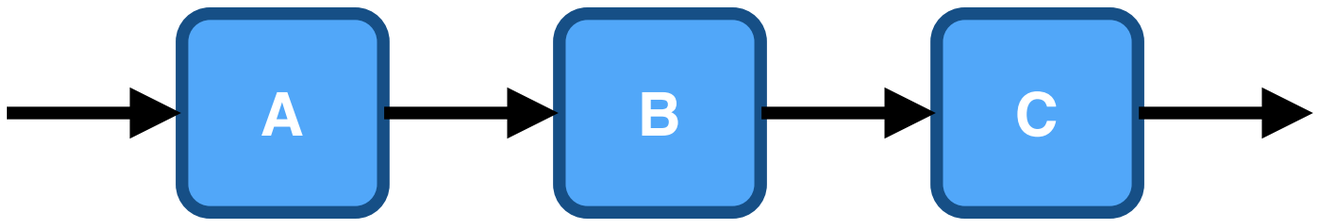
| Connectivity | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 1784s | 8000s+ |
| Giraph | 128 | 200s | 8000s+ |
| GraphLab | 128 | 242s | 714s |
| GraphX | 128 | 251s | 800s |
| Laptop | 1 | 153s | 417s |

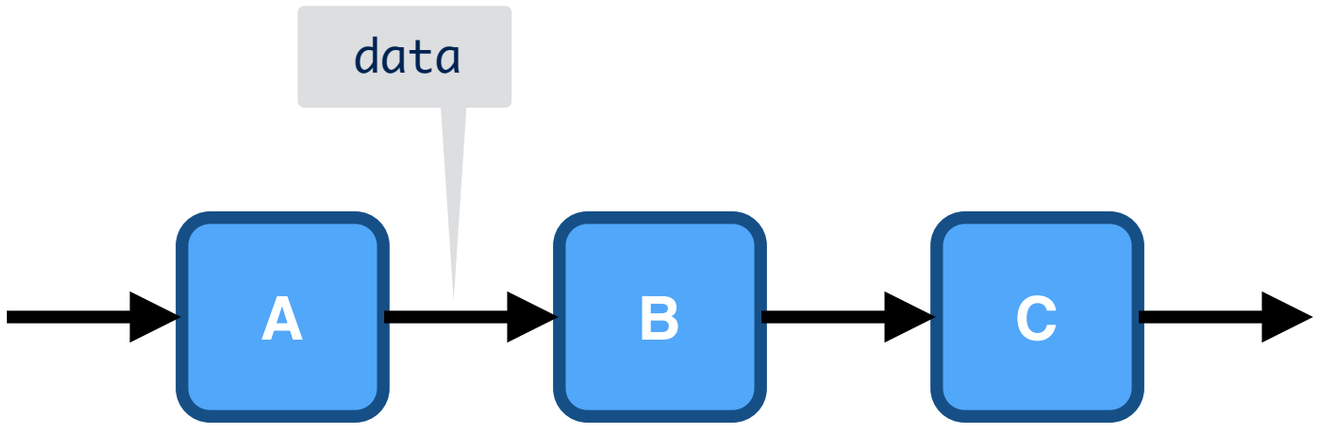
| Connectivity | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| **Spark** | 128 | 1784s | 8000s+ |
| **Giraph** | 128 | 200s | 8000s+ |
| **GraphLab** | 128 | 242s | 714s |
| **GraphX** | 128 | 251s | 800s |
| **Laptop** | 1 | ~~153s~~ 15s | ~~417s~~ 30s |

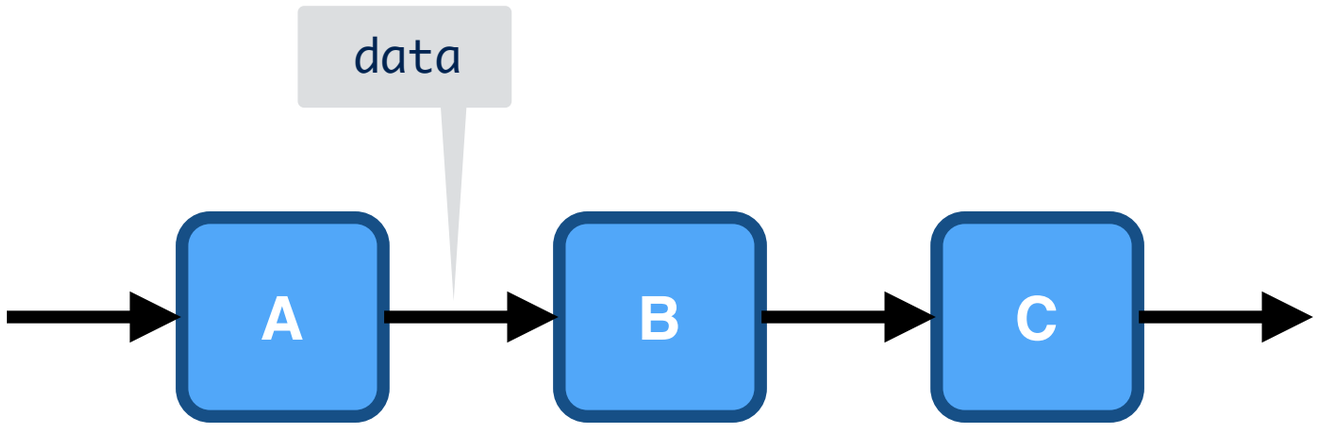
# Current-gen issues

1. Abysmal performance
2. Limited expressivity
    a. Stateless operators
    b. Acyclic dataflow graphs

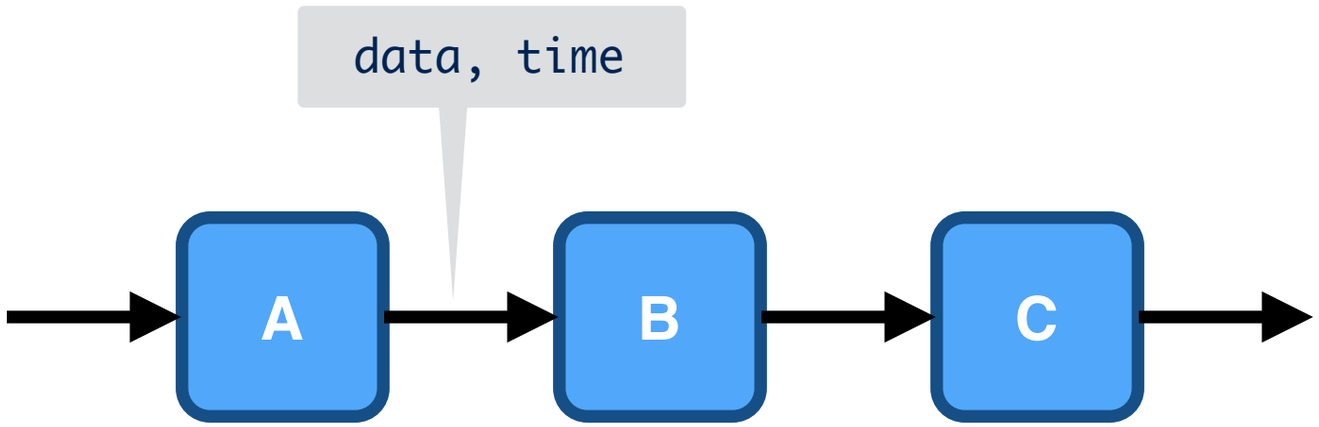
# Let’s build a dataflow system!


