PARALLEL & DISTRIBUTED GRAPH PROCESSING

Analysis of Large Scale Social Networks

Bart Thijs

The focus of this lecture is on the different approaches taken to implement a distributed graph processing framework.

Pregel: A system for large-scale graph processing

Overview of Distributed Graph Processing (Kalavri et al., 2018)

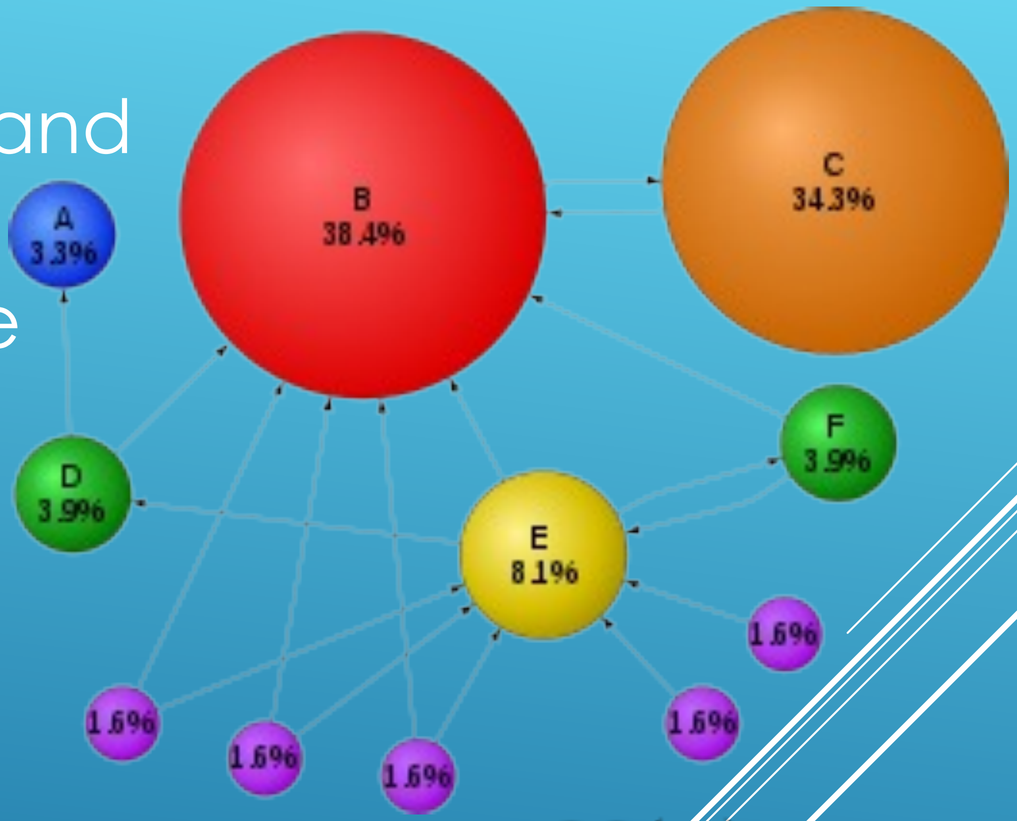
The implementation of a graph algorithm in a DGP (Thijs, 2017)

GraphX as a DGP-framework.

DISTRIBUTED GRAPH PROCESSING

Pagerank :

- Rank webpages based on number and quality of links
- More important pages receive more (important) links
- Value propagation algorithm
- Evenly distributed outgoing links
- Random clicking (1-d)=15%

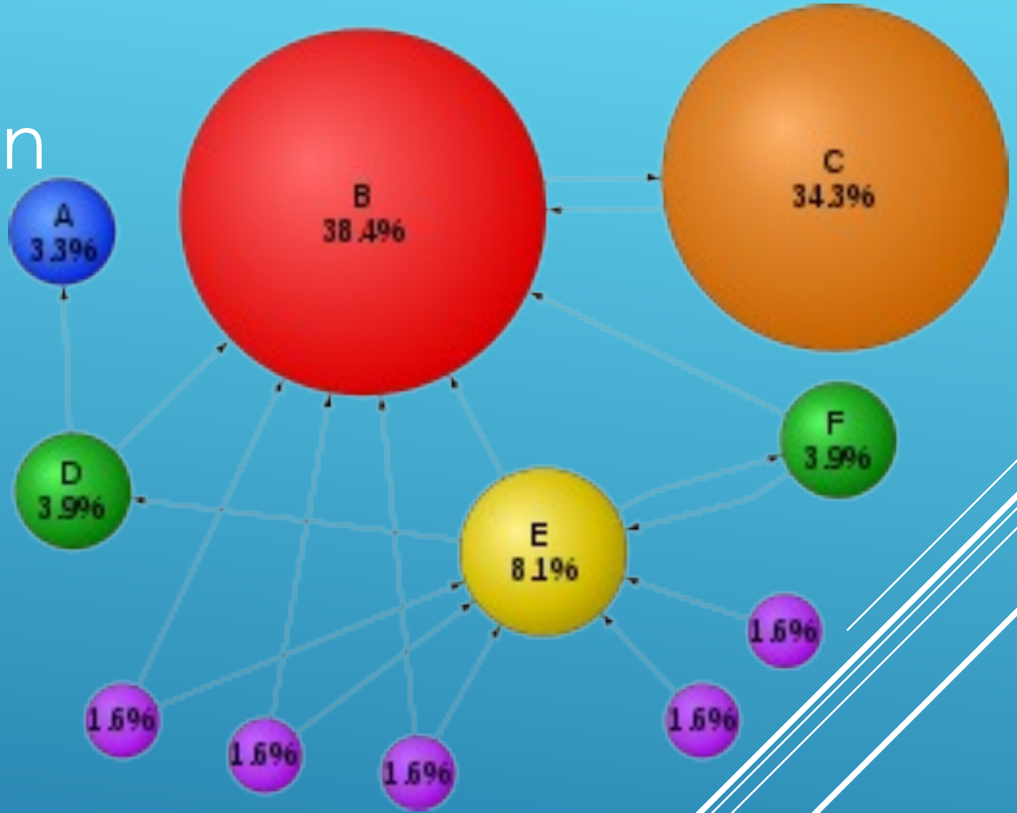


$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

FIRST

Pagerank :

- Matrix Approach starts from transition matrix
- In fact, the Pagerank vector is the eigenvector associated with the eigenvalue of 1.



