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

DS256:Jan16 (3:1)

L12:Distributed Graph Processing

Yogesh Simmhan

21 Feb, 2017

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

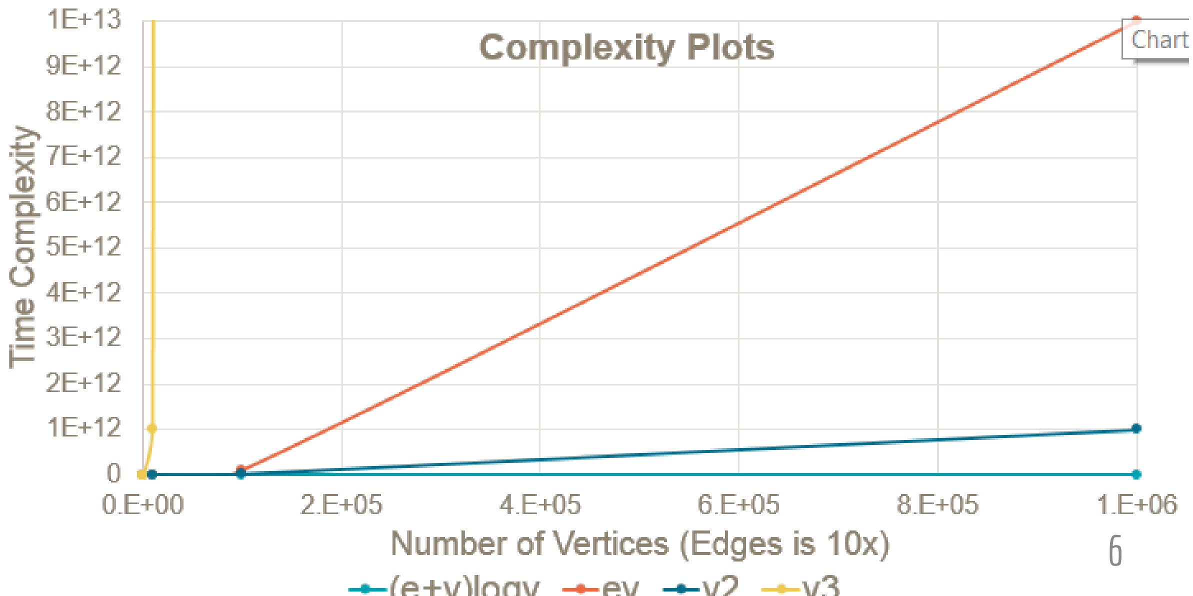
Graph Algorithms

- 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

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

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

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                                   ▷ Recover graph structure
            else
                s ← s + p                               ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        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

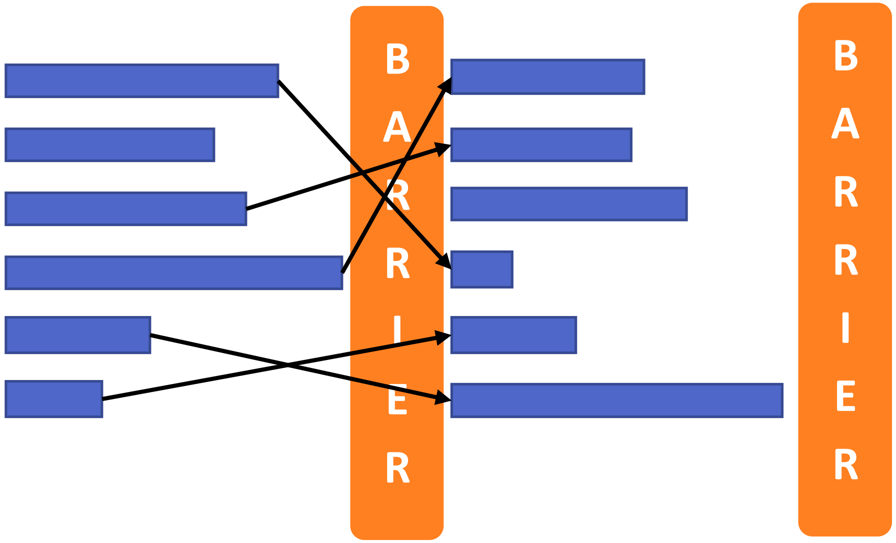
- The high level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel (BSP) model [1].

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

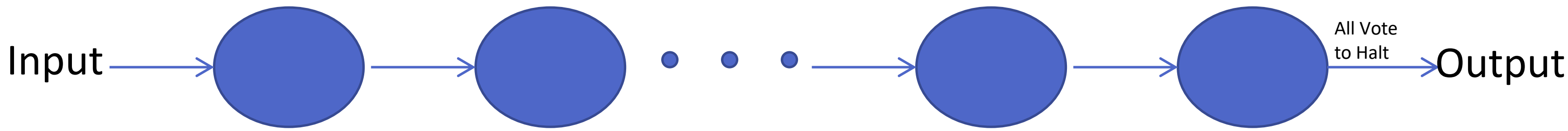
[1] Leslie G. Valiant, A Bridging Model for Parallel Computation. Comm. ACM 33(8), 1990

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.

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

Advantage?

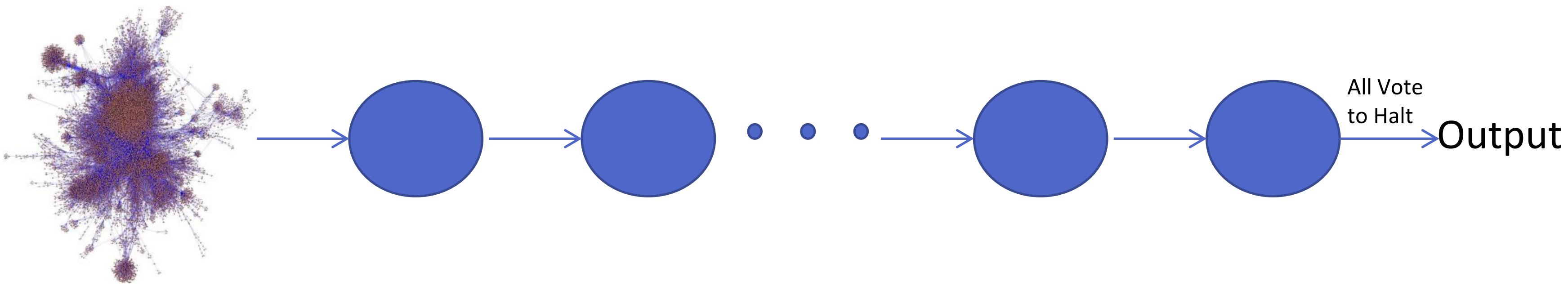
- In Vertex-Centric Approach
- Users focus on a *local action*
  - Think of **Map** method over tuple
- Processing each item *independently*.
- Ensures that Pregel programs are inherently free of *deadlocks* and *data races* common in asynchronous systems.

