Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

Department of Computational and Data Sciences

DS256:Jan17 (3:1)

L6:Distributed Graph Processing

Yogesh Simmhan

Assignment 2 (Storm)

- Deadline extension to Apr 8 (Sunday), 11:59PM
  - Submission instructions
- No class on Thu 29 Mar

Guest Lectures

- Robin Thomas, VMWare, Apr 5 4-5pm, CDS 102
  - Scalability in VMWare's IoT Architecture
- NetApp, Apr 10/12 (TBD), 4-5PM, CDS 102
  - Storage Area Networks and Big Data Platforms

Project

- Teams of 1 or 2
- Project proposal, teams by Apr 2, 11:59pm
  - 2 pages, IEEE 2 column format, PDF
  - Email to simmhan@iisc.ac.in with subject "DS256 Project Proposal:Lastname1, Lastname2"

1. Motivation
2. Related work
3. Contributions
4. Technical challenges addressed
5. Evaluation: Datasets, Scalability, Platforms, Cluster

- Proposal presentation, Apr 3
- Final project code, report due by Apr 24
- Demo, final presentation, Apr 28 Sat

Pregel: a system for large-scale graph processing

Malewicz, et al

SIGMOD 2010

Graphs are commonplace

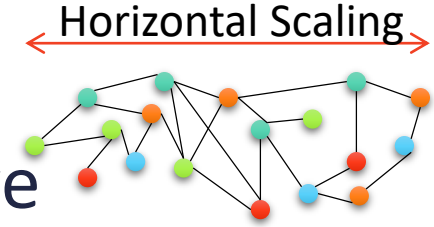
- Web & Social Networks
  - Web graph, Citation Networks, Twitter, Facebook, Internet
- Knowledge networks & relationships
  - Google's Knowledge Graph, NELL
- Cybersecurity
  - Telecom call logs, financial transactions, Malware
- Internet of Things
  - Transport, Power, Water networks
- Bioinformatics
  - Gene sequencing, Gene expression networks

Graph Algorithms

- Traversals: *Paths & flows between different parts of the graph*
  - Breadth First Search, Shortest path, Minimum Spanning Tree, Eulerian paths, MaxCut
- Clustering: *Closeness between sets of vertices*
  - Community detection & evolution, Connected components, K-means clustering, Max Independent Set
- Centrality: *Relative importance of vertices*
  - PageRank, Betweenness Centrality

When Had**p is just not good enough…

- Gap in frameworks for *dynamic graphs*
- Tuple/row/column-oriented frameworks
  - *E.g. Hadoop, Hive, Impala*
- Graph Databases & In-Memory
  - *E.g. Flock DB, Neo4J, MSR Trinity, Giraph*
  - Focus on large simple graphs, complex "queries", distributed in-memory
- Parallel Graph Frameworks
  - MPI, parallel computing, steep curve
  - Specialized HPC/shared-memory hardware

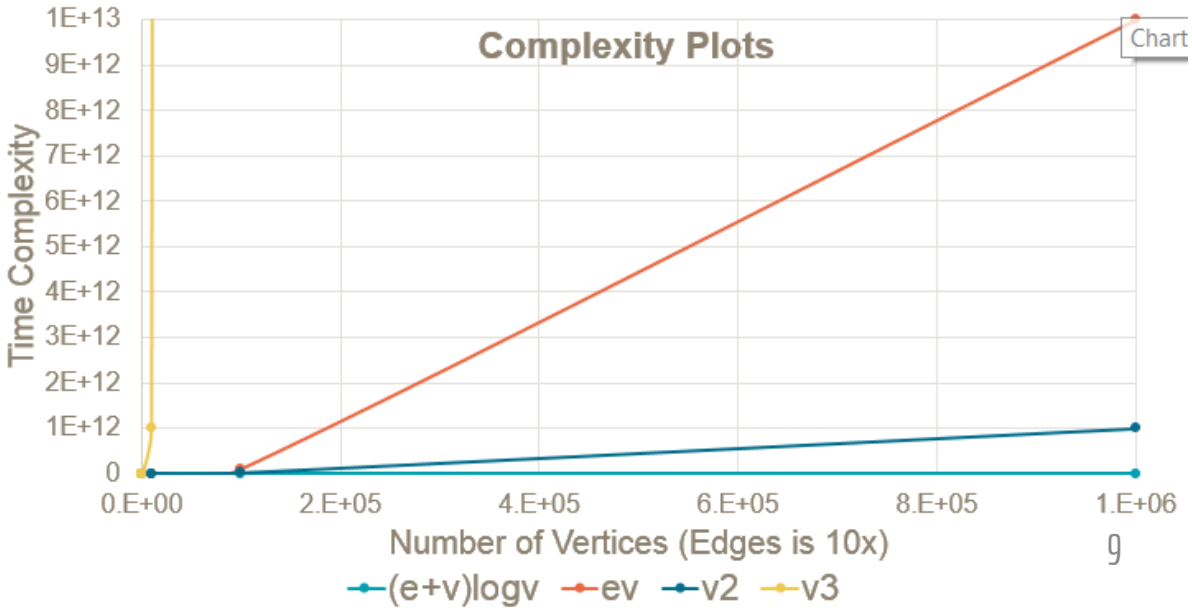


✓Storage & compute are (loosely)coupled

But, Graphs can be challenging

- Computationally complex algorithms
  - Shortest Path: O((E+V) log V) ~ O(EV)
  - Centrality: O(EV) ~ O(V^3)
  - Clustering: O(V) ~ O(V^3)
- *And these are for "shared-memory" algorithms*

Graph500.org's fastest supercomputer, ***K computer*** *with 524,288 cores performed at* ***17E+12 TEPS***



But, Graphs can be challenging

- Graphs sizes can be huge
  - Google's index contains 50B pages
  - Facebook has around 1.1B users
  - Twitter has around 530M users
  - Google+ has around 570M users

Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

But, Graphs can be challenging

- Shared memory algorithms don't scale!
- Do not fit naturally to *Hadoop/MapReduce*
  - Multiple MR jobs (iterative MR)
  - Topology & Data written to HDFS each time
  - *Tuple*, rather than graph-centric, abstraction
- Lot of work on *parallel graph libraries* for HPC
  - Boost Graph Library, Graph500
  - Storage & compute are (loosely) coupled, not fault tolerant
  - But everyone does not have a supercomputer ☺
- Processing and *querying* are different
  - Graph DBs not suited for analytics
  - Focus on large simple graphs, complex "queries"
  - *E.g. Neo4J, FlockDB, 4Store, Titan*