$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

FIRST

How can we efficiently calculate Pagerank?

- Huge data set, too big for regular hardware / software
- Dynamic: ever growing and changing

Need for specialized graph processing systems vs algebraic approach:

- Express program logic as close as possible to data model

WHY?

Three challenges for graph analytical frameworks

1. Capacity (Storage: Memory and Disk)
2. Performance (due to RAM-reads exceeding cache)
3. Implementation (parallelization, small tasks at node and edge level)

THREE CHALLENGES (HONG ET AL)

2 possible approaches for parallelization of the graph analysis:

- Multi-core computers with shared memory
  - HPC (Cerebro @ KU Leuven),
- Distributed computer systems
  - Hadoop/Spark (EMR @ Amazon)

SOLUTION?

Implementing graph algorithms becomes very difficult:

- Right level or form of parallelization or distribution with appropriate communication and synchronyzation
- Correct implementation of described algorithm
- Optimized for each particular environment
- Fault tolerance

Who has all these skills/expertise?

SOLUTION?

Shared memory frameworks

- SNAP
- GraphLab
- Ligra
- Green Marl

PARALLEL GRAPH PROCESSING

Frameworks for distributed graph processing

- High level of abstraction
- Partitioned view of data and methods for read/write in and communication between partitions.

Examples:

- Pregel (Google)
- Giraph (Facebook)
- GraphX (Spark)

DISTRIBUTED GRAPH PROCESSING

Definition

- A **distributed graph programming** model is an abstraction of the underlying computing infrastructure that allows for the definition of graph data structures and the expression of graph algorithms
  - Data partitioning and communication is hidden

KALAVRI (2018)

DISTRIBUTED GRAPH PROCESSING

Bulk-Synchronous Parallel (Vaillant, 1990)

Combination three attributes:

A. A number of components/instances performing processing or memory functions:
   1. perform local computational steps:
   2. Transmit messages to other components:
   3. Receive messages as input for next superstep
B. A router that delivers messages point to point between components
C. Facilities for synchronizing all components at regular intervals through the use of supersteps. (Not required)

A superstep is finalized after all components finish their task

Large scale graph processing at Google:

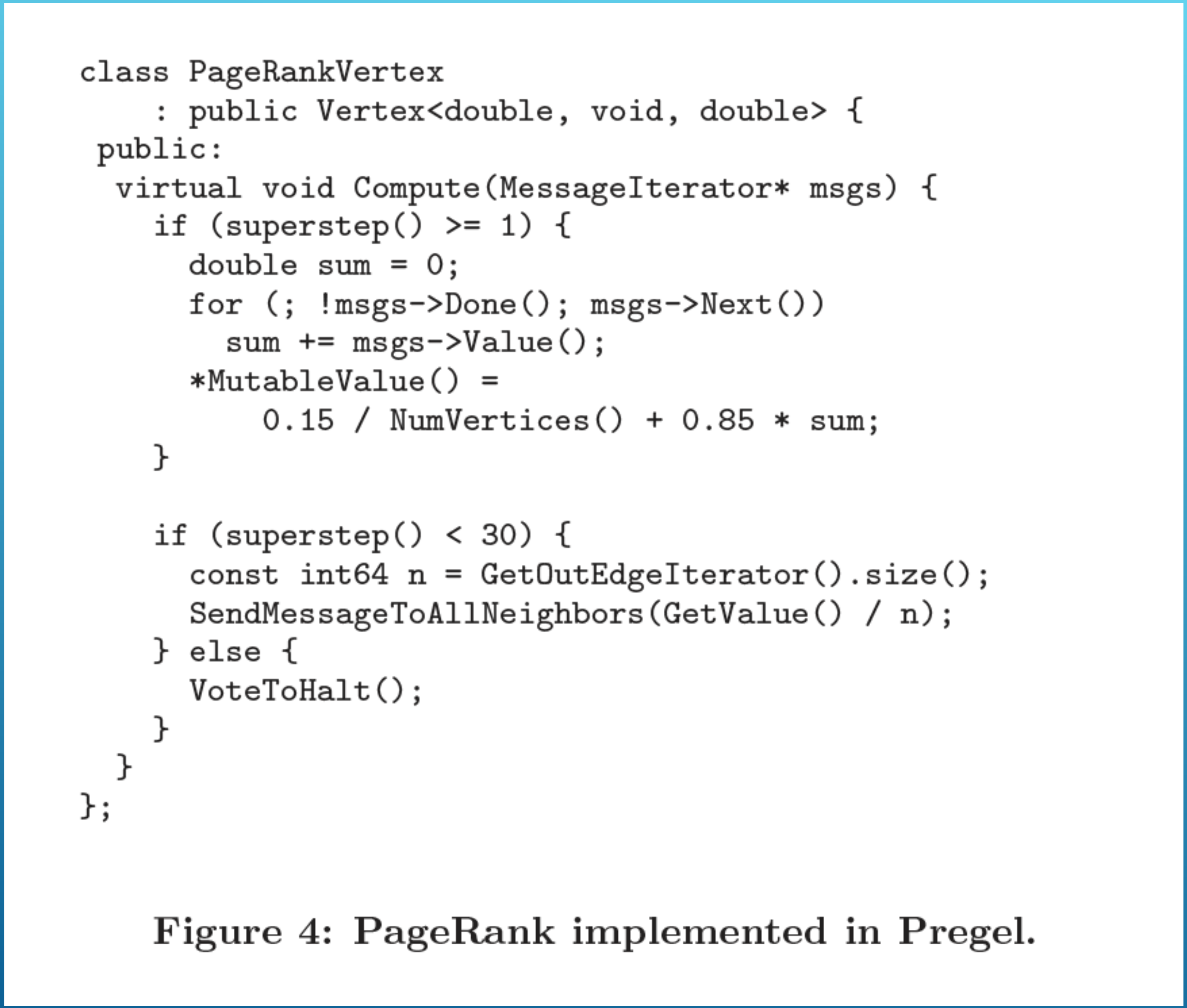
The BSP concept is ported towards networked data:

A superstep consists of tasks performed at the vertices

1. Input of messages from previous superstep to nodes
2. Start execution at node-level of computations and change the state of vertex and (outgoing) edges
3. Collect messages to nodes (along (outgoing) edges).