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

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

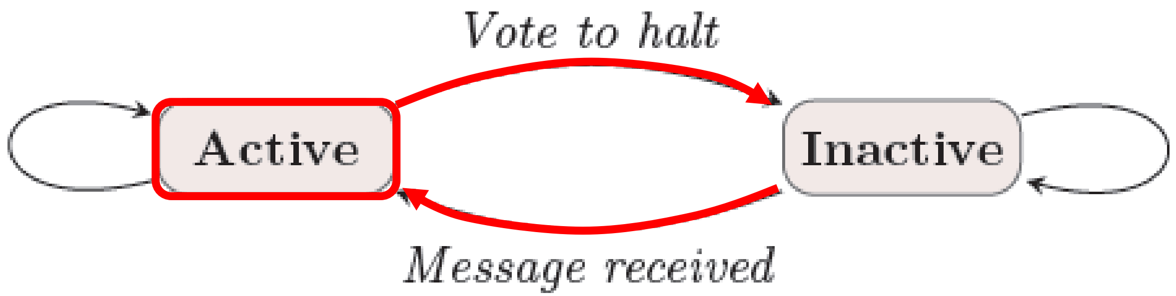
Model of Computation



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

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

Vertex State Machine

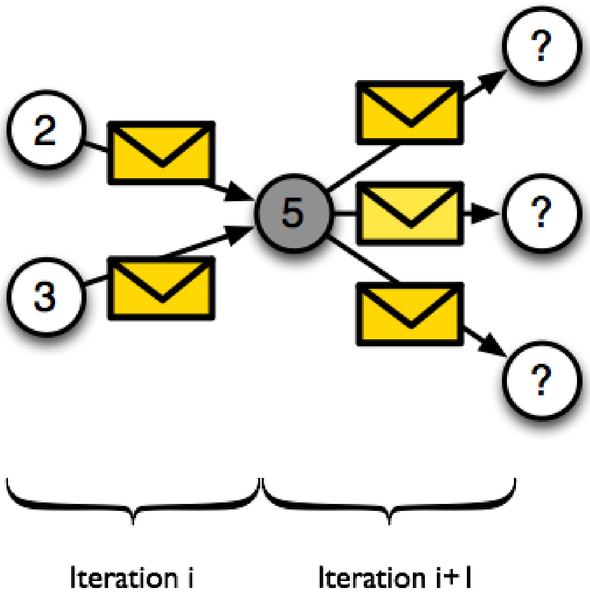


- Algorithm termination is based on every vertex *voting to halt*.
- In superstep 0, every vertex is in the *active* state.
- A vertex deactivates itself by voting to halt.
- It can be reactivated by receiving an (external) message.

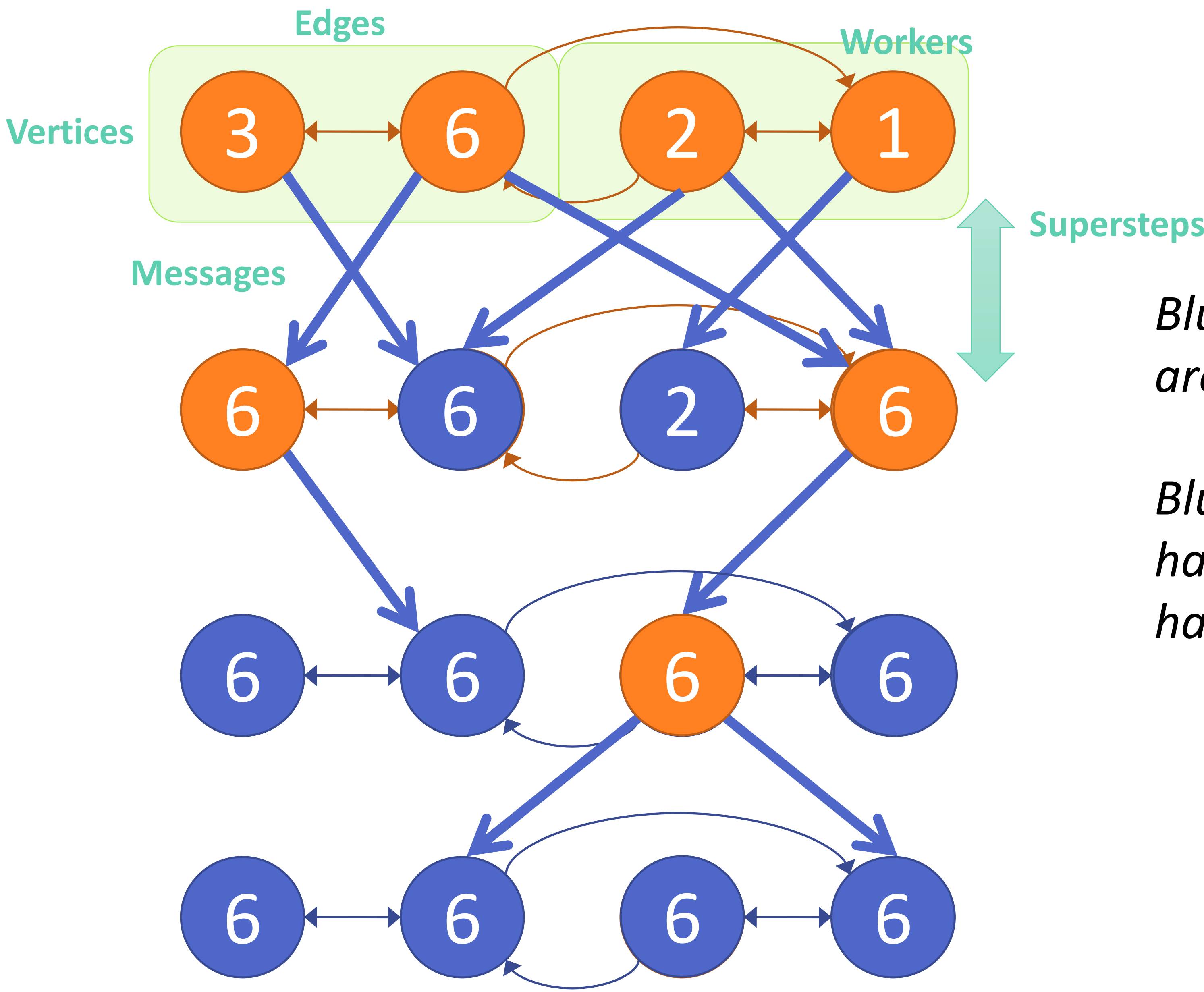
Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, SIGMOD 2010

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

```
procedure COMPUTE(Vertex myVertex, Iterator⟨Message⟩ M)
    hasChanged = (superstep == 1) ? true : false
    while M.hasNext do                        ▷ Update to max message value
        Message m ← M.next
        if m.value > myVertex.value then
            myVertex.value ← m.value
            hasChanged = true
    if hasChanged then                        ▷ Send message to neighbors
        SENDTOALLNEIGHBORS(myVertex.value)
    else
        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(edge, 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

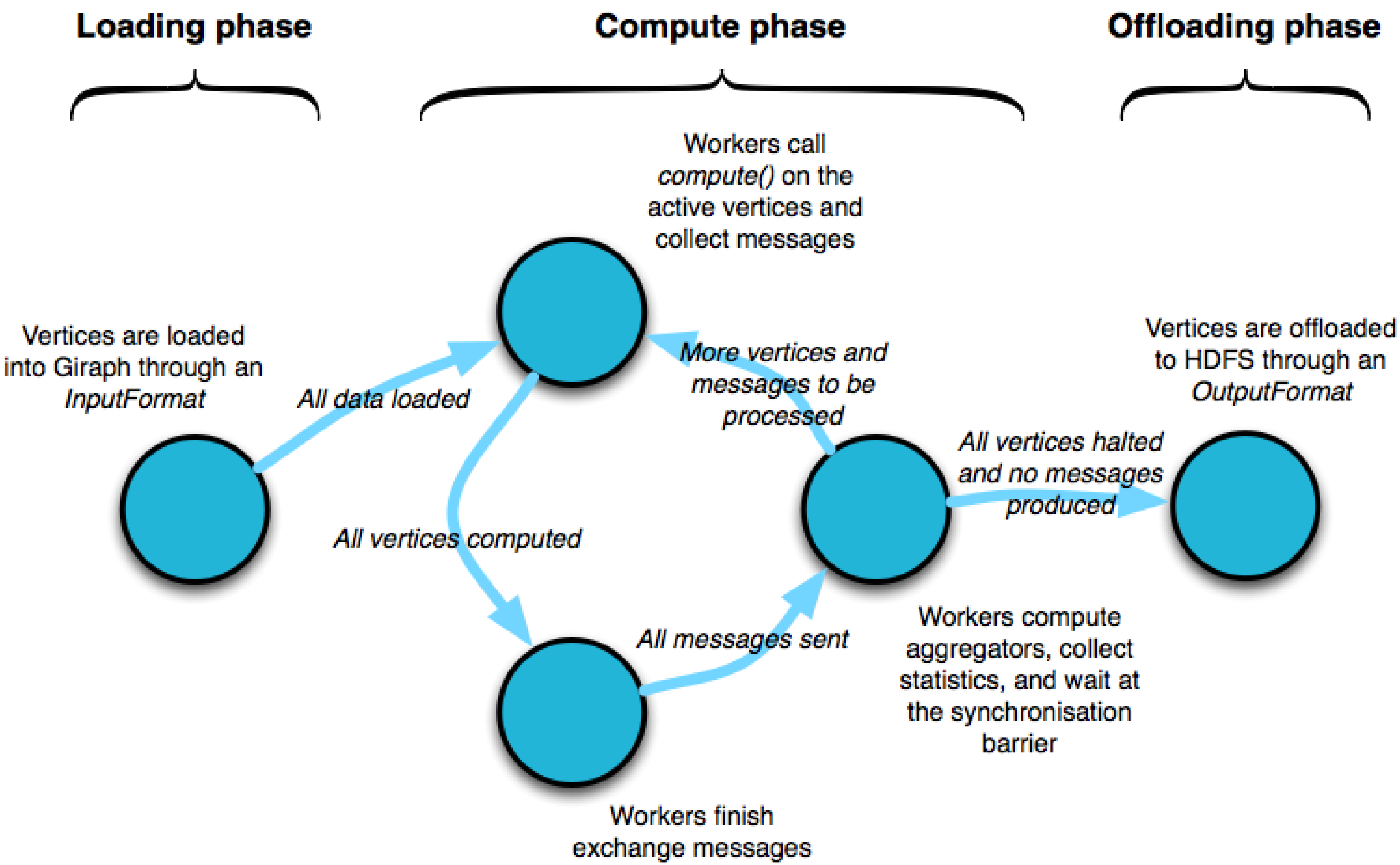
```
public class MaxVertexVertex extends IntIntNullIntVertex {
    public void compute(Iterator<IntWritable> messages)
            throws IOException {
        int currentMax = getVertexValue().get();
        // first superstep is special,
        // because we can simply look at the neighbors
     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