A
B
C

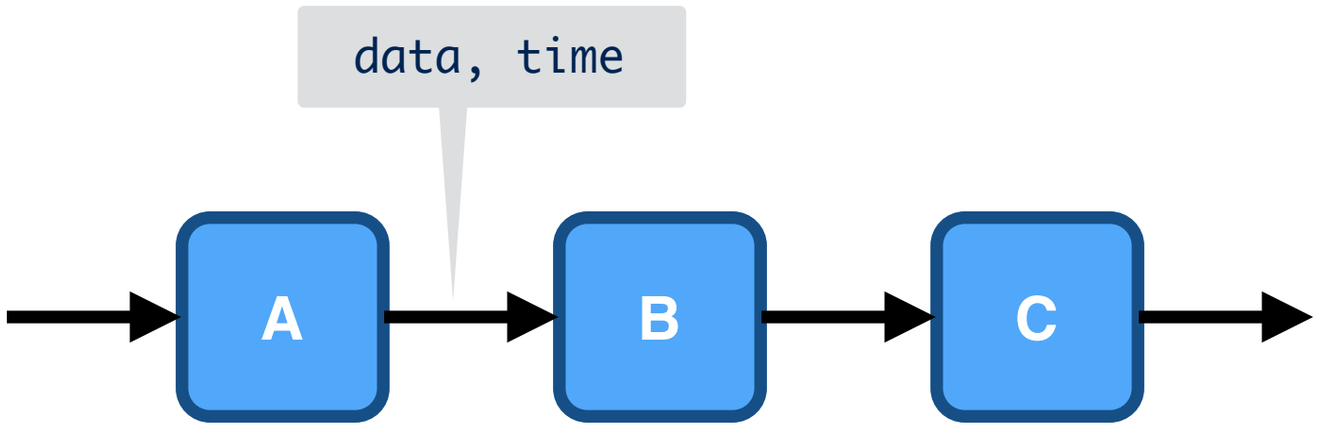

data
A
B
C



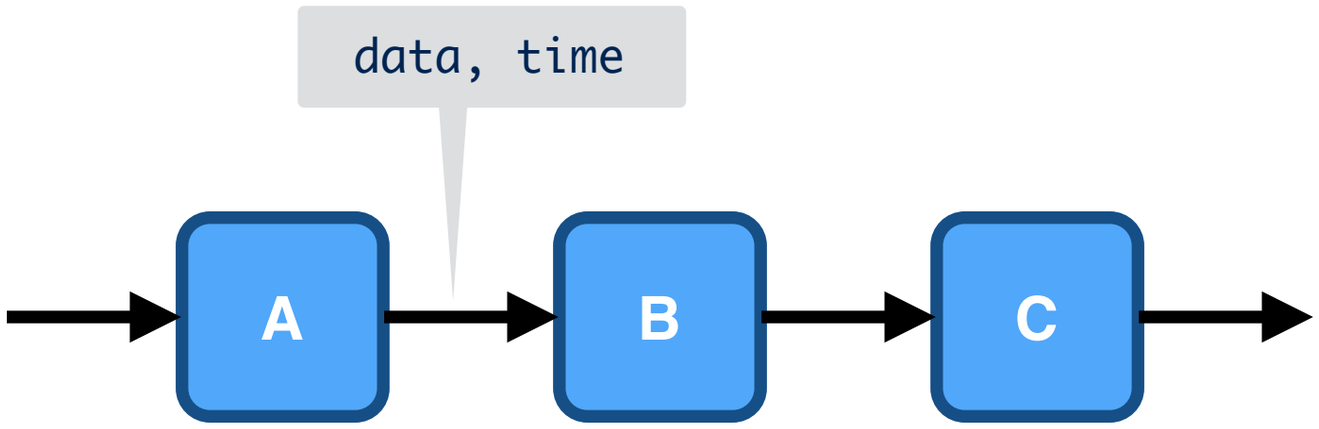

```
Recv(data):          /* your code here  */
Send(data):          /* your code calls */
```



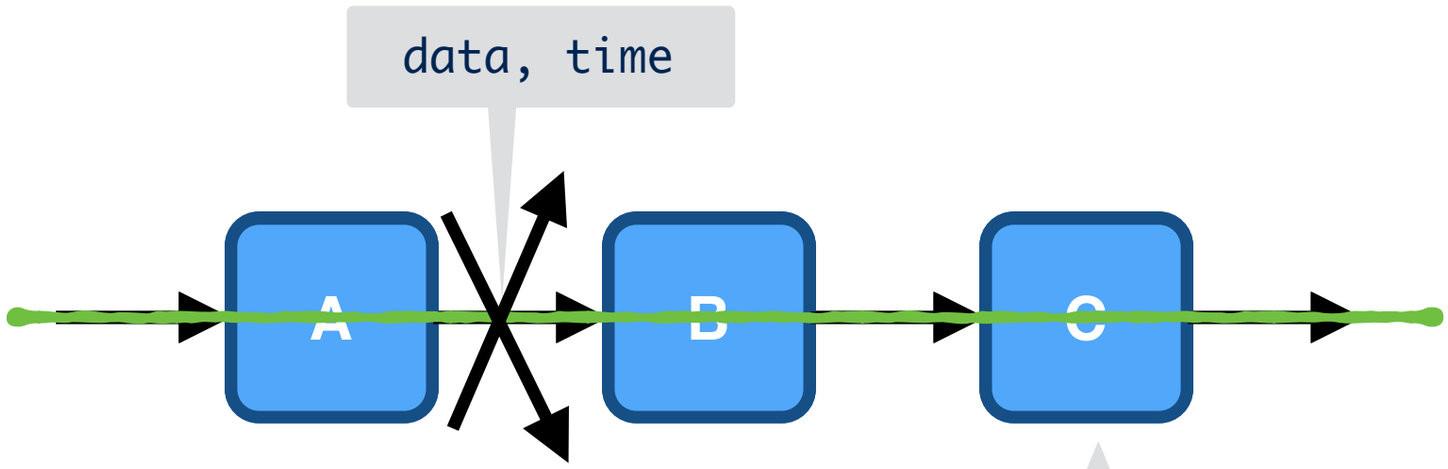

```
Recv(data):          /* your code here  */
Send(data):          /* your code calls */
```



