Timely Dataflow

Frank McSherry
mcsherry@gmail.com

Data-parallel dataflow

Too Simple programming model

Sub Performant implementations

Some background

from [Gonzalez et al, OSDI2014]

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

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		

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

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		

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

What went so wrong?

Too Simple programming model

Sub Performant implementations

Back to basics

(of data-parallel computation)

Batch Dataflow

```
fn main () {

    input.map_reduce(|record| keys_vals(record),
                     |key, vals| reducer(key, vals))

}
```

Parallelism!

Batch Dataflow

```
fn my_func(input) {

    input.map_reduce(|record| keys_vals(record),
                     |key, vals| reducer(key, vals))

}
```

Parallelism!

Batch Dataflow

```
fn my_func(input) {

    input.map_reduce(|record| keys_vals(record),
                     |key, vals| reducer(key, vals))

}

fn main() {
    my_func(my_func(input));



}
```

Parallelism!

Streaming Dataflow

Batch Dataflow

```
fn my_func(input) {

    input.map_reduce(|record| keys_vals(record),
                      |key, vals| reducer(key, vals))

}

fn main() {
    my_func(my_func(input));
     for record in source {
         input.push(record);
     }
}
```

State!

Streaming Datoflow

```
fn my_func(input) {

    input.map_reduce(|record| keys_vals(record),
                     |key, vals| reducer(key, vals))

}

fn main() {
    let output = my_func(my_func(input));
    for record in source {
        input.push(record);
    }
}
```

Streaming Dataflow / Iterative Dataflow

```
fn my_func(input) {

    input.map_reduce(|record| keys_vals(record),
                     |key, vals| reducer(key, vals))

}

fn main() {
    let output = my_func(my_func(input));
    for record in output {
        input.push(record);
    }
}
```

Iteration!

Where is this going?

What is missing?

Structured Programming Languages

Iterative dataflow is like one while loop with if/then/else.

Need modular abstractions which still compose.

Zero-overhead implementations

Abstractions makes a software stack feel valuable.

Important to expose a performant low-level interface.

Structured Programming Languages

a worked example: graph connectivity

```
// repeatedly improves labels until fixed point
fn connected_components(edges: Stream<Edge>)
        -> Stream<Label> {

    edges.map(|edge| Label::new(edge.src, edge.src))
         .iterate(|labels| local_min(edges, labels))
}


// improves labeling by considering labels of neighbors
fn local_min(edges: Stream<Edge>, labels: Stream<Label>)
        -> Stream<Label> {

    labels.join(edges, |l| l.src, |e| e.src)
          .map(|(l,e)| Label::new(e.dst, l.lbl))
          .concat(labels)
          .argmin(|label| label.src, |label| label.lbl)
}
```

```
// removes edges with differently labeled endpoints
fn filter_by_label(edges: Stream<Edge>) -> Stream<Edge> {

    let labels = connected_components(edges);

    edges.join(labels, |e| e.src,      |l| l.src)
         .join(labels, |(e,l)| e.dst, |l| l.src)
         .filter(|((e, l1), l2)| l1 == l2)
         .map(|((edge), _), _)| edge)
}
```

```
// repeatedly improves labels until fixed point
fn connected_components(edges: Stream<Edge>)
        -> Stream<Label> {

    edges.map(|edge| Label::new(edge.src, edge.src))
         .iterate(|labels| local_min(edges, labels))
}

// improves labeling by considering labels of neighbors
fn local_min(edges: Stream<Edge>, labels: Stream<Label>)
        -> Stream<Label> {

    labels.join(edges, |l| l.src, |e| e.src)
          .map(|(l,e)| Label::new(e.dst, l.lbl))
          .concat(labels)
          .argmin(|label| label.src, |label| label.lbl)
}
```