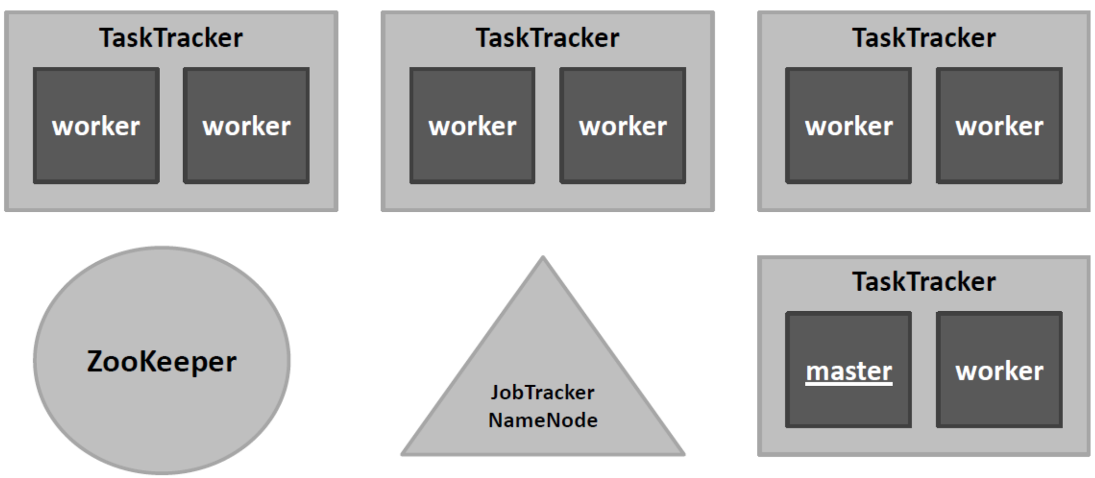
Apache Giraph



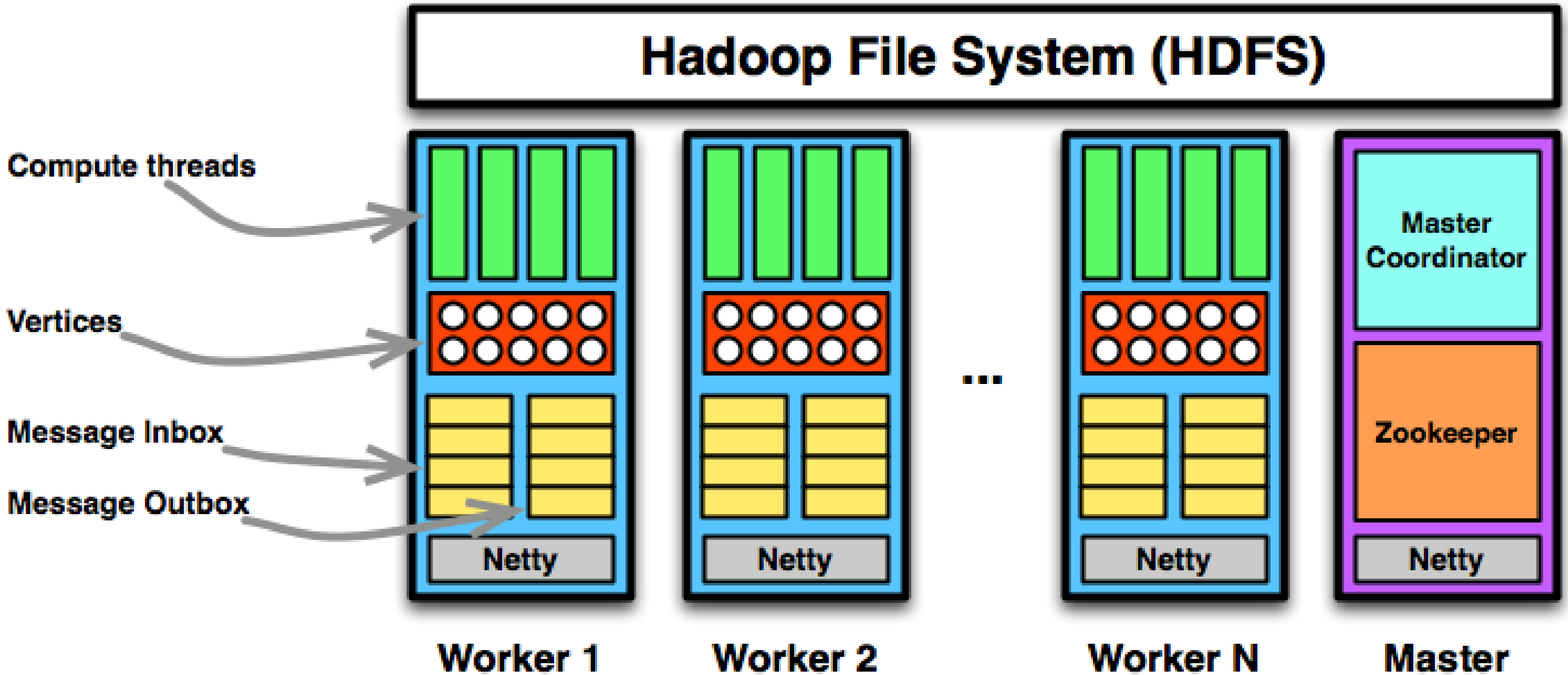
Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