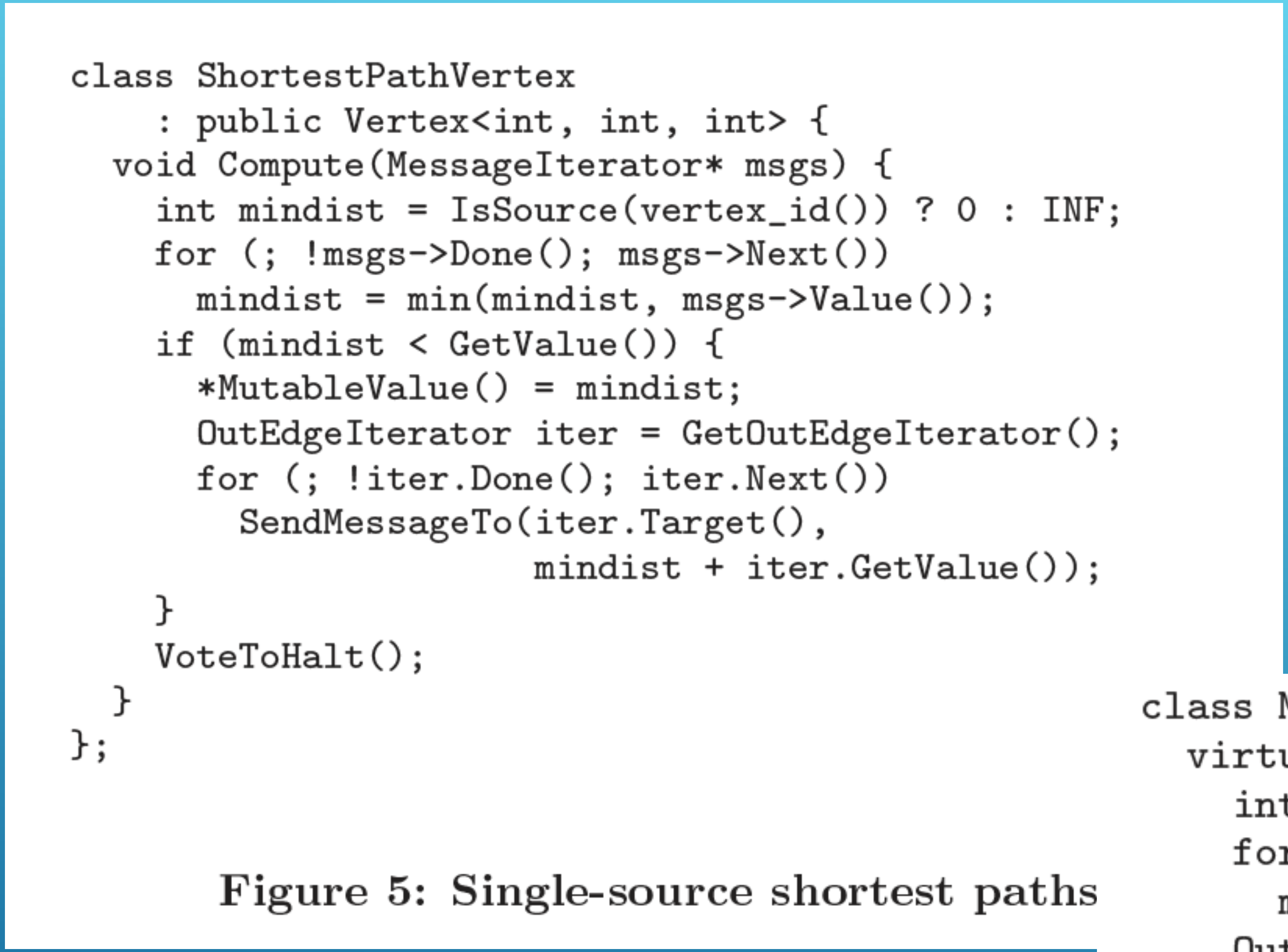
Vertices are either in Active or Inactive State