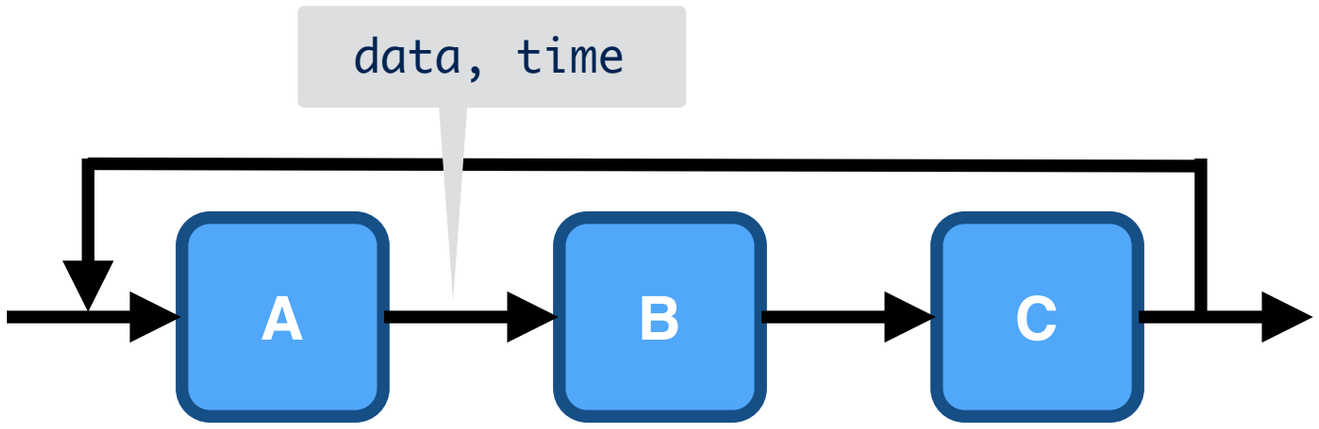

```
Recv(data, time):   /* your code here  */
Send(data, time):   /* your code calls */
```



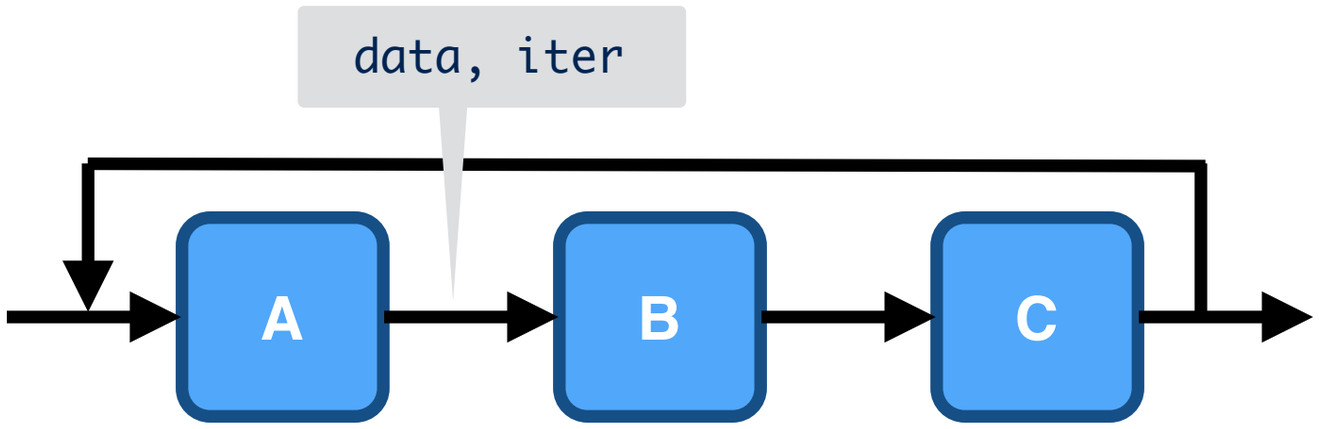

```
Recv(data, time):   /* your code here  */
Send(data, time):   /* your code calls */

Notify(time):       /* your code here  */
```



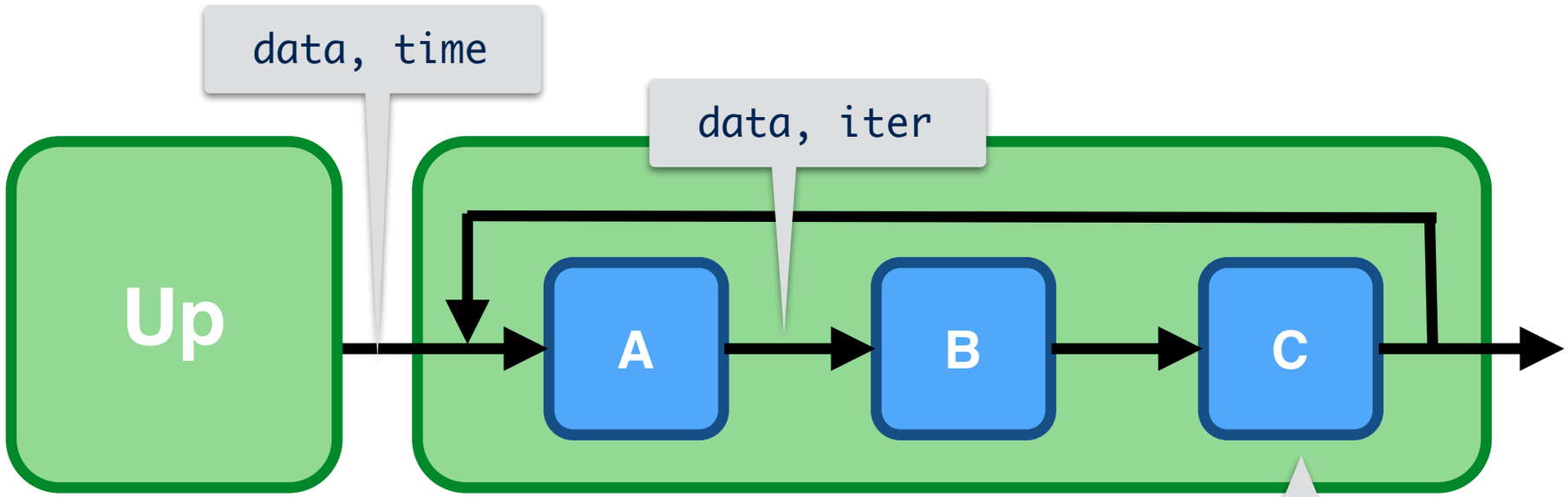

```
Recv(data, time):   /* your code here  */
Send(data, time):   /* your code calls */

Notify(time):       /* your code here  */
```