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

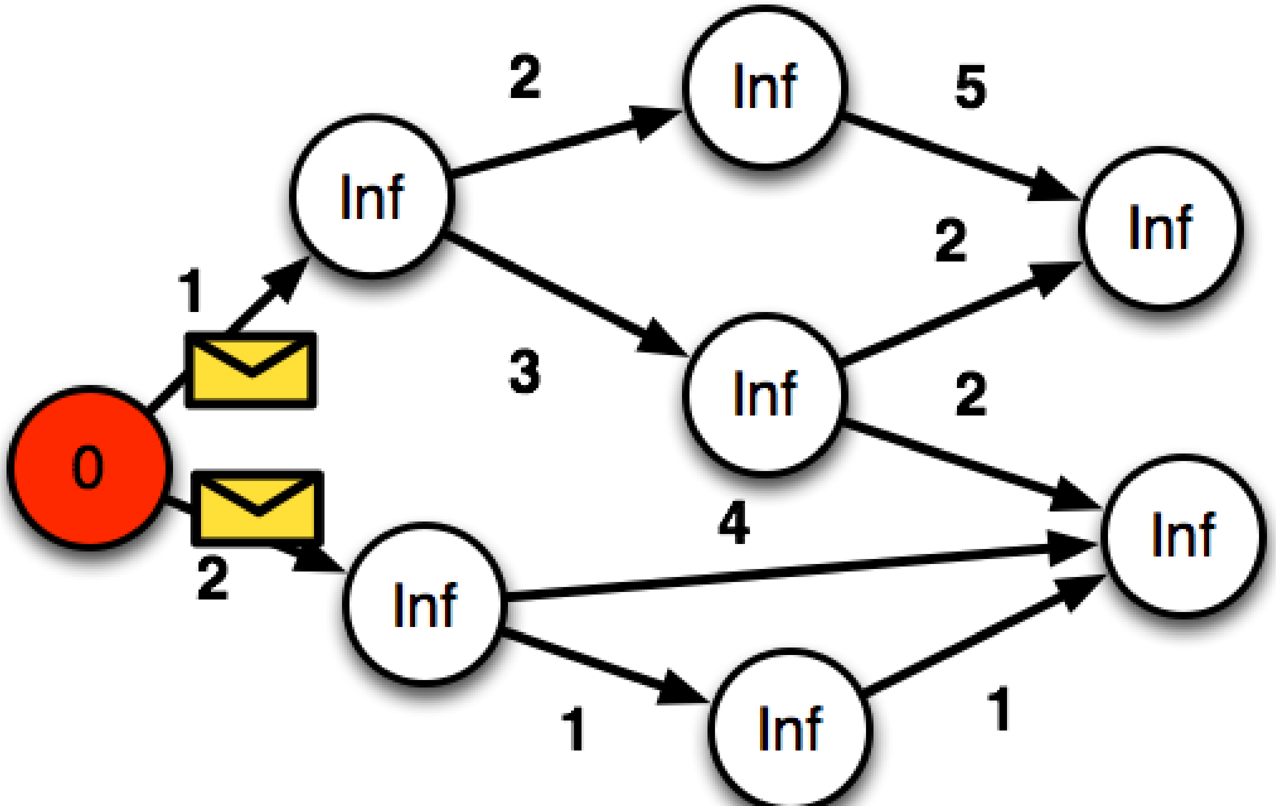
Shortest Path

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

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

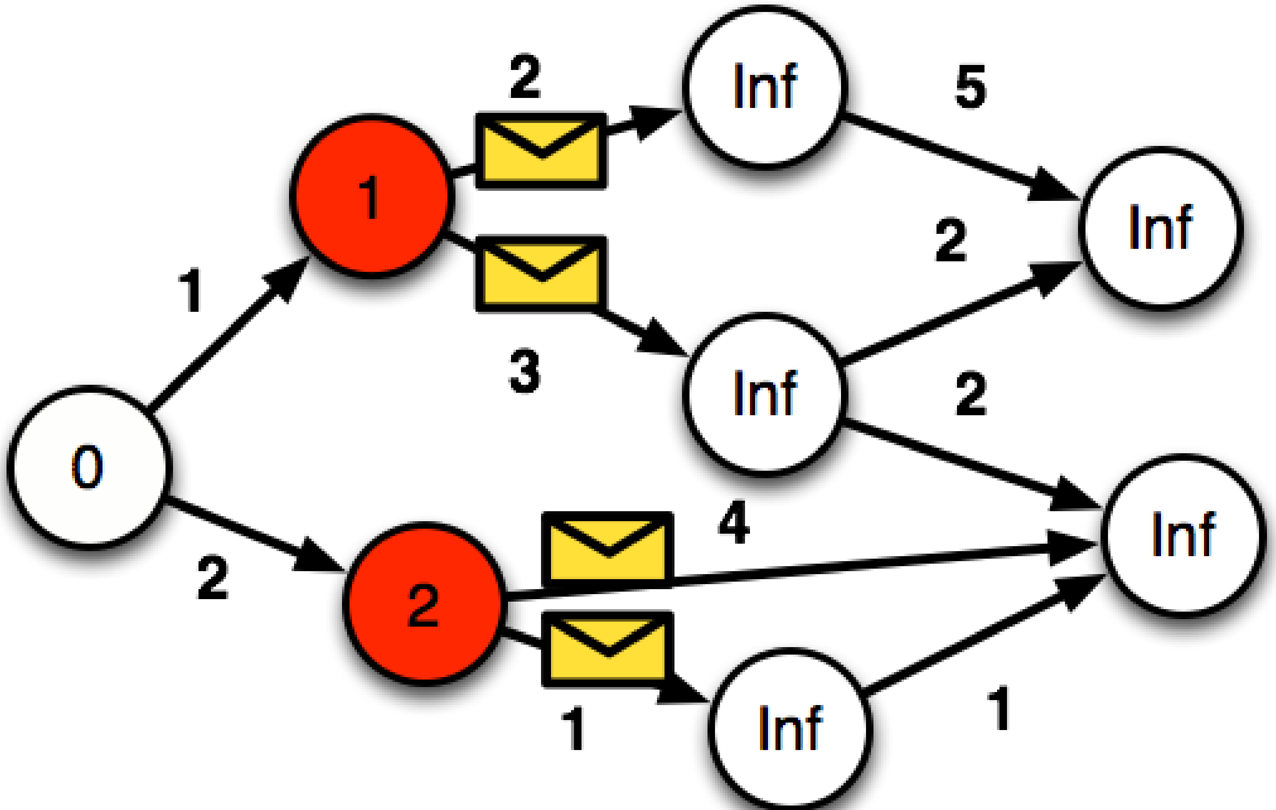

Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, SIGMOD 2010

Shortest Path

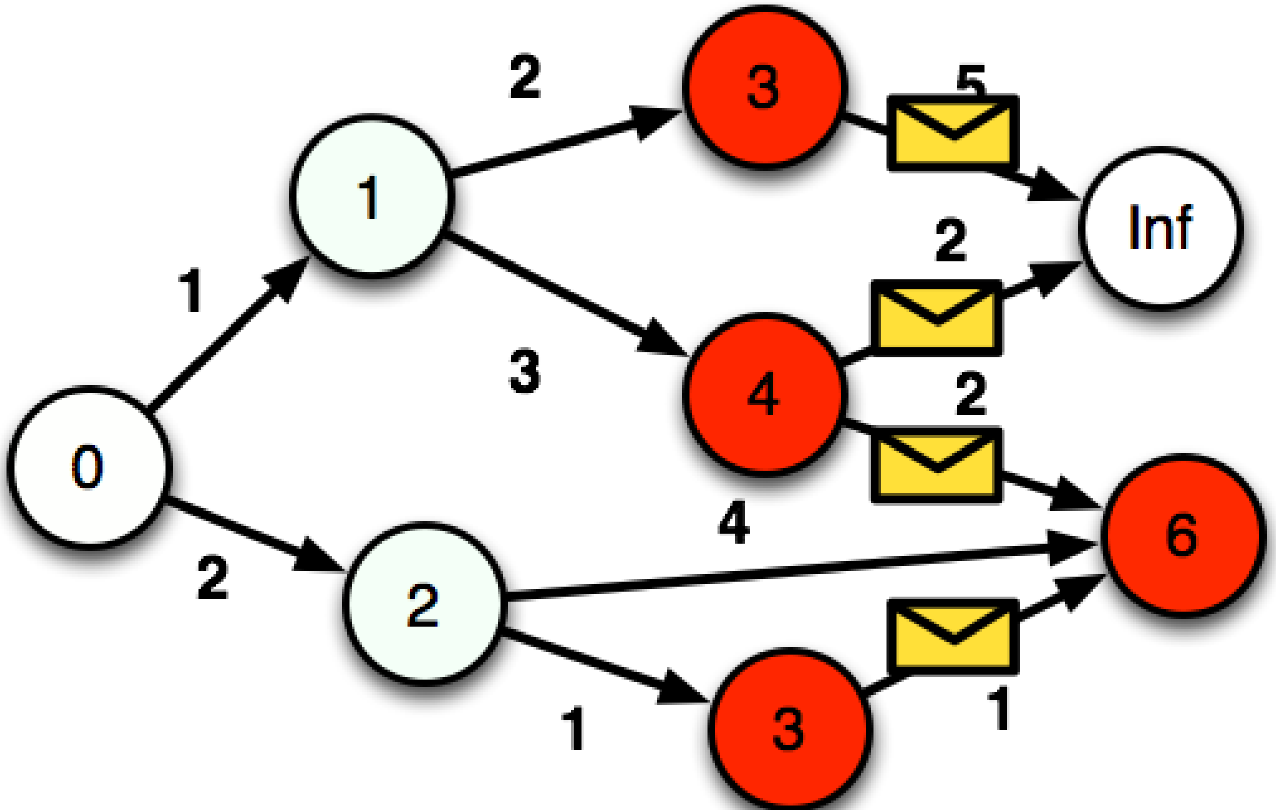


Apache Giraph, Claudio Martella, Hadoop Summit, Amsterdam, April 2014

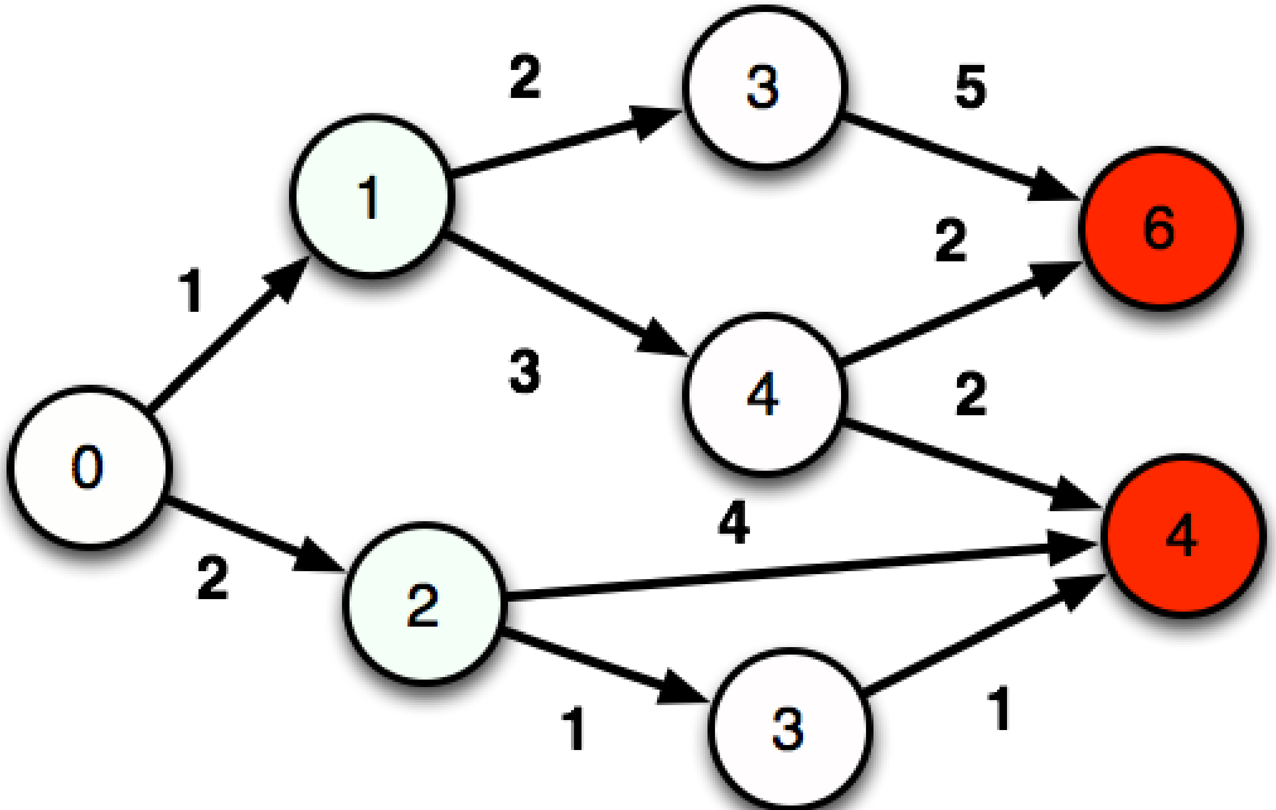
Shortest Path



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

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

class Reducer
    method Reduce(nid m, [p1, p2, . . .])
        M ← ∅
        for all p ∈ counts [p1, p2, . . .] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        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() >= 1) {
                double sum = 0;
                for (; !msgs->Done(); msgs->Next())
                        sum += msgs->Value();
                *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue() / n);
        } else
                VoteToHalt();
  }
};
```

Maximal Bipartite Matching

- Input is a bipartite graph with "left" and "right" vertices
- Find the ***maximal*** set of edges that do not share a common vertex
  - Randomized algorithm [1]… *"Each node is either matched or has no edge to an unmatched node"*
  - Maximal match does not give the *maximum match* ($O(n^2)$)
- Vertex value: left/right, paired vertex ID
- 4 phases, alternate between left and right vertices