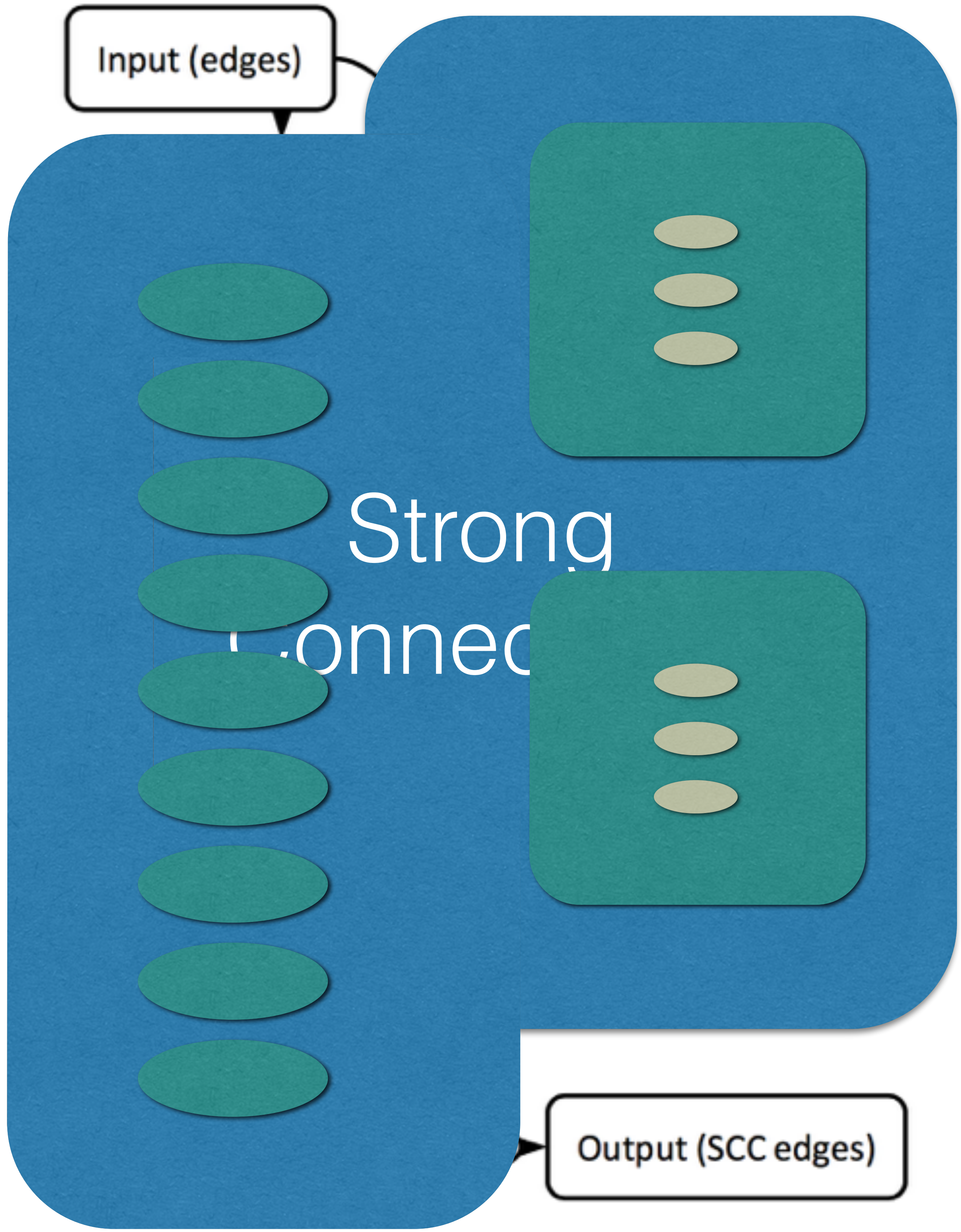
```
// repeatedly filters by label, then flips
fn strong_connectivity(graph: Stream<Edge>) {

    graph.iterate(|edges| edges.filter_by_label()
                               .map(|e| e.reverse())
                               .filter_by_label()
                               .map(|e| e.reverse())) }
```

```
// removes edges with differently labeled endpoints
fn filter_by_label(edges: Stream<Edge>) -> Stream<Edge> {

    let labels = connected_components(edges);

    edges.join(labels, |e| e.src,     |l| l.src)
         .join(labels, |(e,l)| e.dst, |l| l.src)
         .filter(|((e, l1), l2)| l1 == l2)
         .map(|((edge), _), _)| edge)
}
```

```
// repeatedly improves labels until fixed point
fn connected_components(edges: Stream<Edge>)
        -> Stream<Label> {

    edges.map(|edge| Label::new(edge.src, edge.src))
         .iterate(|labels| local_min(edges, labels))
}
```

```
// improves labeling by considering labels of neighbors
fn local_min(edges: Stream<Edge>, labels: Stream<Label>)
        -> Stream<Label> {

    labels.join(edges, |l| l.src, |e| e.src)
          .map(|(l,e)| Label::new(e.dst, l.lbl))
          .concat(labels)
          .argmin(|label| label.src, |label| label.lbl)