DISTRIBUTED GRAPH PROCESSING: PREGEL



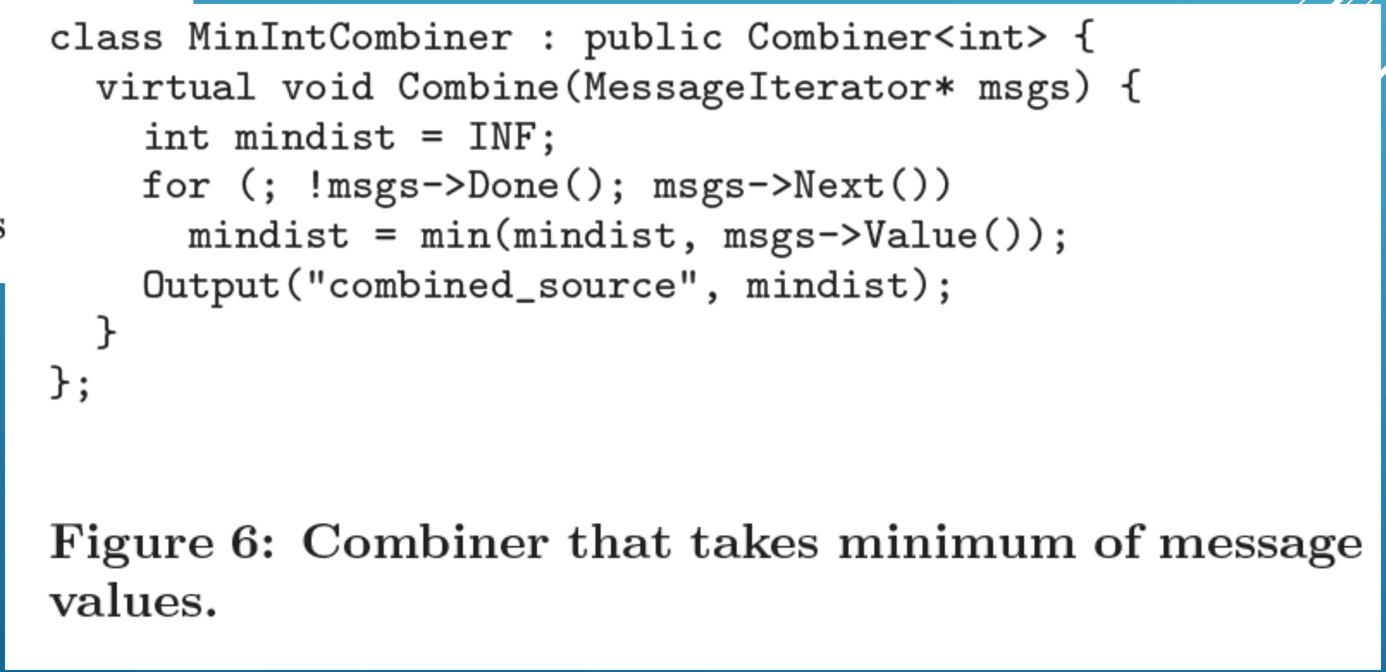
```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Figure 4: PageRank implemented in Pregel.



```
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();
  }
};
```

Figure 5: Single-source shortest paths