PageRank using MapReduce

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Lin, Fig 5.8

PageRank using MapReduce

- MR run over multiple iterations (typically 30)
  - *The graph structure itself must be passed from iteration to iteration!*
- Mapper will
  - Initially, load adjacency list and initialize default PR
    - `<v1, <v2>+>`
  - Subsequent iterations will load adjacency list and new PR
    - `<v1, <v2>+, pr1>`
  - Emit two types of messages from Map
    - PR messages and Graph Structure Messages
- Reduce will
  - Reconstruct the adjacency list for each vertex
  - Update the PageRank values for the vertex based on neighbour's PR messages
  - Write adjacency list and new PR values to HDFS, to be used by next Map iteration
    - `<v1, <v2>+, pr1'>`

Google's Pregel

- Google, to overcome, these challenges came up with Pregel.
  - Provides scalability
  - Fault-tolerance
  - Flexibility to express arbitrary algorithms

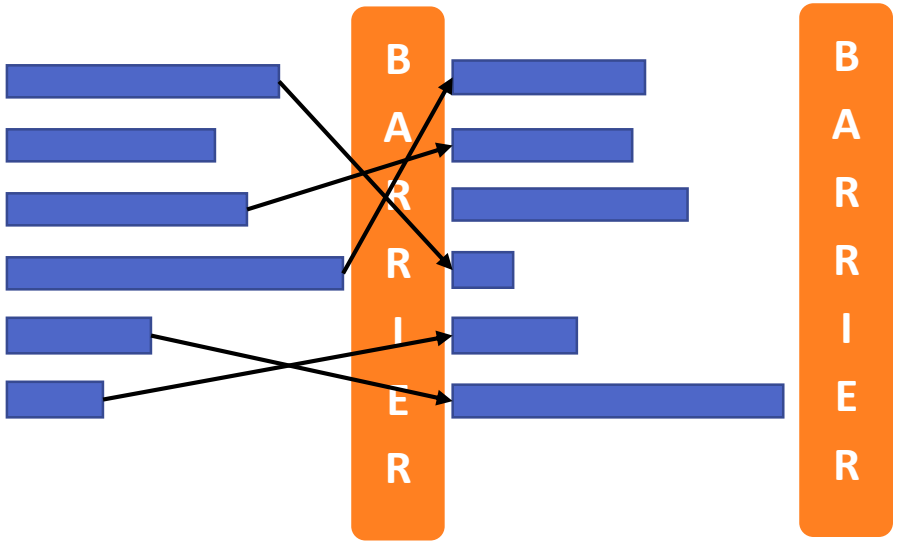
- The high level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel (BSP) model [1].

Slides courtesy "Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, SIGMOD 2010"

[1] Leslie G. Valiant, A Bridging Model for Parallel Computation. Comm. ACM 33(8), 1990

Bulk Synchronous Parallel (BSP)

- Distributed execution model
  - Compute → Communicate → Compute → Communicate → …
  - Bulk messaging avoids comm. costs



Vertex-centric BSP

- Series of iterations (*supersteps*) .
- Each vertex *V* invokes a function in parallel.
- Can read messages sent in previous superstep (*S-1*).
- Can send messages, to be read at the next superstep (*S+1*).
- Can modify state of outgoing edges.

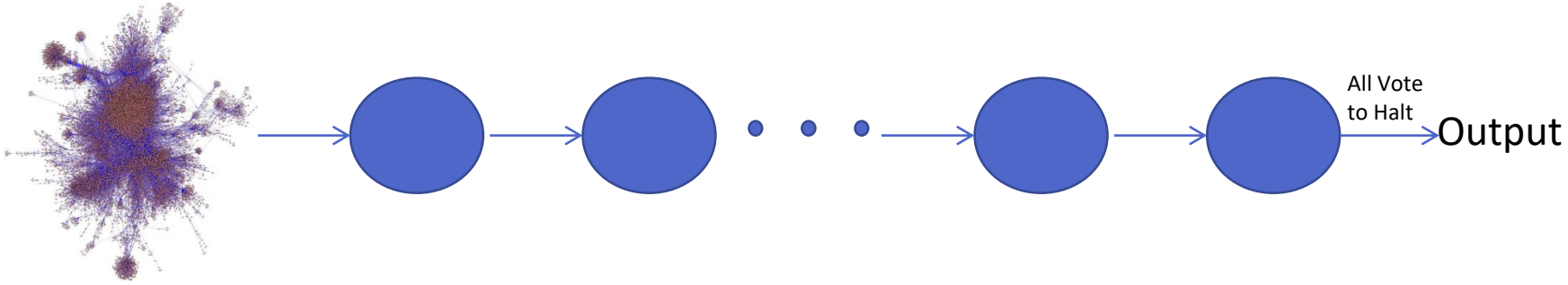
Advantage?

- In Vertex-Centric Approach
- Users focus on a *local action*
  - Think of **Map** method over tuple
- Processing each item *independently*.
- Ensures that Pregel programs are inherently free of *deadlocks* and *data races* common in asynchronous systems.

Apache Giraph
Implements *Pregel* Abstraction

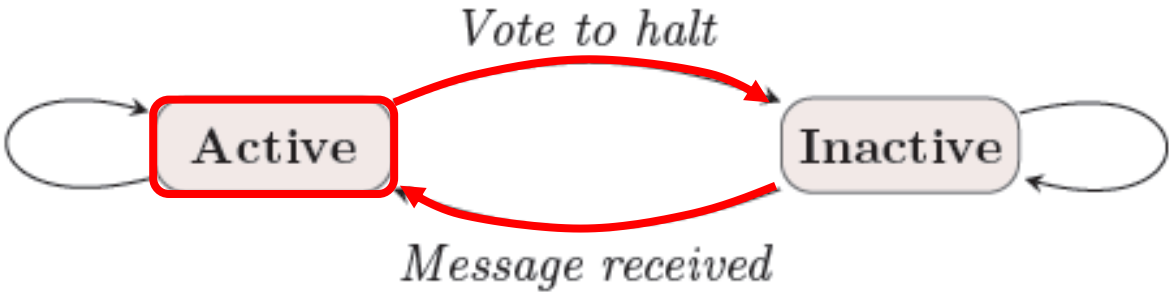
- Google's Pregel, SIGMOD 2010
  - Vertex-centric Model
  - Iterative BSP computation
- Apache Giraph donated by Yahoo
  - Feb 6, 2012: Giraph 0.1-incubation
  - May 6, 2013: Giraph 1.0.0
  - Nov 19, 2014: Giraph 1.1.0
- Built on Hadoop Ecosystem

Model of Computation



- A Directed Graph is given to Pregel.
- It runs the computation at each vertex.
- Until all nodes vote for halt.
- Pregel gives you a directed graph back.