}
```

```
// repeatedly filters by label, then flips
fn strong_connectivity(graph: Stream<Edge>) {

    graph.iterate(|edges| edges.filter_by_label()
                               .map(|e| e.reverse())
                               .filter_by_label()
                               .map(|e| e.reverse())) }



// removes edges with differently labeled endpoints
fn filter_by_label(edges: Stream<Edge>) -> Stream<Edge> {

    let labels = connected_components(edges);

    edges.join(labels, |e| e.src,     |l| l.src)
         .join(labels, |(e,l)| e.dst, |l| l.src)
         .filter(|((e, l1), l2)| l1 == l2)
         .map(|((edge), _), _)| edge)
}


// repeatedly improves labels until fixed point
fn connected_components(edges: Stream<Edge>)
        -> Stream<Label> {

    edges.map(|edge| Label::new(edge.src, edge.src))
          .iterate(|labels| local_min(edges, labels))
}

// improves labeling by considering labels of neighbors
fn local_min(edges: Stream<Edge>, labels: Stream<Label>)
        -> Stream<Label> {

    labels.join(edges, |l| l.src, |e| e.src)
          .map(|(l,e)| Label::new(e.dst, l.lbl))
          .concat(labels)
          .argmin(|label| label.src, |label| label.lbl)
}
```