```
class MinIntCombiner : public Combiner<int> {
  virtual void Combine(MessageIterator* msgs) {
    int mindist = INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    Output("combined_source", mindist);
  }
};
```

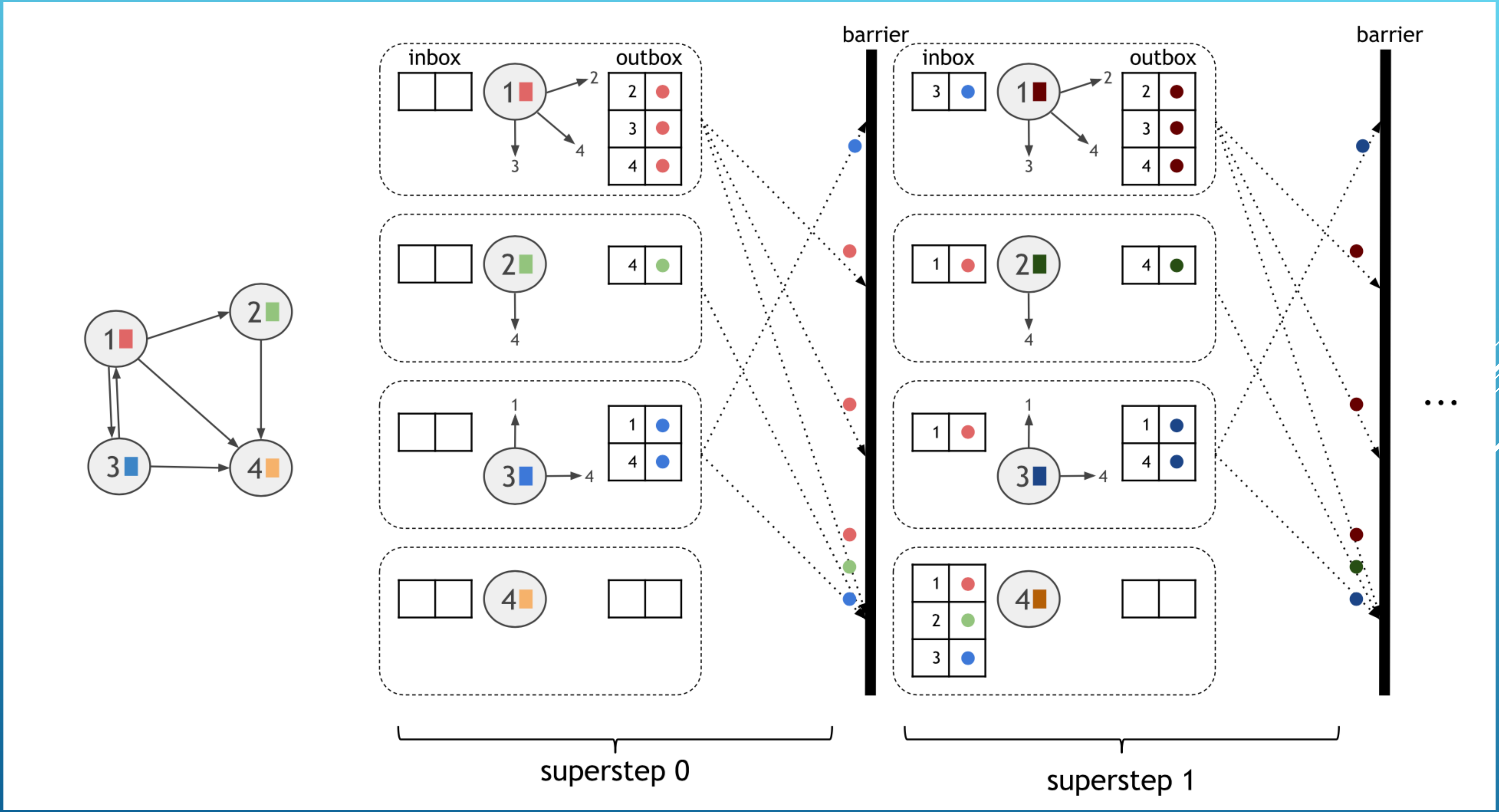
Figure 6: Combiner that takes minimum of message values.