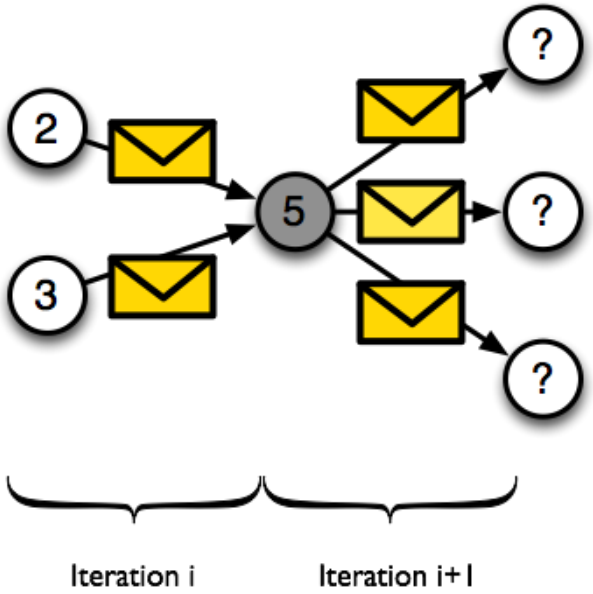
Vertex State Machine



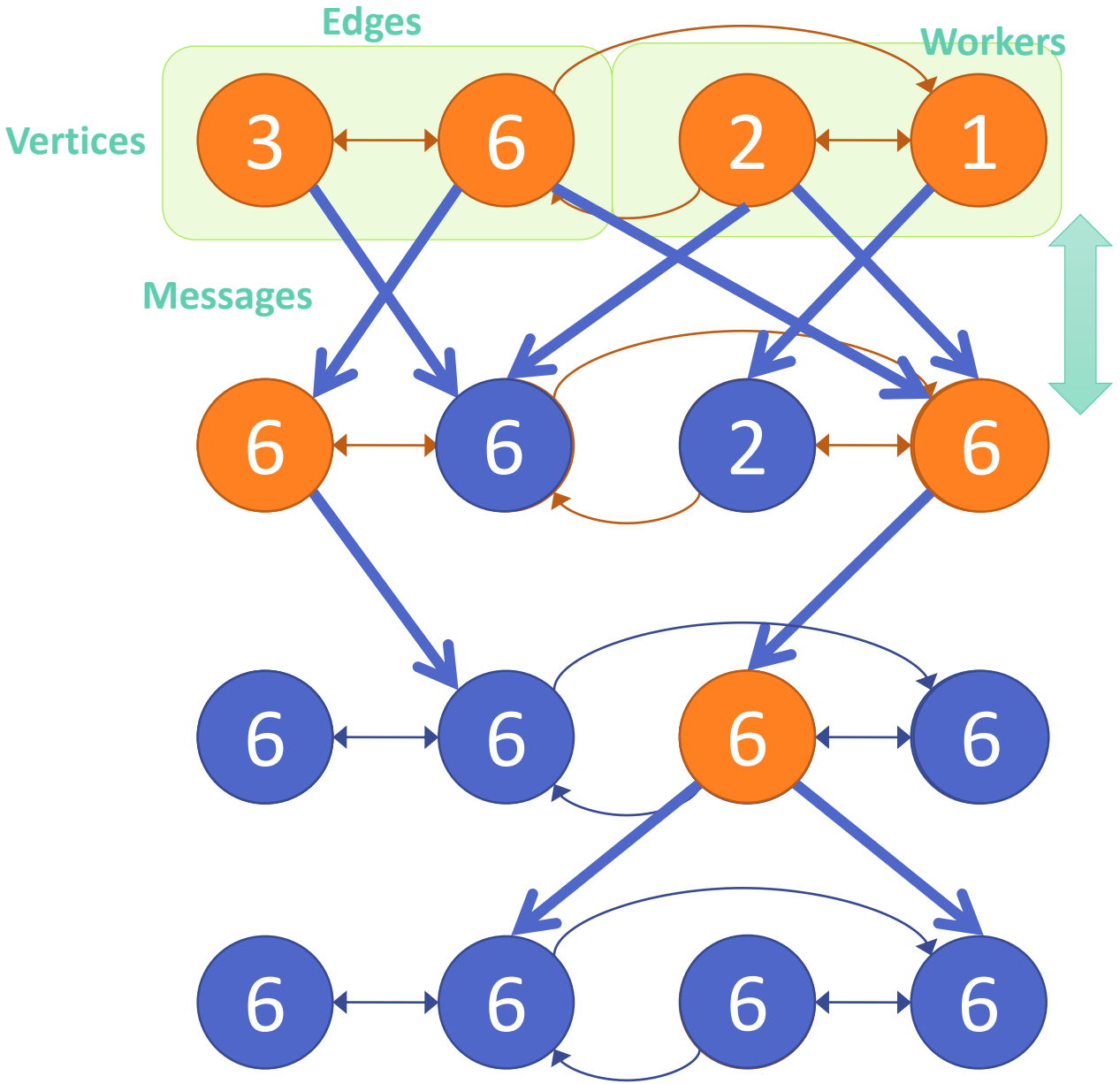
- Algorithm termination is based on every vertex *voting to halt*.
- In superstep 0, every vertex is in the *active* state.
- A vertex deactivates itself by voting to halt.
- It can be reactivated by receiving an (external) message.

Vertex Centric Programming

- Vertex Centric Programming Model
  - Logic written from perspective on a single vertex. Executed on all vertices.
- Vertices know about
  - Their own value(s)
  - Their outgoing edges



Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014



Blue Arrows are messages.

Blue vertices have voted to halt.

Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, SIGMOD 2010

Max Vertex

Algorithm 1 Max Vertex Value using Vertex Centric Model

```
1:  procedure COMPUTE(Vertex myVertex, Iterator⟨Message⟩ M)
2:      hasChanged = (superstep == 1) ? true : false
3:      while M.hasNext do            ▶ Update to max message value
4:          Message m ← M.next
5:          if m.value > myVertex.value then
6:              myVertex.value ← m.value
7:              hasChanged = true
8:      if hasChanged then            ▶ Send message to neighbors
9:          SENDTOALLNEIGHBORS(myVertex.value)
10:     else
11:         VOTETOHALT( )
```

GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics, Simmhan, et al, EuroPar 2014

Advantages

- Makes distributed programming easy
  - No locks, semaphores, race conditions
  - Separates computing from communication phase
- Vertex-level parallelization
  - Bulk message passing for efficiency
- Stateful (in-memory)
  - Only messages & checkpoints hit disk

Apache Giraph: API

```
void compute(Iterator<IntWritable> msgs)
    getSuperstep()
    getVertexValue()
    edges = iterator()
    sendMsg(sinkVtx, value)
    sendMsgToAllEdges(value)
    voteToHalt()
```

Message passing

- No guaranteed message delivery order.
- Messages are delivered exactly once.
- Can send messages to any node.
  - Though, typically to neighbors

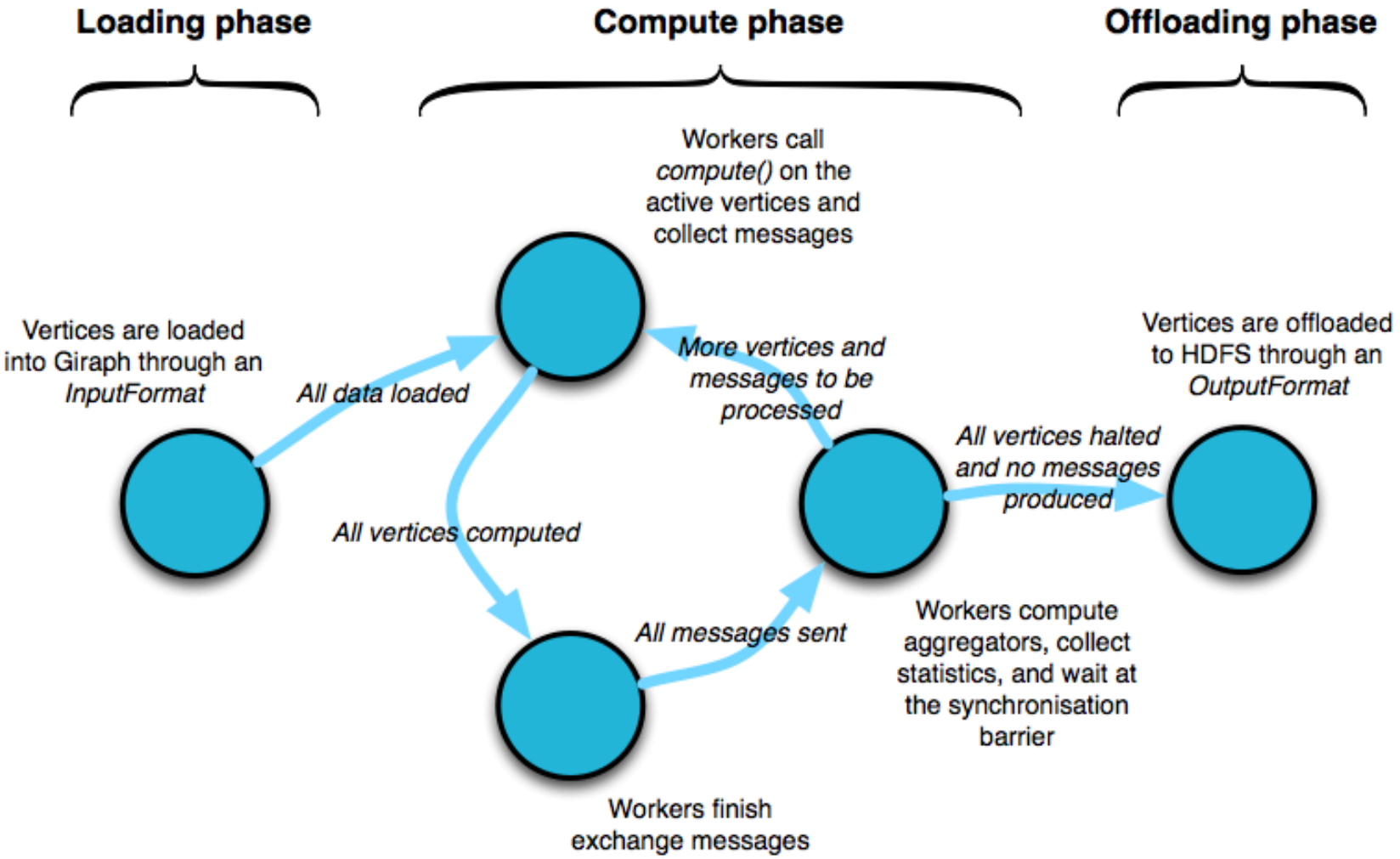
```
public class MaxVertexVertex extends IntIntNullIntVertex {
    public void compute(Iterator<IntWritable> messages)
            throws IOException {
        int currentMax = getVertexValue().get();
        // first superstep is special,
        // because we can simply look at the neighbors
        if (getSuperstep() == 0) {
            Iterator<IntWritable> edges = iterator();
            while(edges.hasNext()) {
                int neighbor = edges.next().get();
                if (neighbor > currentMax) {
                   currentMax = neighbor;
                }
            } ...
```

Based on org.apache.giraph.examples.ConnectedComponentsVertex

```
    ...
    // only need to send value if it is not the own id
    if (currentMax != getVertexValue().get()) {
      setVertexValue(new IntWritable(currentMax));
    Iterator<IntWritable> edges = iterator();
    while(edges.hasNext()) {
      int neighbor = edges.next().get();
      if (neighbor < currentMax) {
        sendMsg(new IntWritable(neighbor),
          getVertexValue());
      }
    }
  }
  voteToHalt();
  return;
} // end getSuperstep==0
```

```
    boolean changed = false; // getSuperstep != 0
    // did we get a smaller id?
    while (messages.hasNext()) {
      int candidateMax = messages.next().get();
      if (candidateMax > currentMax) {
            currentMax = candidateMax;
            changed = true;
        }
    }
    // propagate new component id to the neighbors
    if (changed) {
        setVertexValue(new IntWritable(currentMax));
        sendMsgToAllEdges(getVertexValue());
    }
    voteToHalt();
} // end compute()
```