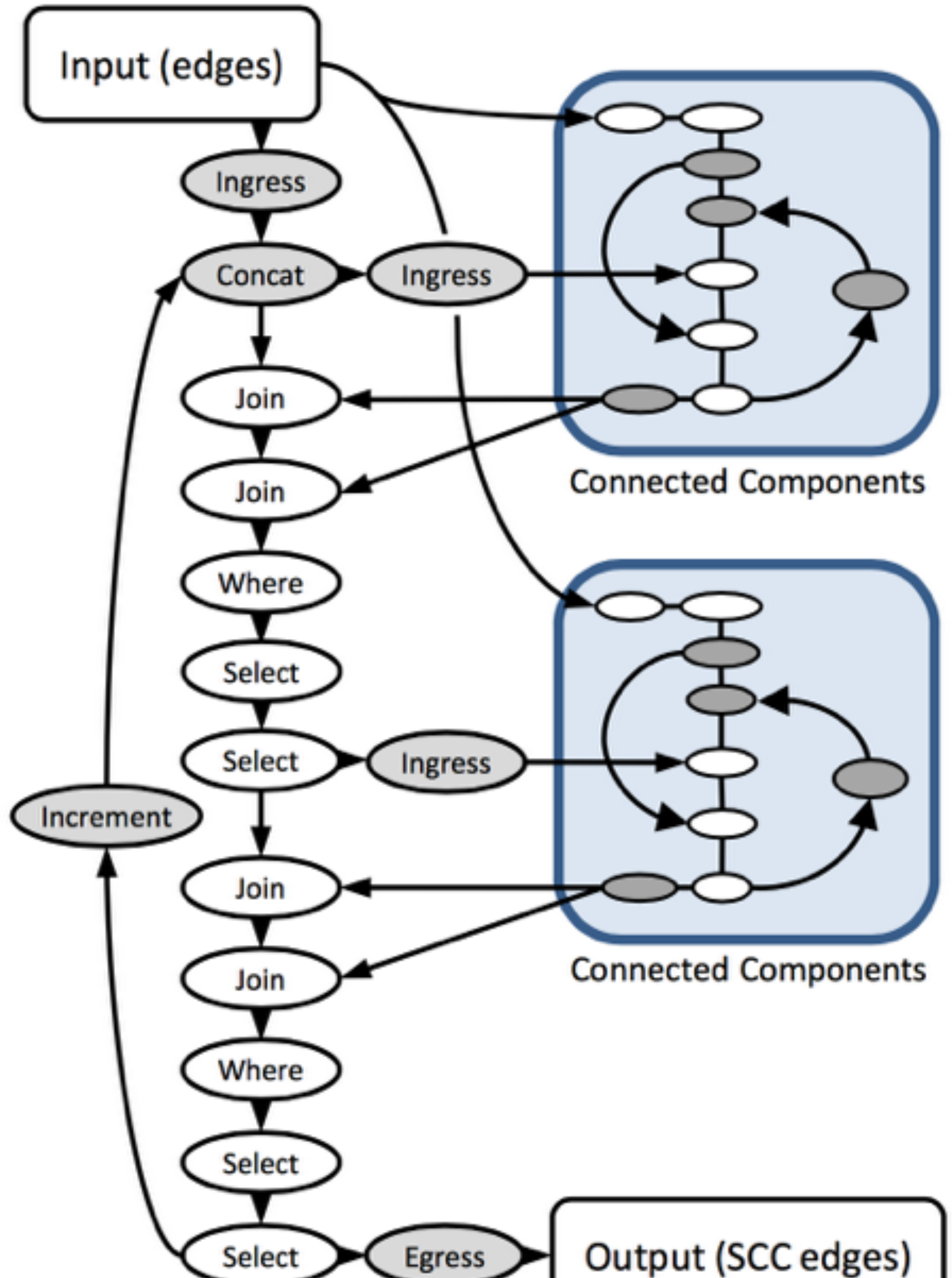


Good news everyone!

You can do this today with “Timely Dataflow”

Operators need time-generic implementations

```
RecvAt(time, data):   // you write
SendBy(time, data):   // you call
NotifyAt(time):       // you call
OnNotify(time):       // you write
```