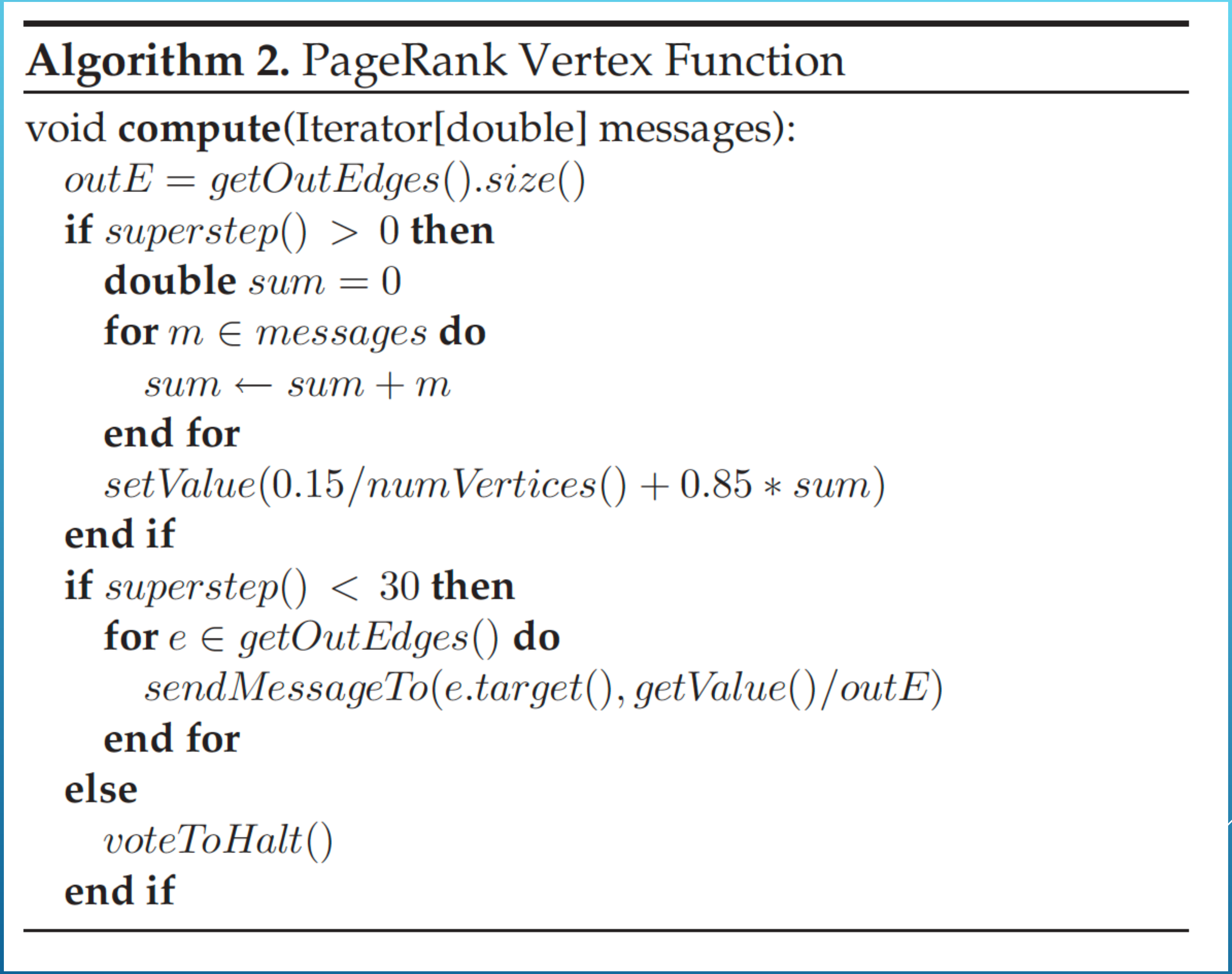
Six models

- Vertex Centric
- Scatter-Gather
- Gather-Sum-Apply-Scatter
- *Subgraph-Centric*
- *Filter-Process*
- *Graph Traversals*

KALAVRI (2018)

Vertex Centric



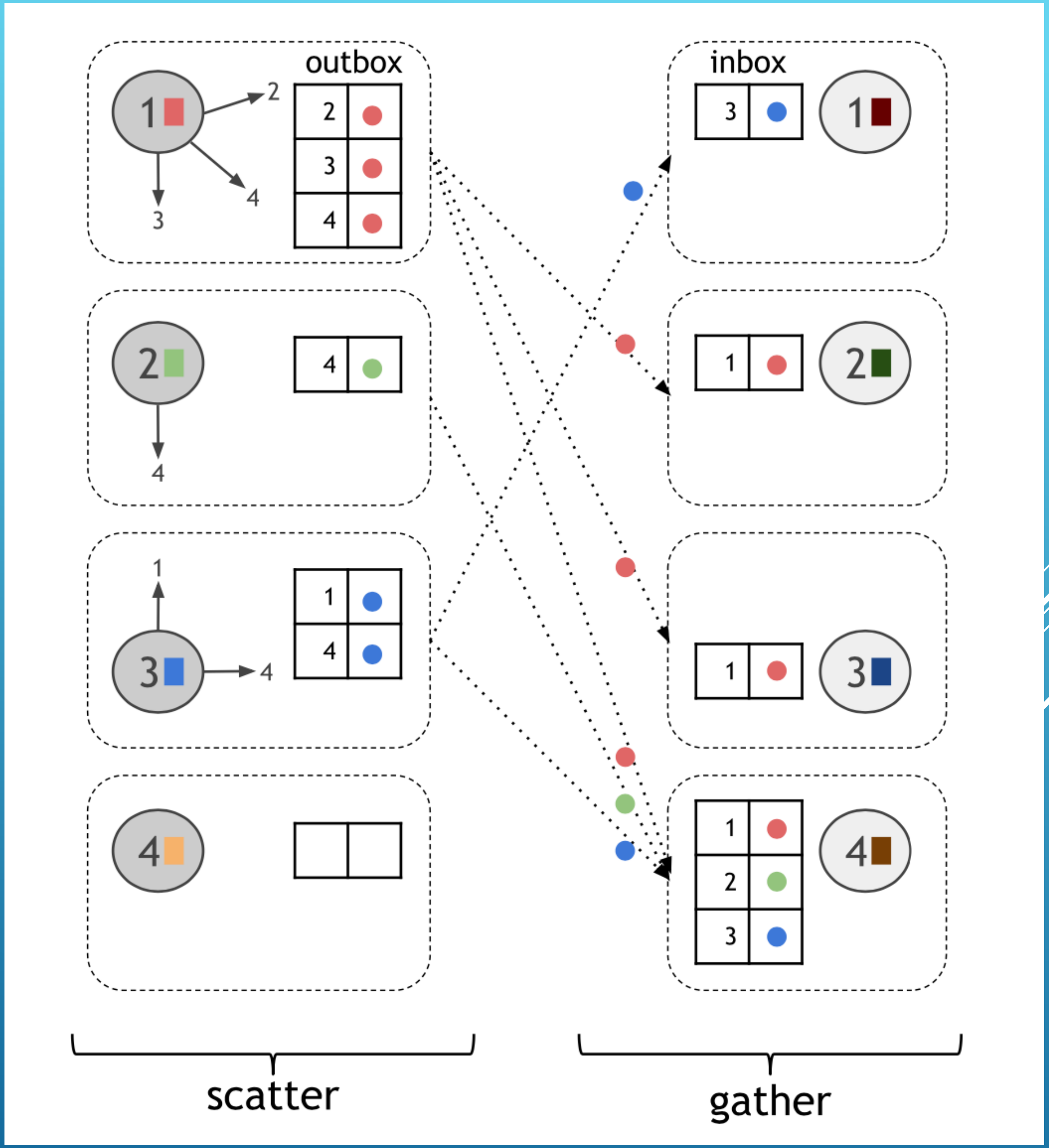


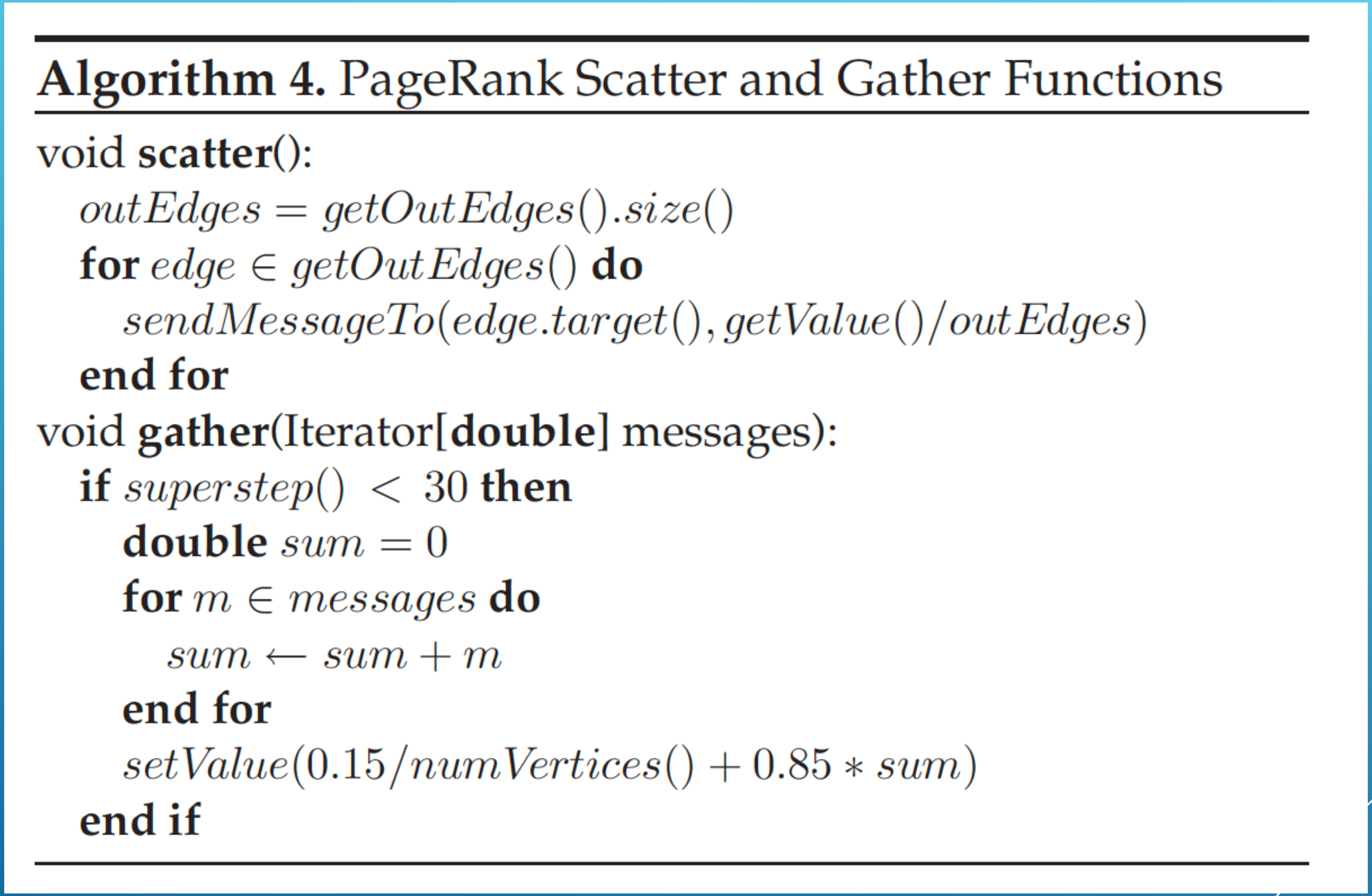
Algorithm 2. PageRank Vertex Function