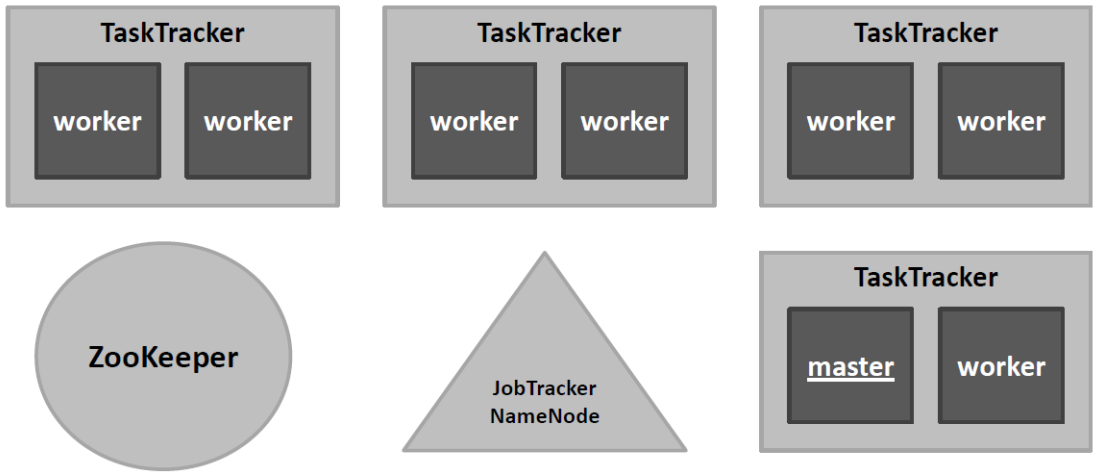
Apache Giraph



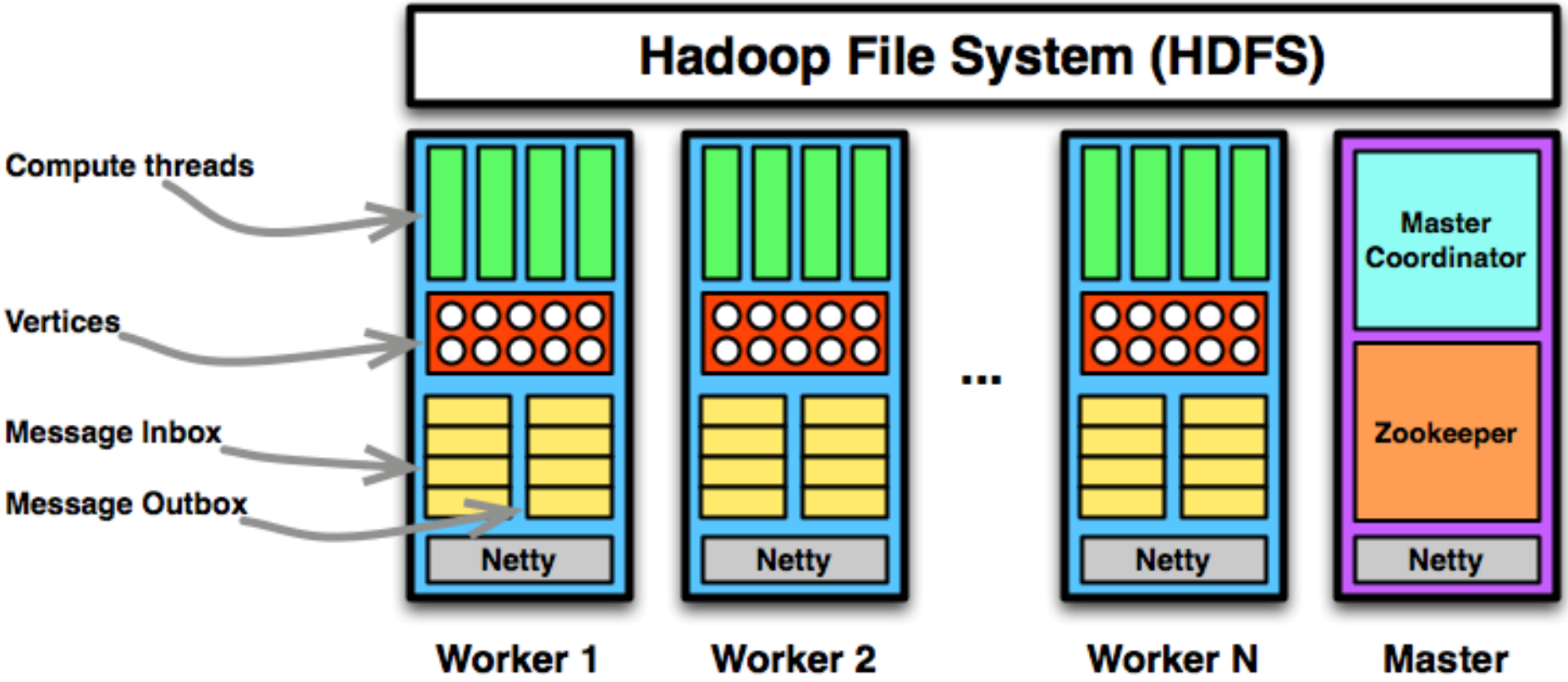
Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

Giraph Architecture

- Hadoop Map-only Application
- **ZooKeeper**: responsible for **computation state**
  - Partition/worker mapping, global #superstep
- **Master**: responsible for **coordination**
  - Assigns partitions to workers, synchronization
- **Worker:** responsible for **vertices**
  - Invokes active vertices compute() function, sends, receives and assigns messages