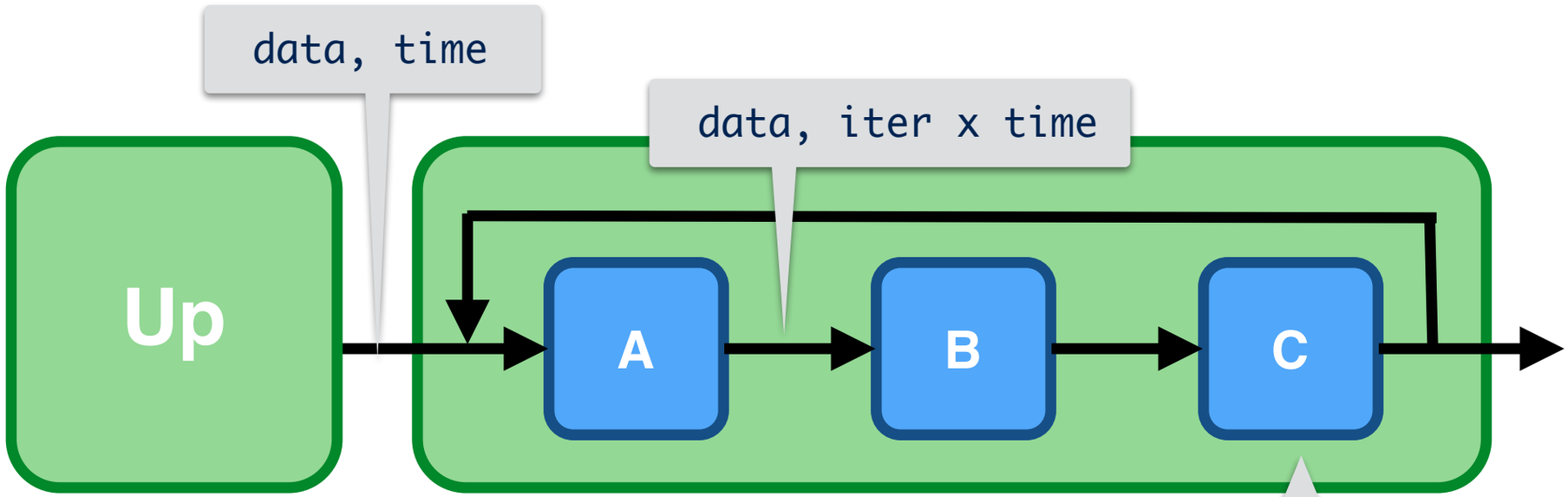

```
Recv(data, time):   /* your code here  */
Send(data, time):   /* your code calls */

Notify(time):       /* your code here  */
```



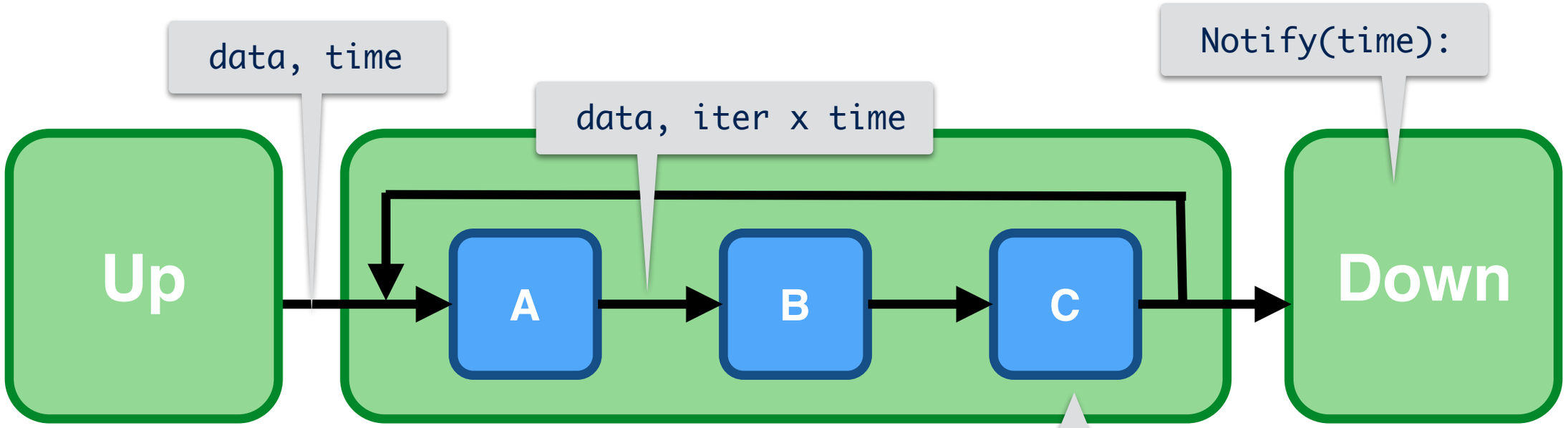

```
Recv(data, iter):   /* your code here  */
Send(data, iter):   /* your code calls */

Notify(iter):       /* your code here  */
```




```
Recv(data, iter):   /* your code here  */
Send(data, iter):   /* your code calls */

Notify(iter):       /* your code here  */
```



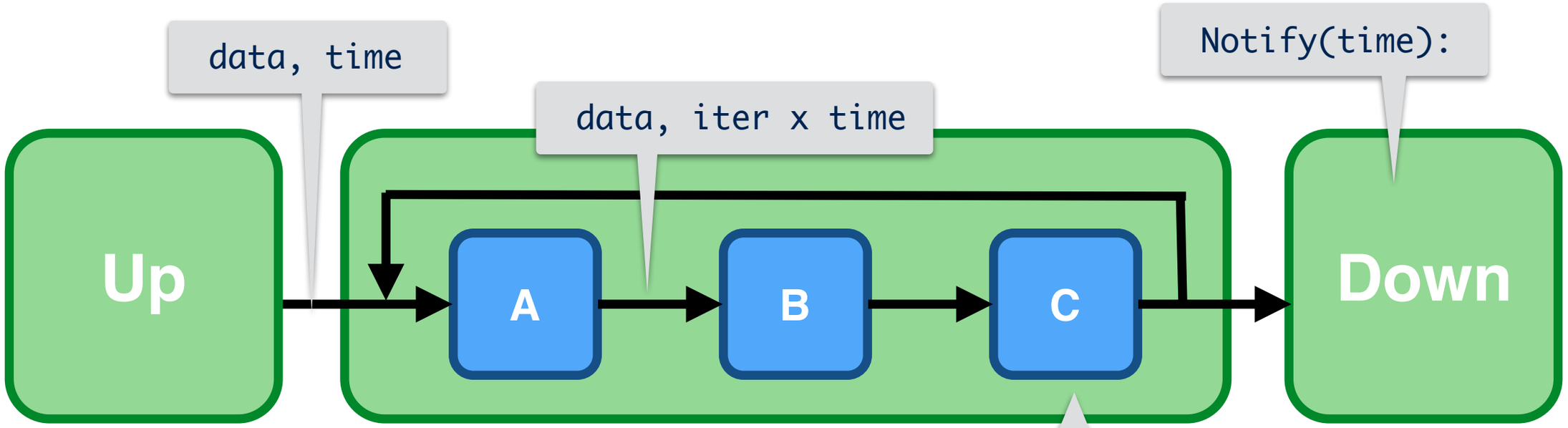

```
Recv(data, iter):   /* your code here  */
Send(data, iter):   /* your code calls */

Notify(iter):       /* your code here  */
```