```
void **compute**(Iterator[double] messages):
    $outE = getOutEdges().size()$
    **if** $superstep() > 0$ **then**
        **double** $sum = 0$
        **for** $m \in messages$ **do**
            $sum \leftarrow sum + m$
        **end for**
        $setValue(0.15/numVertices() + 0.85 * sum)$
    **end if**
    **if** $superstep() < 30$ **then**
        **for** $e \in getOutEdges()$ **do**
            $sendMessageTo(e.target(), getValue()/outE)$
        **end for**
    **else**
        $voteToHalt()$
    **end if**
```

Scatter Gather

- barrier after the Gather
- Gather has access to the individual incoming messages
- No access to in and outgoing messages.



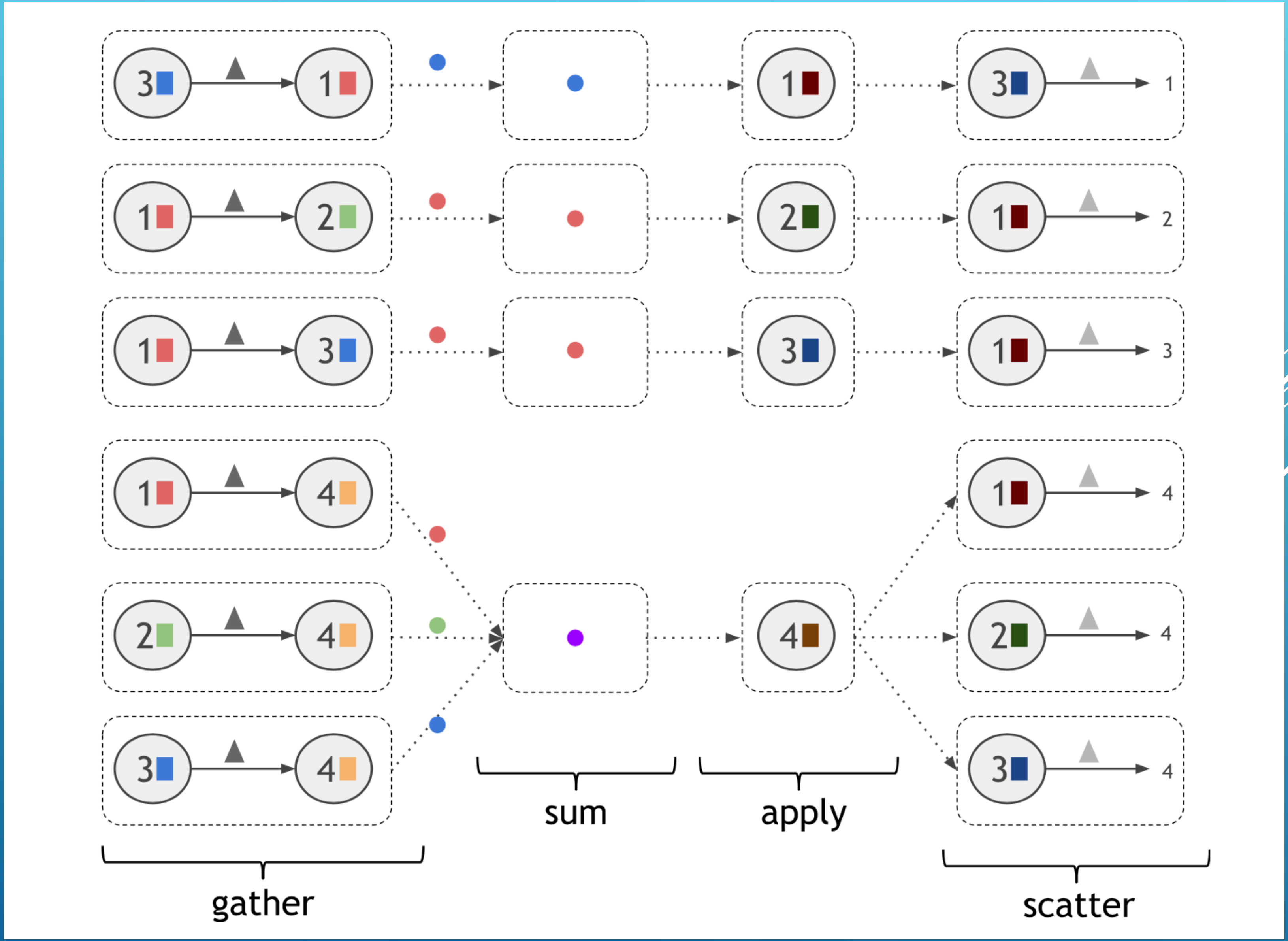


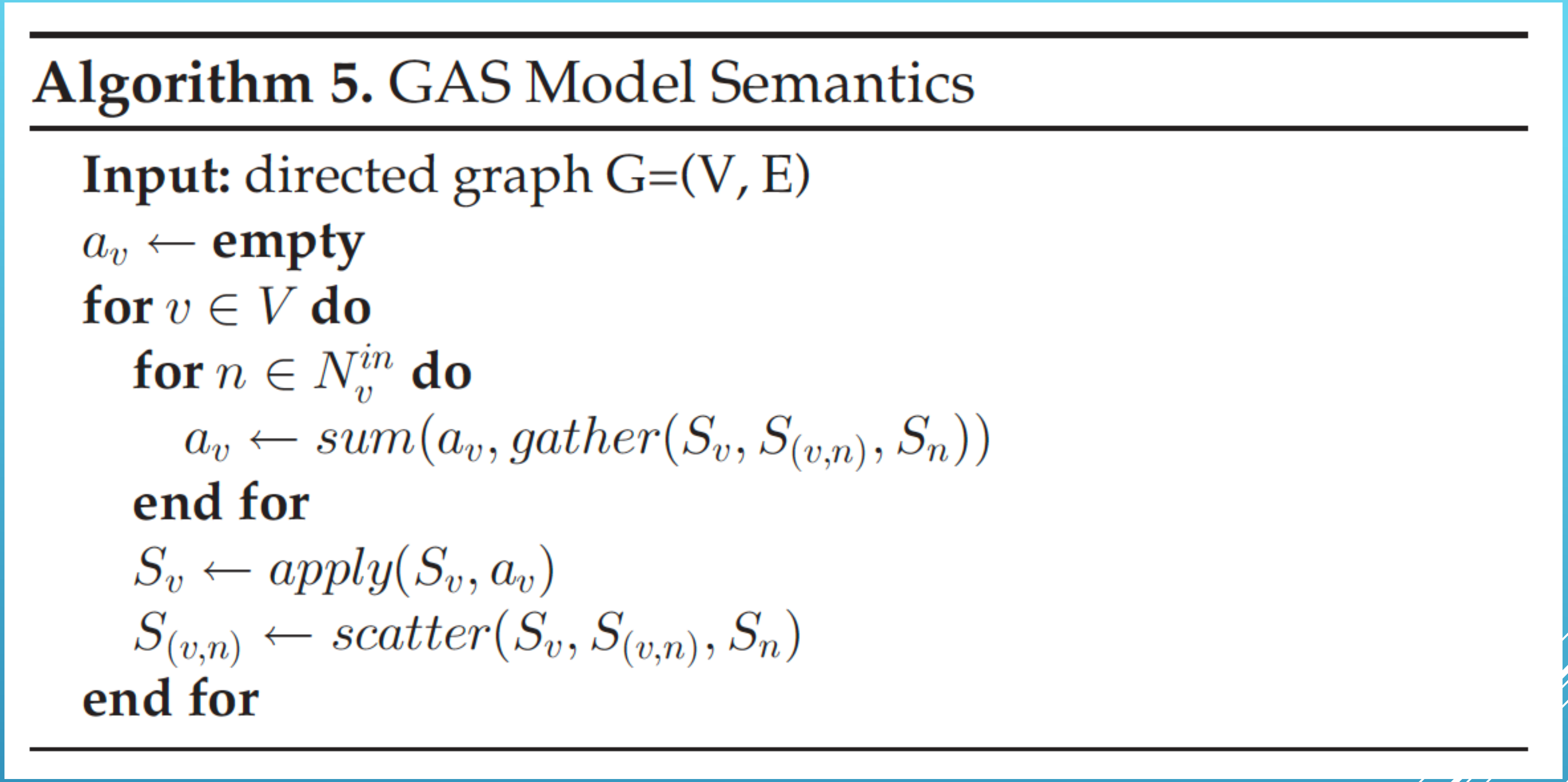
Algorithm 4. PageRank Scatter and Gather Functions