Giraph Architecture



- Checkpointing of supersteps possible

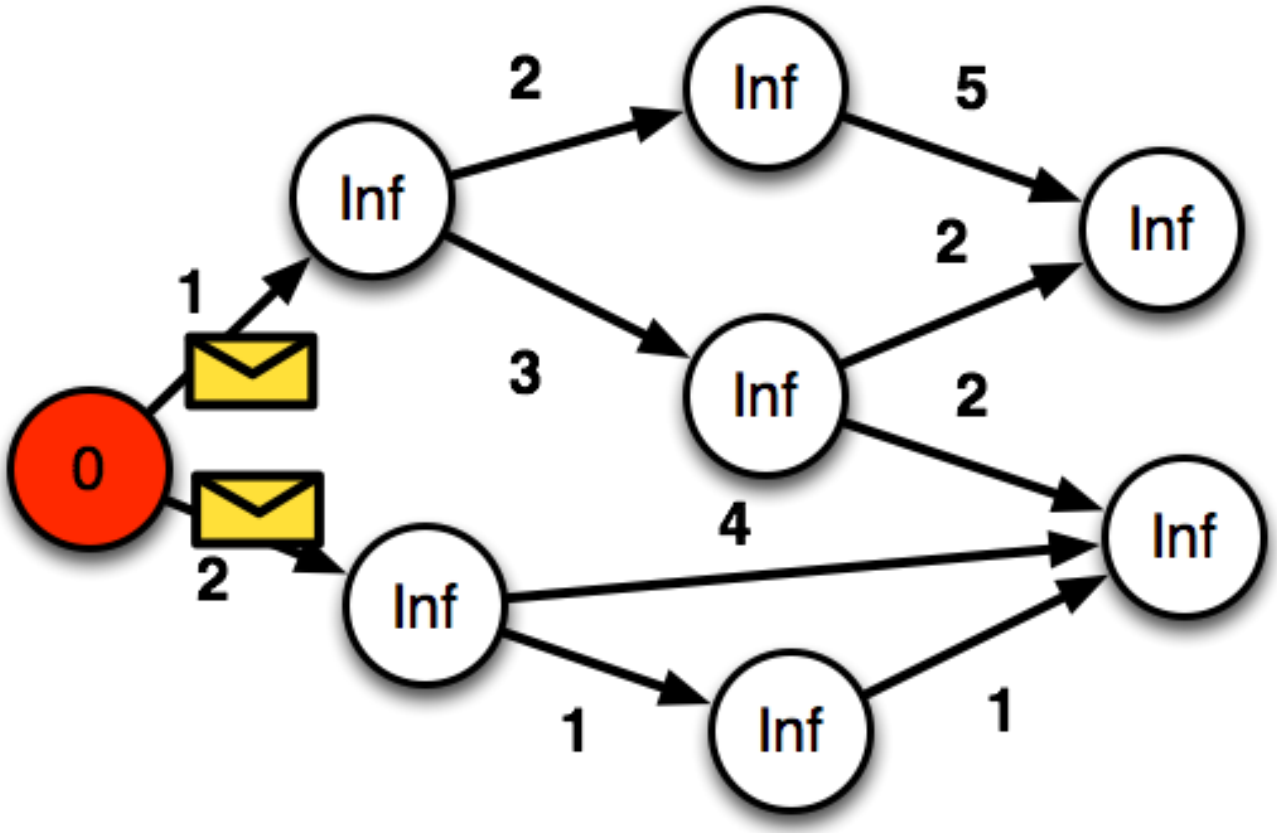
Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

Shortest Path

```
class ShortestPathVertex
        : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
        mindist = IsSource(vertex_id()) ? 0 : INF;
        while ( (m = msgs->Next()) != NULL)
            mindist = min(mindist, m->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator edges = GetOutEdgeIterator();
            while( (e = edges.Next()) != NULL)
                SendMessageTo(e.Target(),
                    mindist + e.GetValue());
        }
        VoteToHalt();
  }
};
```

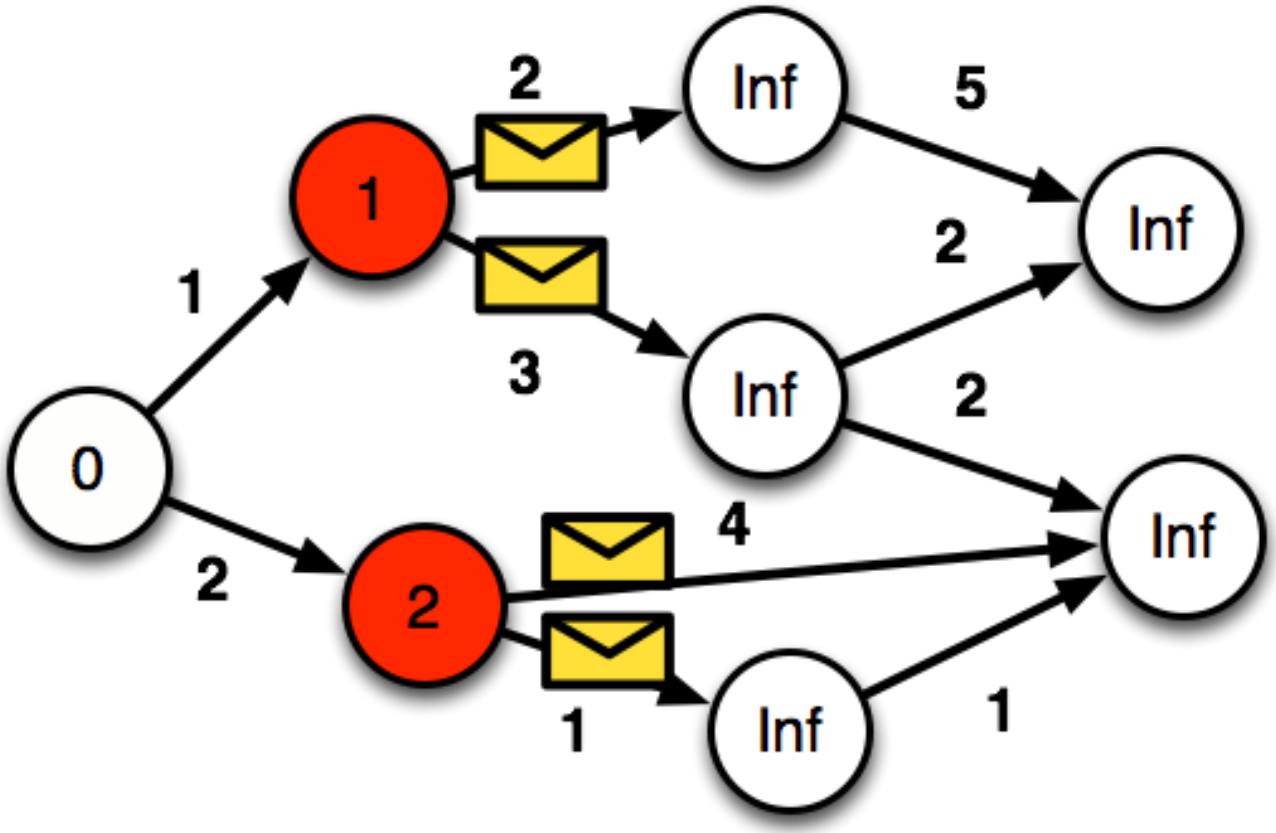
In the 1st superstep, only the source vertex will update its value (from INF to zero)