Input (edges)
Ingress
Concat
Ingress
Join
Join
Where
Select
Select
Ingress
Increment
Join
Join
Where
Select
Select
Egress
Output (SCC edges)
Connected Components
Connected Components

Zero-Overhead Implementations

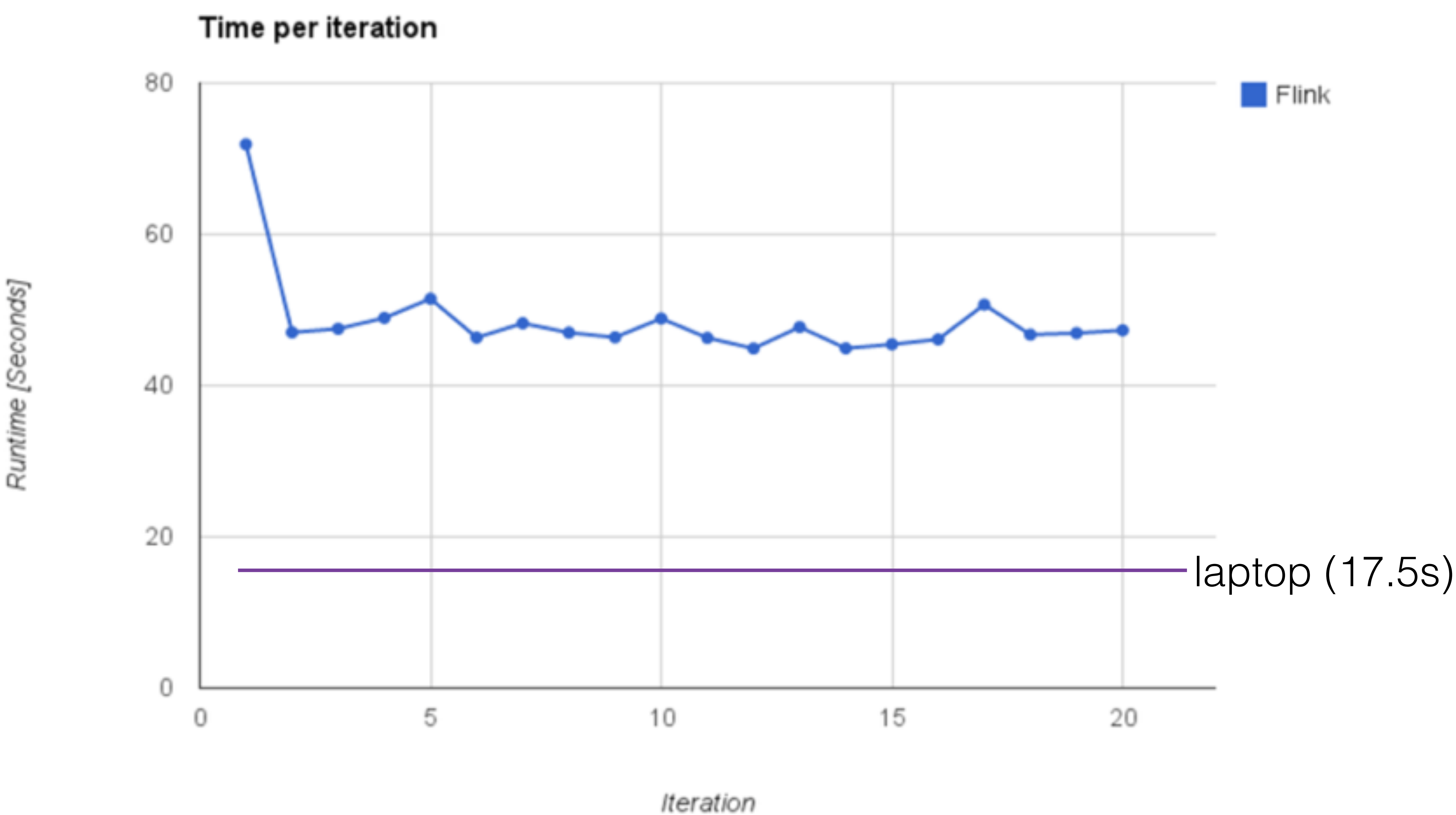
Sales pitch for Rust (www.rust-lang.org)