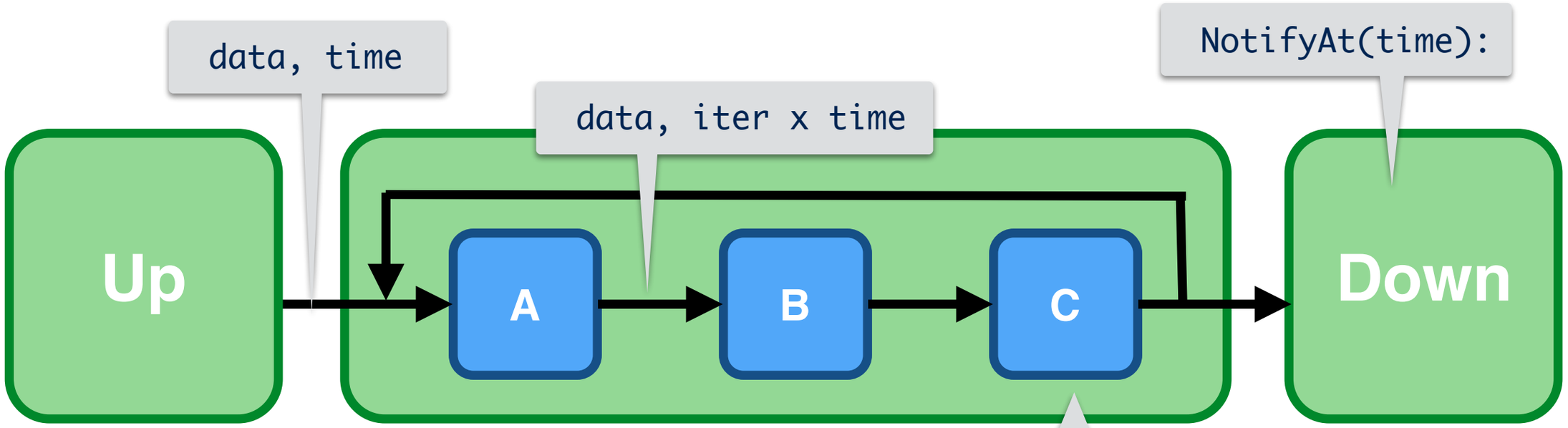

```
Recv(data, iter x time):    /* your code */
Send(data, iter x time):    /* you call  */

Notify(iter x time):        /* your code */
```

# Timely Dataflow

"Naiad: a timely dataflow system"
[SOSP 2013]

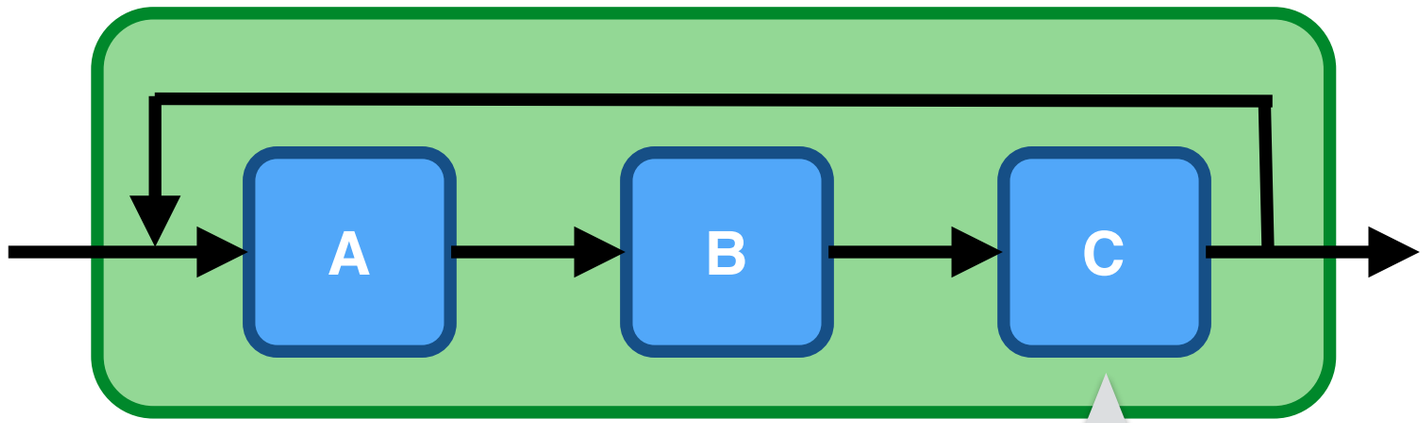



```
Recv(data, iter x time):    /* your code */
Send(data, iter x time):    /* you call  */

Notify(iter x time):        /* your code */
```

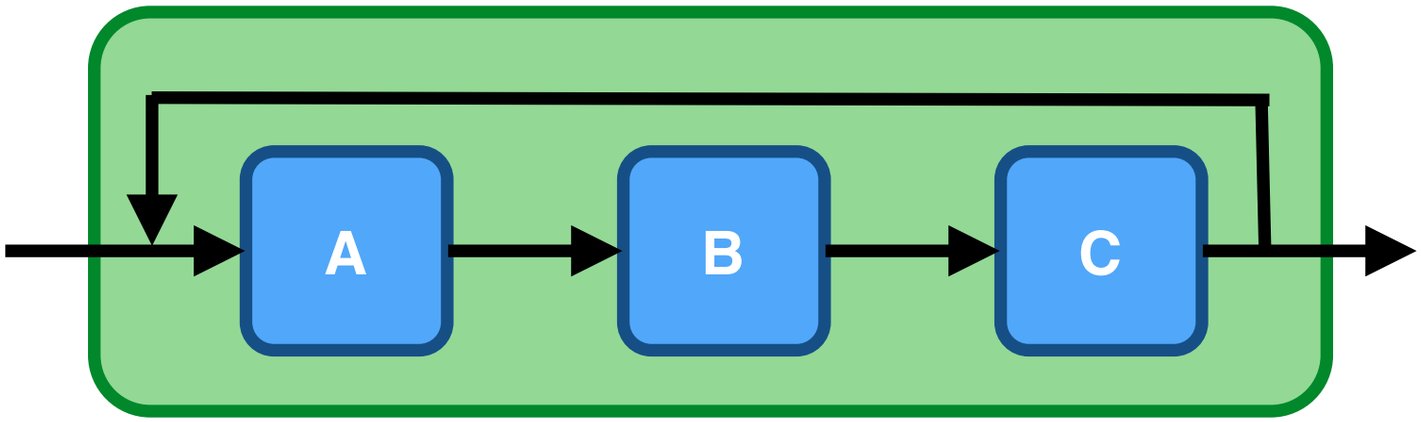



```
Recv(data, iter x time):    /* your code */
Send(data, iter x time):    /* you call  */

Notify(iter x time):        /* your code */
OnNotify(iter x time):      /* your code */
NotifyAt(iter x time):      /* you call  */
```