Shortest Path

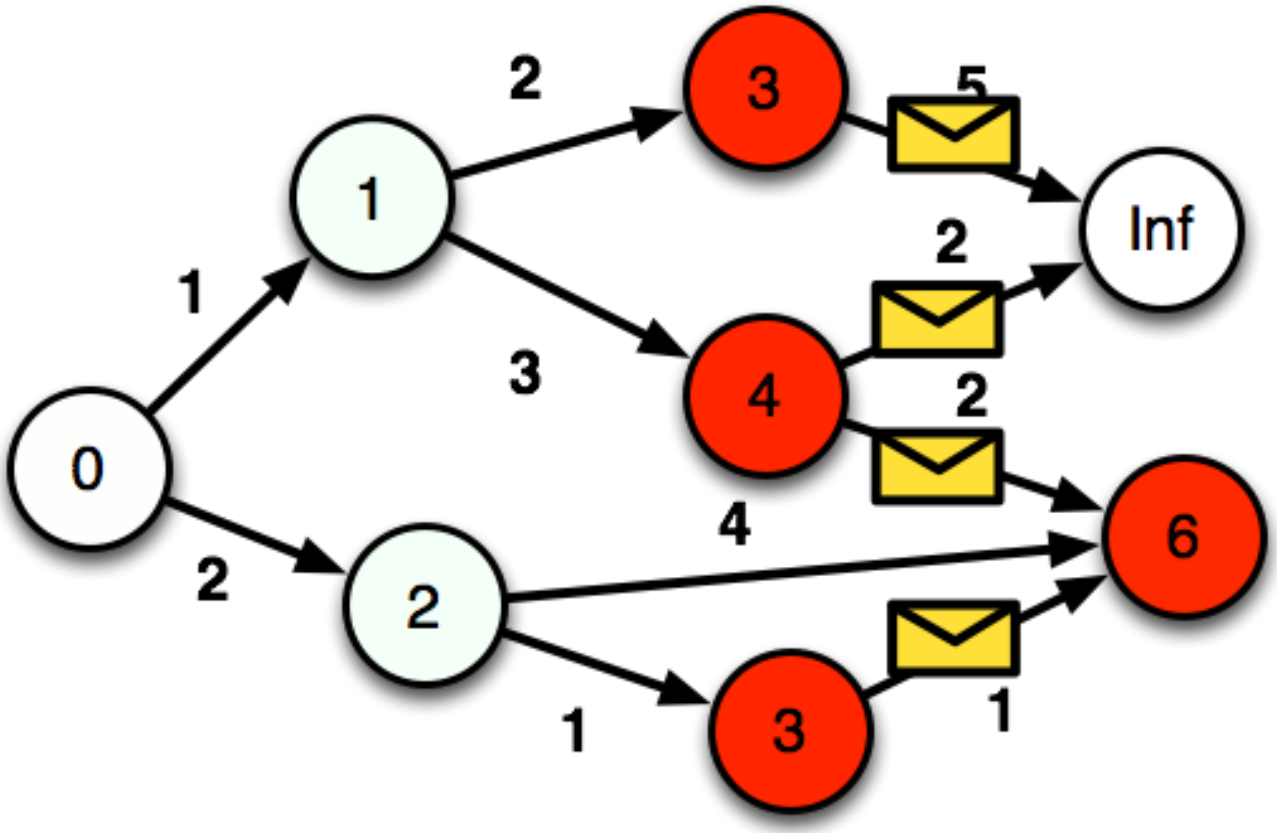


Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

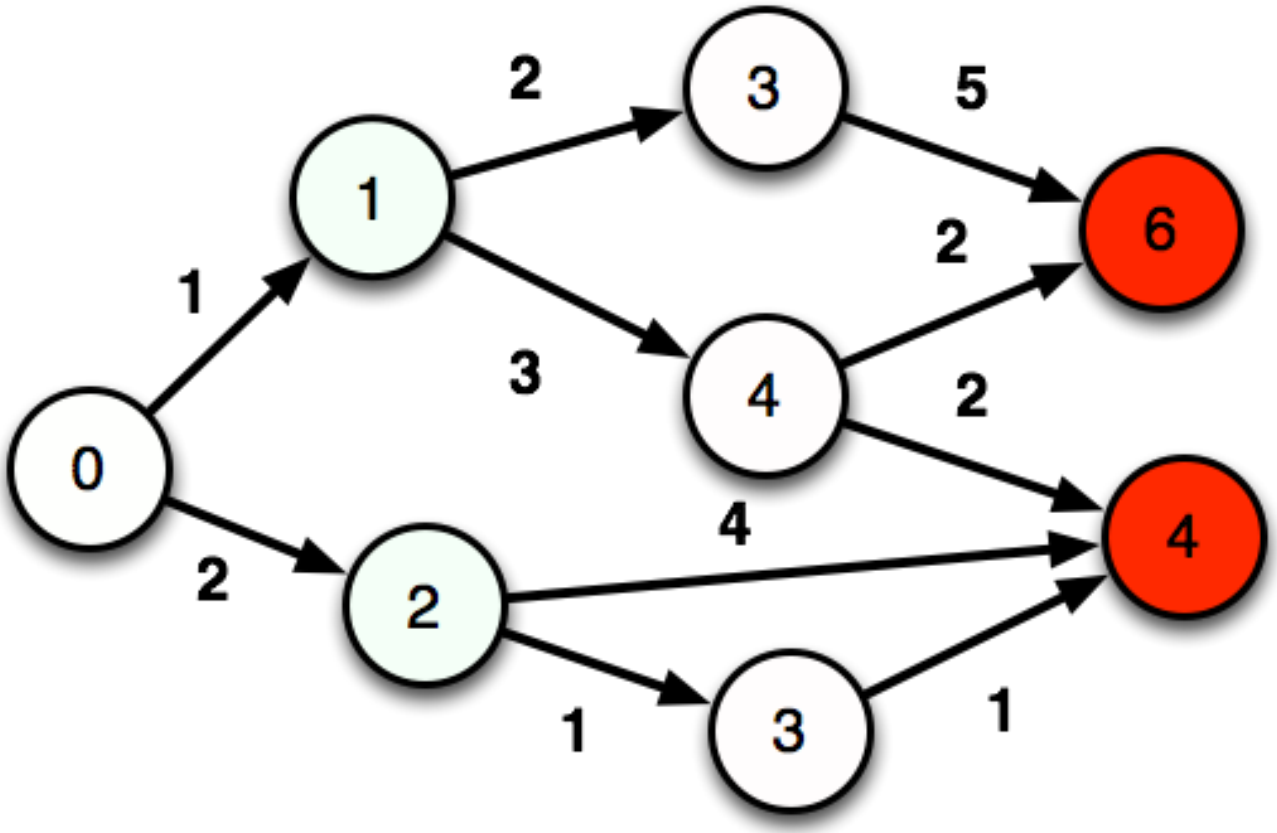
Shortest Path



Shortest Path



Shortest Path



PageRank, recursively

$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

- P(n) is PageRank for webpage/URL ‘n’
  - Probability that you’re in vertex ‘n’
- |G| is number of URLs (vertices) in graph
- α is probability of random jump
- L(n) is set of vertices that link to ‘n’
- C(m) is out-degree of ‘m’

PageRank using MapReduce

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Lin, Fig 5.8

Application – Page Rank

Store and carry PageRank

```
class PageRankVertex
        : public Vertex<double, void, double> {
public:
  virtual void Compute(MessageIterator* msgs) {
        if (superstep() == 0) *MutableValue() = 1 / NumVertices();
        else
        if (superstep() >= 1) {
                double sum = 0;
                while((m = msgs->Next()) != NULL)
                        sum += m->Value();
                *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue() / n);
        } else
                VoteToHalt();
  }
};
```

Combiners

- Sending a message to remote vertex has overhead
  - Can we merge multiple *incoming* message into one?
- User specifies a way to reduce many messages into one value (ala Reduce in MR)
  - by overriding the **Combine()** method.
  - Must be commutative and associative.

```
originalMessage =
 combine(vid, originalMessage, messageToCombine)
```

- Exceedingly useful in certain contexts (e.g., 4x speedup on shortest-path computation).