What is this code doing?

Scala | Java8 | Java7

```scala
val iteration = initialRanks.iterate(numIterations) {
  pages => {
    val rankContributions = pages.join(adjacency).where("id").equalTo("id") {
      (page, adj, out : Collector[Page]) => {
        val rankPerTarget = DAMPENING_FACTOR*page.rank/adj.neighbors.length;

        // send random jump to self
        out.collect(Page(page.id, RANDOM_JUMP))

        // partial rank to each neighbor
        for (neighbor <- adj.neighbors) {
          out.collect(Page(neighbor, rankPerTarget));
        }
      }
    }

    rankContributions.groupBy("id").reduce( (a,b) => Page(a.id, a.rank + b.rank))
  }
}
```

What is this code doing?



What is this code doing?

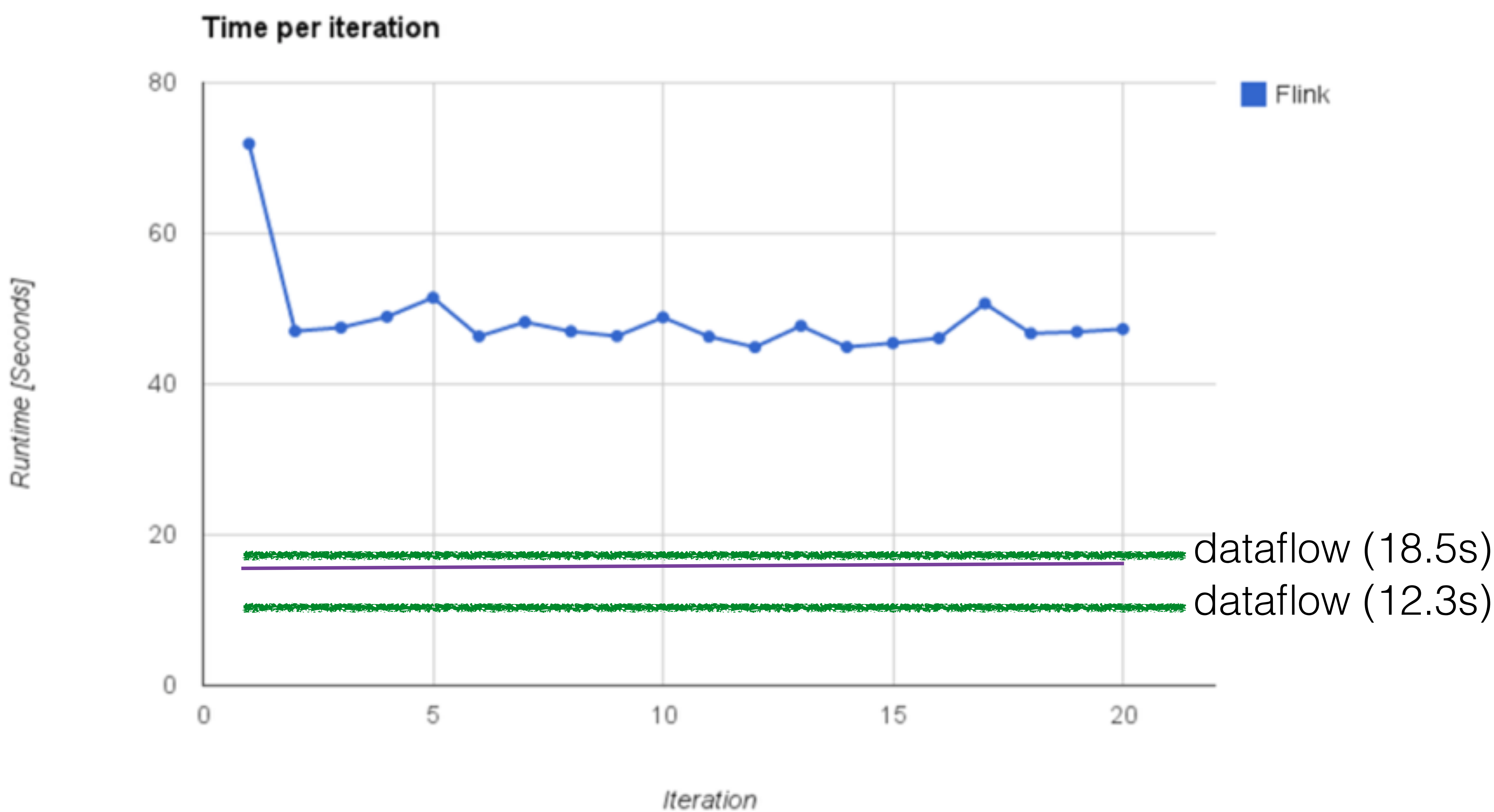
Scala Java8 Java7

```
val iteration = initialRanks.iterate(numIterations) {
  pages => {
    val rankContributions = pages.join(adjacency).where("id").equalTo("id") {
      (page, adj, out : Collector[Page]) => {
        val rankPerTarget = DAMPENING_FACTOR*page.rank/adj.neighbors.length;

        // send random jump to self
        out.collect(Page(page.id, RANDOM_JUMP))

        // partial rank to each neighbor
        for (neighbor <- adj.neighbors) {
          out.collect(Page(neighbor, rankPerTarget));
        }
      }
    }

    rankContributions.groupBy("id").reduce( (a,b) => Page(a.id, a.rank + b.rank))
  }
}
```

```rust
fn pagerank<G: Graph>(graph: &G, iterations: usize) {

    let mut src = vec![1.0f32; graph.nodes()];
    let mut dst = vec![0.0f32; graph.nodes()];

    for iteration in 0..iterations {
        for node in 0..src.len() {
            src[node] = 0.15 + 0.85 * src[node];
        }
        for node in 0..src.len() {
            let edges = graph.edges(node);
            let value = src[node] / edges.len() as f32;
            for &edge in edges {
                dst[edge as usize] += value;
            }
        }
        for node in 0..src.len() {
            src[node] = dst[node];
            dst[node] = 0;
        }
    }
}
```

```
for node in 0..src.len() {
    src[node] = 0.15 + 0.85 * src[node];
}
for node in 0..src.len() {
    let edges = graph.edges(node);
    let value = src[node] / edges.len() as f32;
    for &edge in edges {
        dst[edge as usize] += value;
    }
}
```

What is this code doing?



Simple programming model

enriched to reflect program structure

Performant implementations

which provide zero-cost abstractions