**Super Important**




```
Recv(data, “time”):    /* your code */
Send(data, “time”):    /* you call  */

OnNotify(“time”):      /* your code */
NotifyAt(“time”):      /* you call  */
```




Outstanding Work

B time : pointstamp

Notification Requests

C time

A time + 1

# Distributed Ref-Counting

Worker 1

1x B time

Worker 2

1x B time

-1x B time

1x C time

# Distributed Ref-Counting

Worker 1

1x C time

Worker 2

1x C time

# What's new / next?

Design

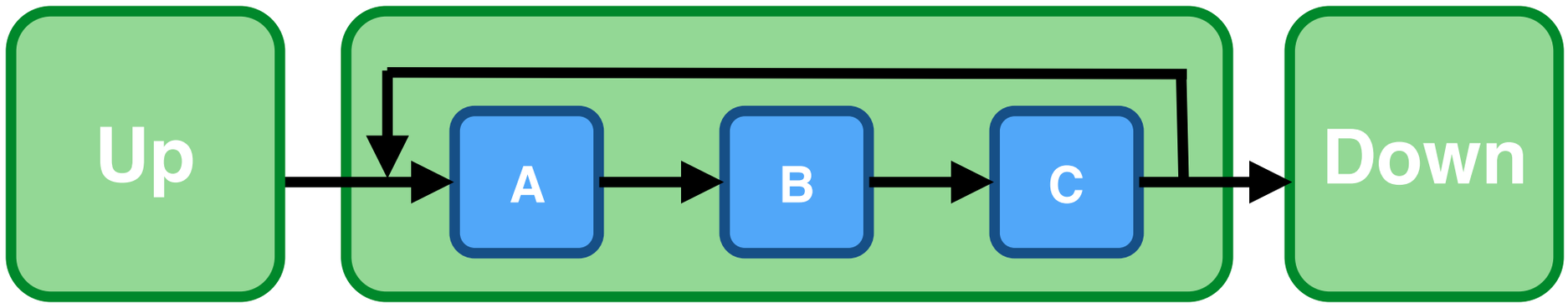
Doing all the math right with partial orders and graph theory and stuff

Implementation

Use Rust instead of C#

# Design

Naiad's progress tracking used a flat dataflow graph




Timestamps were **UInt64[]**s, despite static knowledge

Homogenous type, induces total order

Heap allocated

# Design

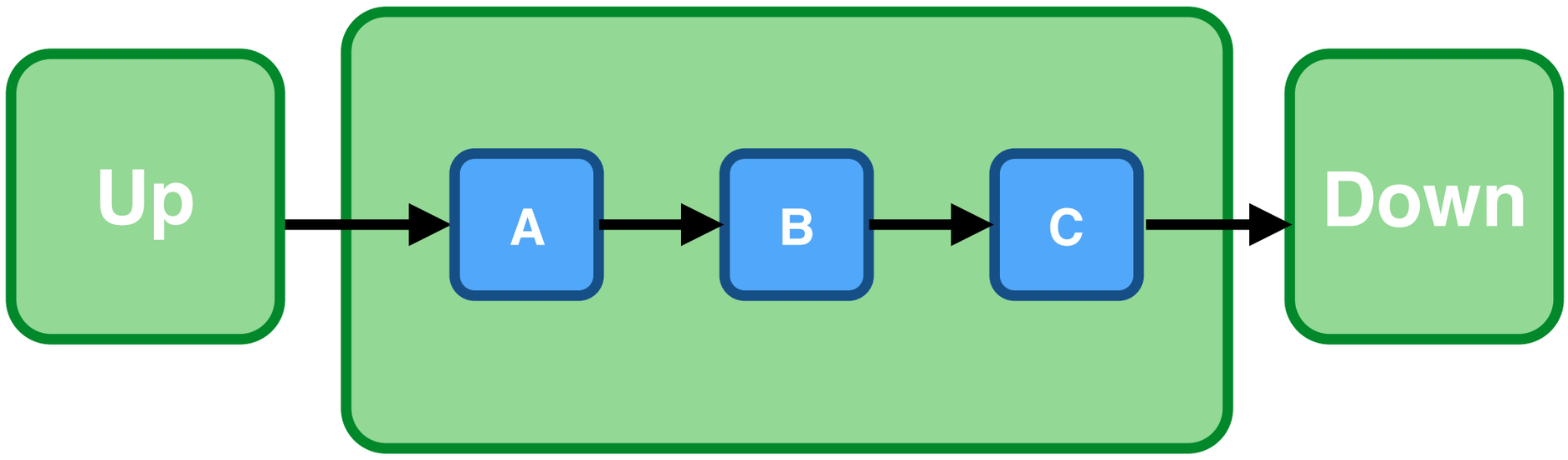
Naiad's progress tracking used a flat dataflow graph


Abstraction boundary allows non-uniform types, logic

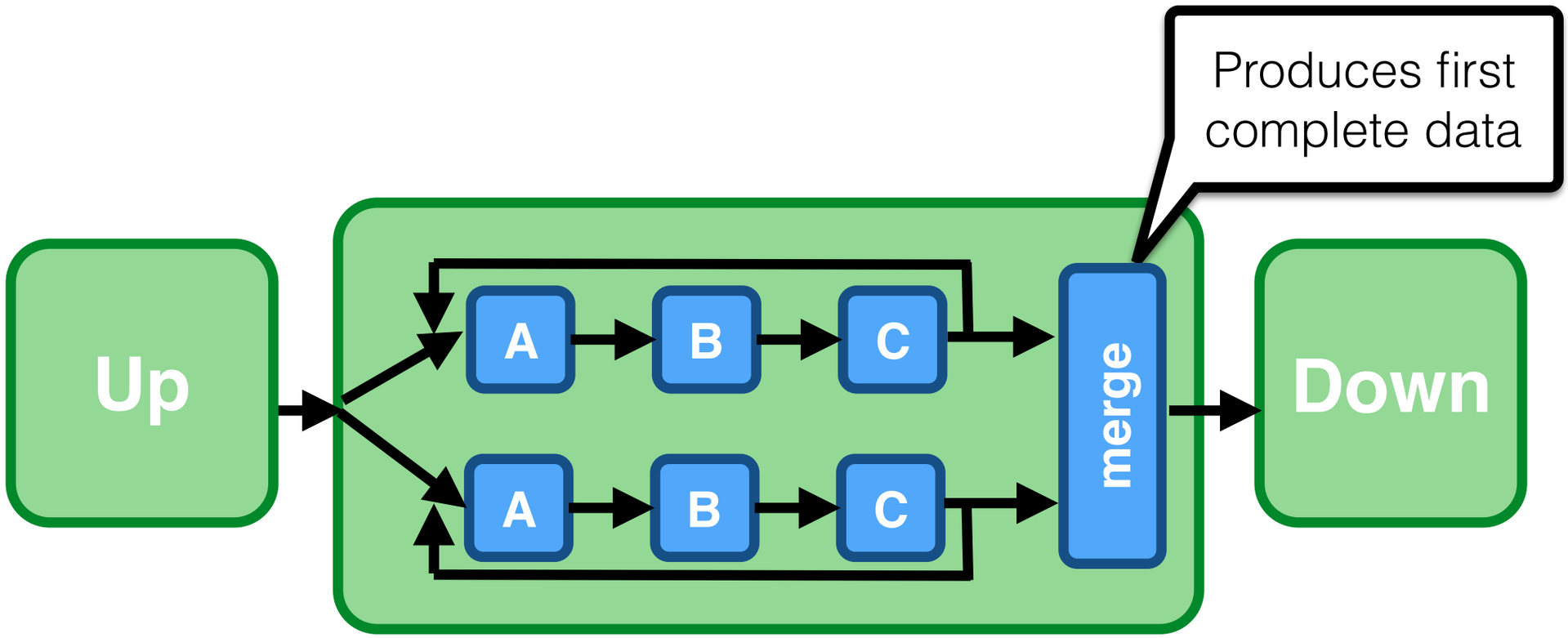
Detail: scopes must present input-output reachability