- Repeat for fixed iterations or all possible vertices matched
  - Worst case O(n) for 'n' vertices on each side

Pregel: A System for Large-Scale Graph Processing, Grzegorz Malewicz et al, SIGMOD 2010
[1] Thomas Anderson, et al. High-Speed Switch Scheduling for Local-Area Networks. ACM Trans. Comp. Syst. 11(4), 1993

```
// Bipartite Matching
void compute(Message[] m){
    if(superstep%4 == 0 && v.side==L)
        if(v.other == -1)
            sendToNeighbors(v.id);
            VoteToHalt;
    else if(superstep%4 == 1 && v.side==R && v.other == -1)
        sentToVertex(m[0].id, true);
        foreach(i in m[1..size-1])
            sentToVertex(m[i].id, false);
        VoteToHalt;
    else if(superstep%4 == 2 && v.side==L)
        v.other = m.findFirst(msg => msg.value == true).id
        sentToVertex(v.other, true);
    else if(superstep%4 == 3 && v.side==R)
         v.other = m[0].id
        VoteToHalt;
}
```

Semi-Clustering

- Divide the graph into different parts to meet a goal
  - connectivity within the entities in each part
  - discrimination between entities in different parts
  - balancing of entities across parts
- Cluster into C_{max} semi-clusters each with at most V_{max} vertices, given by user
- *Vertices can be part of more than one semi-cluster*
- Semi-cluster Score:
  - I_c : sum of internal edge weights
  - B_c : Sum of boundary edge weights
  - V_c : number of vertices
  - f_b : coefficient (0.0-1.0)

$$S_c = \frac{I_c - f_B B_c}{\boxed{V_c(V_c - 1)/2}},$$

Normalization based on max edges in clique

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