  - e.g. for MAX, om = om < mtc ? mtc : om

MasterCompute

- Runs before slave compute()
- Has a global view
- A place for aggregator manipulation

- `MasterCompute`: Executed on master
- `WorkerContext`: Executed per worker
- `PartitionContext`: Executed per partition

Aggregators

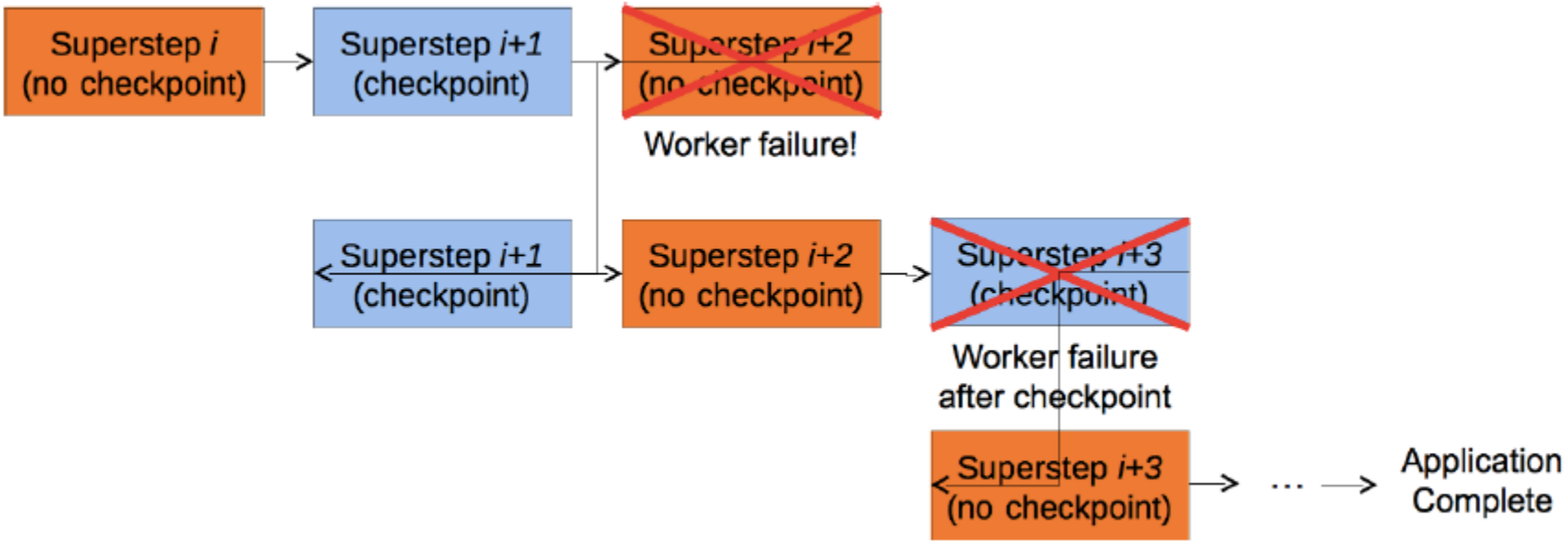
- A mechanism for global communication, monitoring, and data.
  - Each vertex can produce a value in a superstep S for the Aggregator to use.
  - The Aggregated value is available to all the vertices in superstep S+1.
- Implemented using Master Compute
- Aggregators can be used for statistics and for global communication.
  - E.g., **Sum** applied to out-edge count of each vertex.
    - generates the total number of edges in the graph and communicate it to all the vertices.

Partitioner

- Maps vertices to partitions that are operated by workers
  - Default is a hash partitioner
- Done once at the start of the application
- Called at the end of each superstep, for dynamic migration of partitions

Checkpointing

- Optionally capture the state of vertex, messages at periodic supersteps, e.g. 2
- Globally revert to last checkpoint superstep on failure



Topology mutations

- Some graph algorithms need to change the graph's topology.
  - E.g. A clustering algorithm may need to replace a cluster with a node
- Vertices can create / destroy vertices at will.
- Resolving conflicting requests:
  - Partial ordering:

    E Remove,V Remove,V Add, E Add.
  - User-defined handlers:

    You fix the conflicts on your own.