awesome

# Neat example



# Neat example

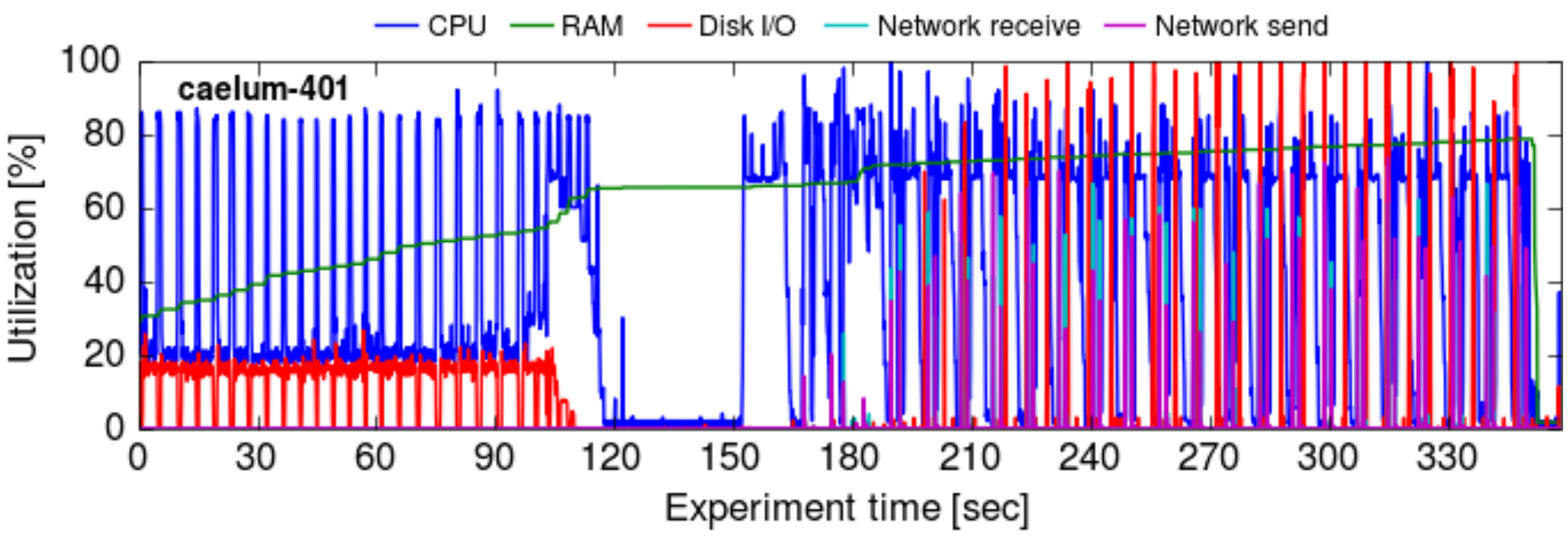



Active-active replication (high availability)