```
// Semi-Clustering
void compute(Message[] m){
   if(superstep == 0)
      // create singleton clusters. Share with neighbors.
      v.map.put(cid,{v.id})
      sendToNeighbors(v.map)
   else if(superstep < MAX)
      // Update local clust. Merge & send top clusters to neighbors
      foreach(clust in m[])
         if(!clust.val.contains(v.id) && clust.size()<Vmax)
            tmpmap.put(clust.id, clust.val)
            tmpmap.put(clust.id', clust.val U v.id)
            changed = true
         else v.map.put(cid, clust,val)
      // sort by cluster score, prune to top Cmax, send to neighbors
      tmpmap.scoreAndSort().trim(Cmax)
      v.map.scoreAndSort().trim(Cmax)
      sendToNeighbors(tmpmap)
      voteToHalt()
   else  // Max iterations done
      voteToHalt()
}
```

Pregel: A System for Large-Scale Graph Processing, Grzegorz Malewicz et al, SIGMOD 2010
https://github.com/grafos-ml/okapi/blob/master/src/main/java/ml/grafos/okapi/graphs/SemiClustering.java

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`

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

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.

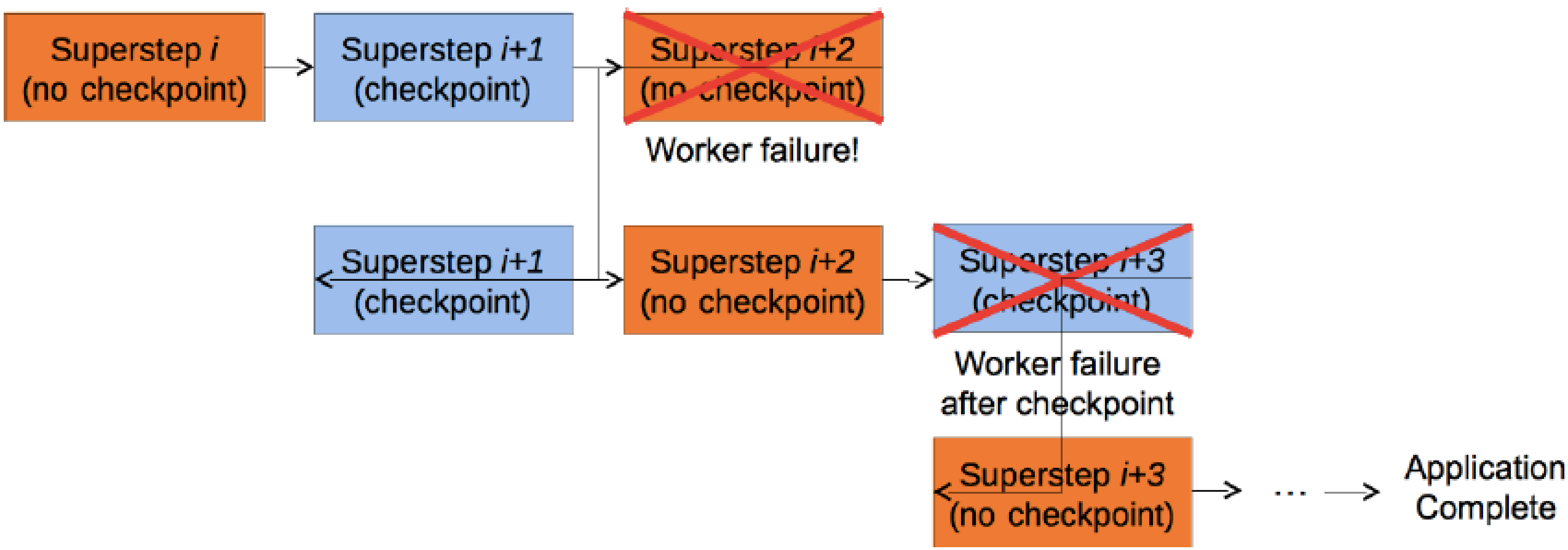
Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, SIGMOD 2010

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.

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

More Algorithms

K-Means Clustering

```
Input: undirected G(V, E), k, τ
 int numEdgesCrossing = INF;
 while (numEdgesCrossing > τ)
   int [] clusterCenters = pickKRandomClusterCenters(G)
   assignEachVertexToClosestClusterCenter(G, clusterCenters)
   numEdgesCrossing = countNumEdgesCrossingClusters(G)
```

Figure 3: A simple k-means like graph clustering algorithm.

```
public class EdgeCountingVertex extends
        Vertex<IntWritable, IntWritable> {
  @Override
  public void compute(Iterable<IntWritable> messages,
                              int superstepNo){
    if (superstepNo == 1) {
      sendMessages(getNeighborIds(), getValue().value());
    } else if (superstepNo == 2) {
      for (IntWritable message : messages) {
        if (message.value() != getValue().value()) {
          minValue = message.value();
          updateGlobalObject("num-edges-crossing-clusters",
            new IntWritable(1));}}
      voteToHalt(); }}}
```

Figure 4: Counting the number of edges crossing clusters with *vertex.compute()*.

- Multiple phases
  - k centers
  - assign vertex to cluster
  - find edge cuts
- Use multi-source BFS or Euclidian distance to find nearest cluster
- Use MasterCompute
  - k initial vertices
  - Calc edge-cut count
  - Decide termination

GPS: A Graph Processing System, Semih Salihoglu et al, SSDBM 2013
One Trillion Edges: Graph Processing at FacebookScale, Avery Ching, et al, VLDB 2015

K-Core

- k-core is a graph where each node has degree >=k
- Use graph mutations to iteratively delete vertices with degree < k
  - Pass *edge deletion* messages to all neighbors

```
repeat
    begin Superstep n
        for Messages received
            Delete corresponding out-edge
        end for
        if New degree < k then
            Delete node and out-edges
            Send message to neighbours
        end if
        Vote to halt
    end
until All nodes inactive
```

Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis, Louise Quick, et al, ASONAM 2012

Strongly Connected Components

- Transpose graph by flipping edges
- Trim trivial vertices
  - Only in/out edges
- Forward Traversal:
  - Label vertices with max Vid of connecting vertex
- Backward traversal
  - Traverse from Vid and label all it can reach
- Remove SCC & repeat for rest of graph

```
Coloring(G(V, E))
G^T = constructTransposeGraph(G)
while V ≠ ∅
 Trim G and G^T
 // colors vertices into disjoint color sets
 MaxForwardReachable(G, start from every v ∈ V)
 foreach p ∈ P in parallel:
  if color(p) == p:
    let S_p be the vertices colored p
    SCC_p = S_p ∩ BackwardReachable(G^T, p)
    remove SCC_p from G and G^T
```

Figure 3: Original Coloring algorithm for computing SCCs [38].

```
public void doFwStart() {
  value(). colorID = getId();
  sendMessages(getOutgoingNeighbors(),
                 new SCCMessage(getId()));}
public void doFwRest(Iterable <SCCMessage> messages) {
  int maxColorID = findMaxColorID(messages);
  if (maxColorID > value().colorID) {
    sendMessages(getOutgoingNeighbors(),
                 new SCCMessage(value().colorID));
    updateGlobalObject("updated-vertex-exists", true);}}
```

Figure 4: *SCCVertex* subroutines for the Forward-Traversal phase.

Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees, Da Yan, et al, VLDB 2014
Optimizing Graph Algorithms on Pregellike Systems, Semih Salihoglu, et al, VLDB 2014

Betweenness Centrality

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

- Intuition
- Forward traversal
  - SSSP from each vertex
  - Keep track of parent vertex used to arrive at shortest path
- Reverse traversal
  - Accumulate values of centrality from child to parent
- Repeat for each vertex

A Faster Algorithm for Betweenness Centrality, Ulrik Brandes

Approximate BC

Algorithm 1 High level pseudo code of the algorithm of approximate betweenness centrality computation.

APPROXIMATEBETWEENNESS
Require. A network (graph) $\mathbf{G}$, the number of samples T.
Ensure. Betweenness centrality of vertices of $\mathbf{G}$.
Compute probabilities $p_1, \ldots, p_n$
for all vertices $v \in V(\mathbf{G})$ **do**
    $B[v] \leftarrow 0$
end for
for all $t = 1$ **to** T **do**
    Select a vertex i with probability p_i
    Form the SPD D rooted at i
    Compute dependency scores of every vertex v on i
    for all vertex $v \in V(\mathbf{G})$ **do**
        $B[v] \leftarrow B[v] + \frac{\delta_{i\bullet}(v)}{p_i}$
    end for
end for
for all $i \in \{1, \ldots, n\}$ **do**
    $B[i] \leftarrow \frac{B[i]}{T}$
end for
return B

An Efficient Algorithm for Approximate Betweenness Centrality Computation Mostafa Haghir Chehreghani

Others

- Triangle Count: Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis, Louise Quick, et al, ASONAM 2012
- Label Propagation: One Trillion Edges: Graph Processing at FacebookScale, Avery Ching, et al, VLDB 2015
- Graph Coloring, Minimum Spanning Forest: Optimizing Graph Algorithms on Pregellike Systems, Semih Salihoglu, et al, VLDB 2014

GoFFish

Subgraph-centric, Time-series graph processing

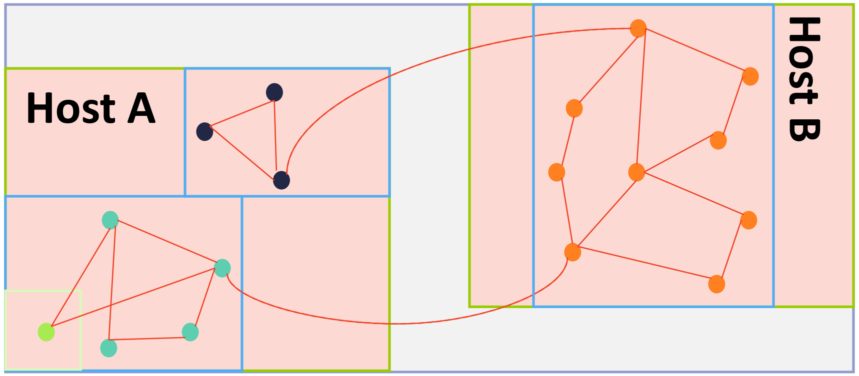
Vertex➲Subgraph Centric

- Challenges with Pregel
  - *Ab initio* algorithm design
  - Large number of messages between vertices [1]
    - O(e) for pagerank in each superstep, even for collocated vertices
    - Network & memory pressure
  - Many supersteps to converge
    - O(diameter): Ok for powerlaw graphs, poor for spatial graphs
    - Coordination overhead accumulates
- Idea: Coarsen the unit of computation to subgraph [2]
  - Weakly connected component within a partition
  - Logic for subgraph given, progress on full subgraph in one superstep
  - Reduces explicit messaging, number of supersteps
  - Leverage shared memory algorithms on subgraph

[1] Optimizations and Analysis of BSP Graph Processing Models on Public Clouds, IPDPS 2013
[2] GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics, EuroPar, 2014

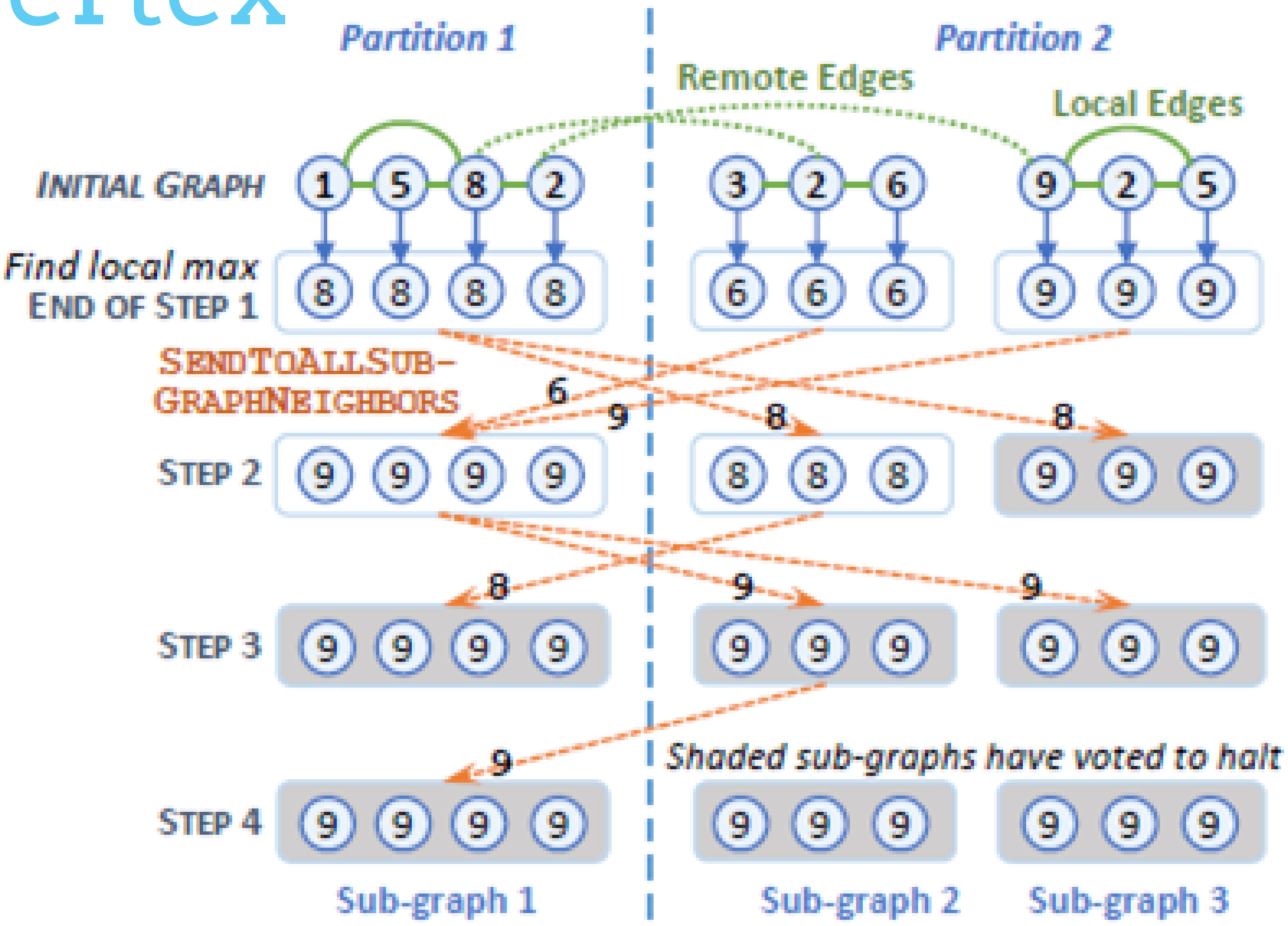
Graph Data Model

- Designed for "sub-graph" centric *distributed* computing
  - Graphs
    - Partitions … *Distributed evenly across machines*
      - Sub-graphs … *Logical unit of operation*
        - Vertices



- Sub-graph is unit of *distributed* data access & operation
  - Extends Google Pregel/Apache Giraph's vertex-centric BSP model … *no global view*

Sub-graph Centric Max Vertex



Sub-graph Centric Max Vertex

Algorithm 2 Max Vertex using Sub-Graph Centric Model