Implemented at language vs system level

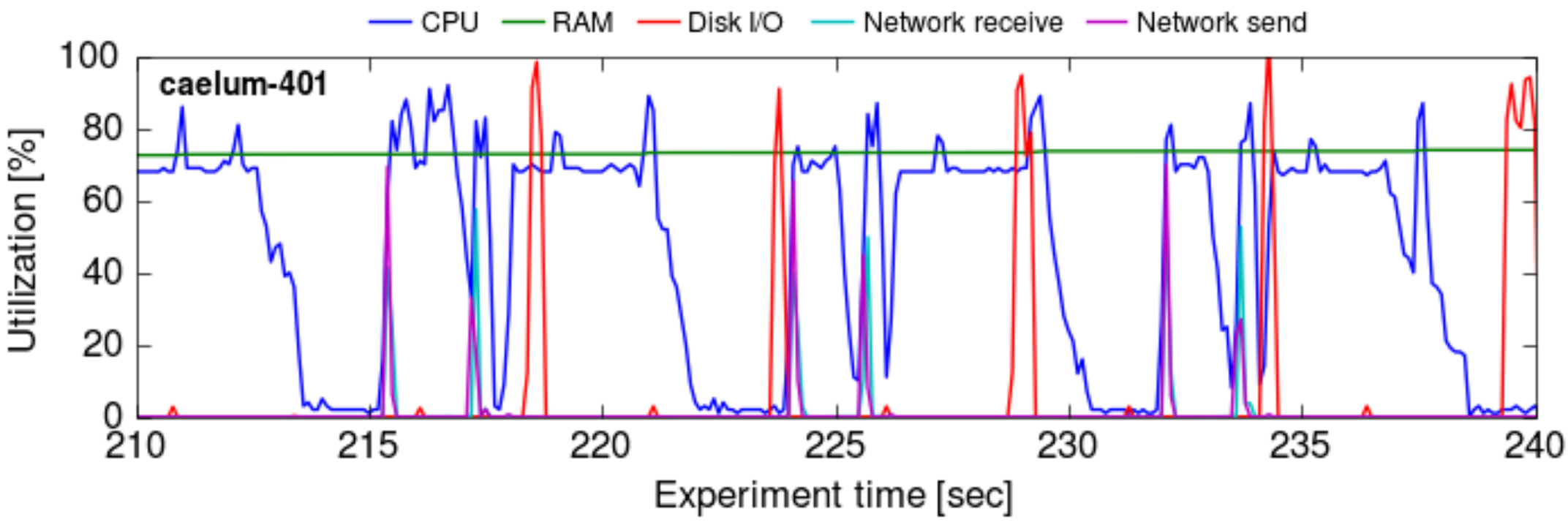
# Some Evaluation

# 20x PageRank



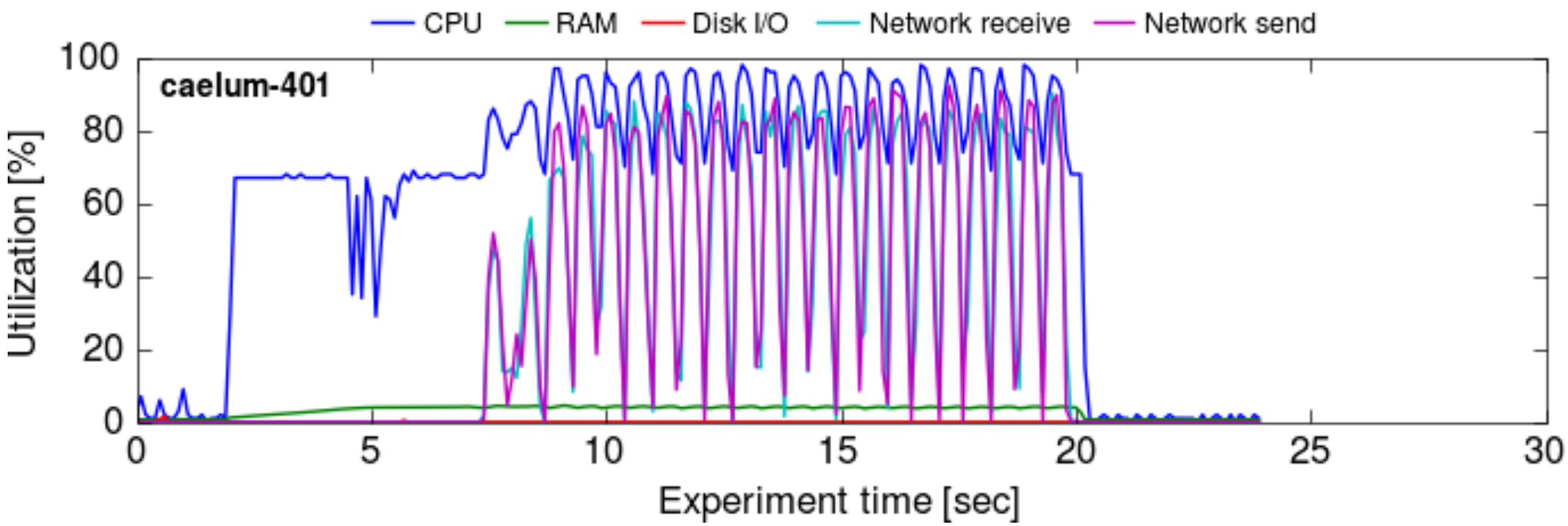

GraphX

# 20x PageRank




GraphX

# 20x PageRank



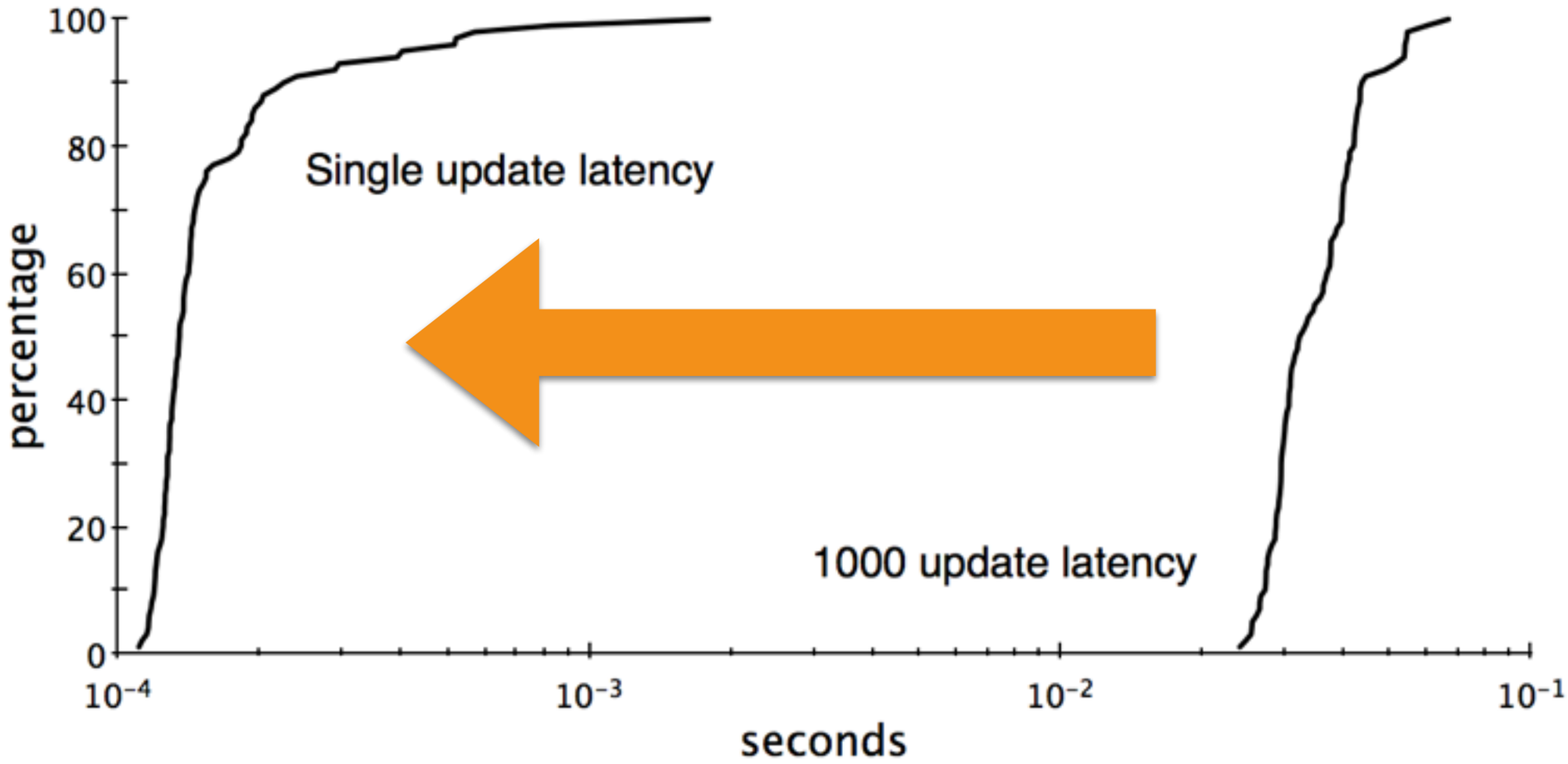

Timely dataflow in Rust

# Differential Dataflow

Fully incremental MapReduce + FixedPoint

# Differential Dataflow

## Fully incremental MapReduce + FixedPoint



Thank you