```
procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
    if superstep = 1 then                    ▷ Find local max in subgraph
        mySG.value ← −∞
        for all Vertex myVertex in mySG.vertices do
            if mySG.value < myVertex.value then
                mySG.value ← myVertex.value
    hasChanged = (superstep == 1) ? true : false
    while M.hasNext do
        Message m ← M.next
        if m.value > mySG.value then
            mySG.value ← m.value
            hasChanged = true
    if hasChanged then
        SENDTOALLSUBGRAPHNEIGHBORS(mySG.value)
    else
        VOTETOHALT( )
```


GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics, Yogesh Simmhan et al, EuroPar, 2012

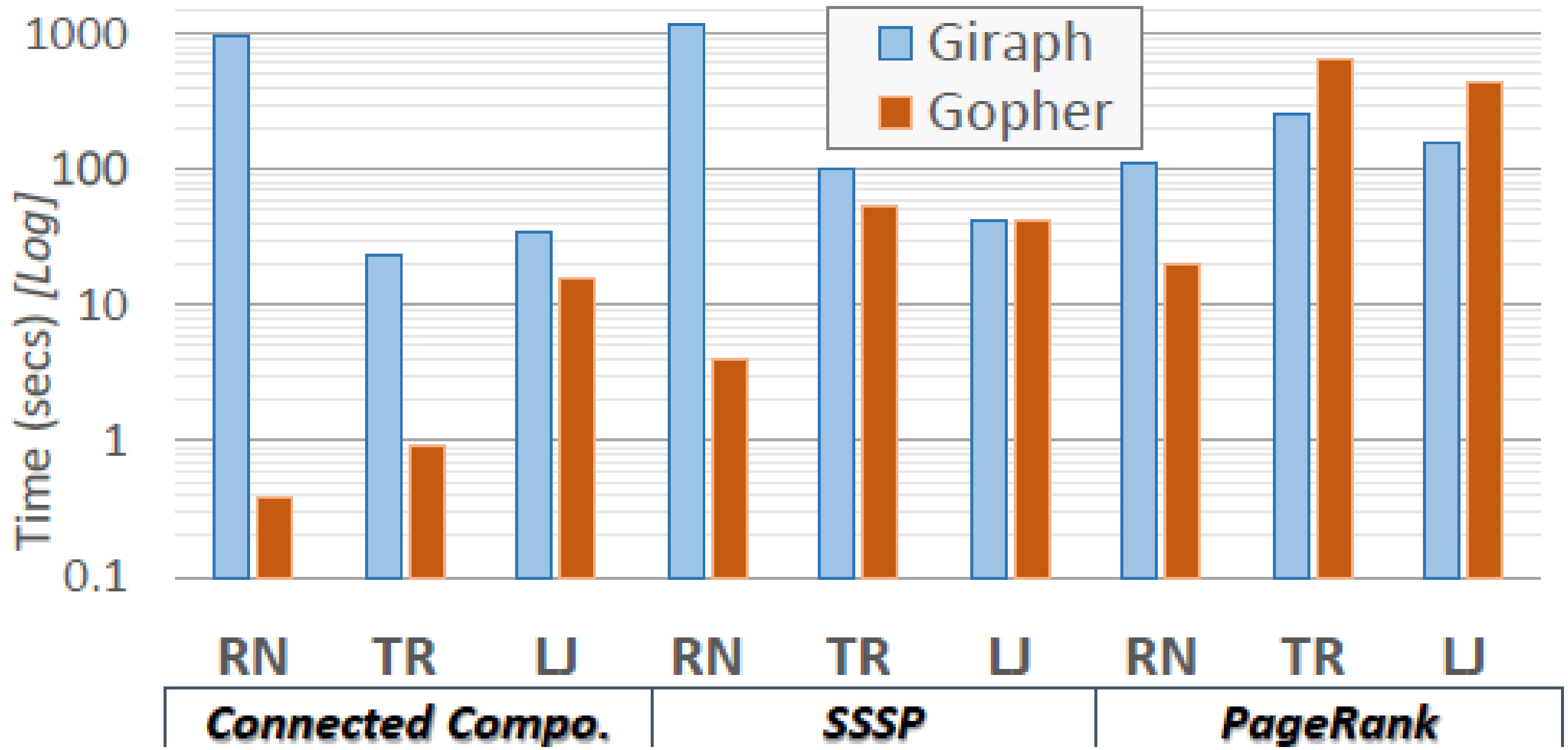
SSSP

Algorithm 3 Sub-Graph Centric Single Source Shortest Path

```
procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
    openset ← ∅                          ▷ Vertices with improved distances
    if superstep = 1 then                ▷ Initialize distances
        for all Vertex v in mySG.vertices do
            if v = SOURCE then
                v.value ← 0              ▷ Set distance to source as 0
                openset.add(v)           ▷ Distance has improved
            else
                v.value ← −∞             ▷ Not source vertex
    for all Message m in M do            ▷ Process input messages
        if mySG.vertices[m.vertex].value > m.value then
            mySG.vertices[m.vertex].value ← m.value
            openset.add(m.vertex)        ▷ Distance improved
            ▷ Call Dijkstras and get remote vertices to send updates
    remoteSet ← DIJKSTRAS(mySG, openset)
            ▷ Send new distances to remote sub-graphs/vertices
    for all ⟨remoteSG,vertex,value⟩ in remoteSet do
        SENDTOSUBGRAPHVERTEX(remoteSG, vertex, value)
    VOTETOHALT( )
```


GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics, Yogesh Simmhan et al, EuroPar, 2012

Performance on Single Graphs

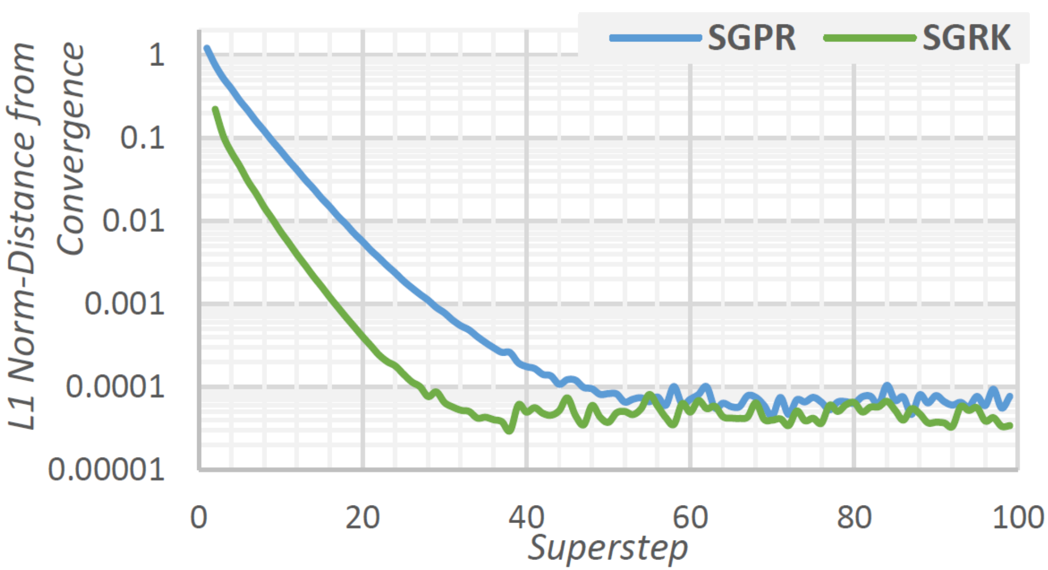


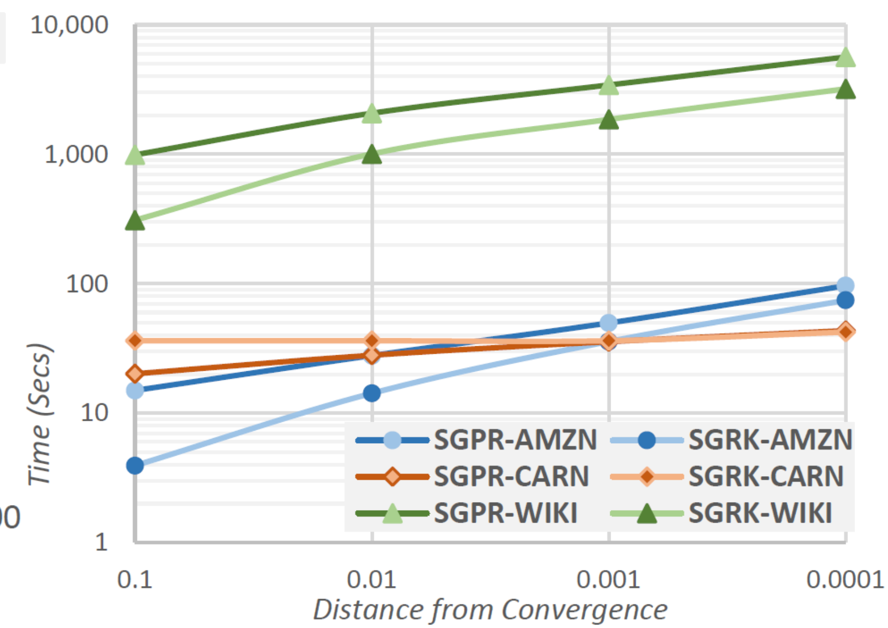
Data Set	Vertices	Edges	Diameter
RN: CA Road Network	1,965,206	2,766,607	849
TR: Internet Tracesroutes	19,442,778	22,782,842	25
LJ: LiveJournal Social N/W	4,847,571	68,475,391	10

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

Algorithmic Benefits on PageRank

- PageRank → Block Rank → Subgraph Rank
  - Coarse-grained rank for "good" initialization





Subgraph Rank: PageRank for Subgraph-Centric Distributed Graph Processing, Badam & Simmhan, *COMAD*, 2014

Reading

- Pregel: A System for Large-Scale Graph Processing, Malewicz, et al, *SIGMOD* 2010
- GPS: A Graph Processing System, Salihoglu and Widon, *SSDBM*, 2013
- GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics, Simmhan, et al, *EuroPar*, 2014