```
void **scatter**():
    $outEdges = getOutEdges().size()$
    **for** $edge \in getOutEdges()$ **do**
        $sendMessageTo(edge.target(), getValue()/outEdges)$
    **end for**
void **gather**(Iterator[**double**] messages):
    **if** $superstep() < 30$ **then**
        **double** $sum = 0$
        **for** $m \in messages$ **do**
            $sum \leftarrow sum + m$
        **end for**
        $setValue(0.15/numVertices() + 0.85 * sum)$
    **end if**
```

Gather-Sum-Apply-Scatter

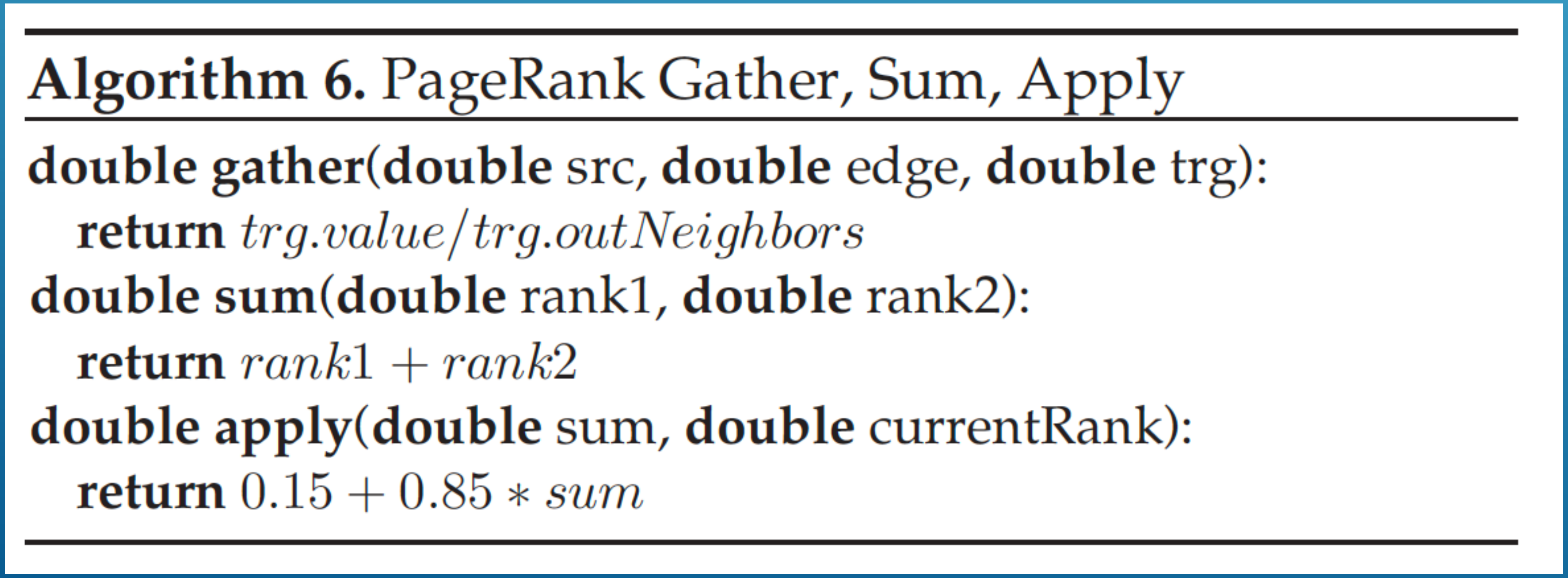
- Parallelizing over the edges in the graph
- Sum = commutative & associative function
- Gather - Sum is similar to Map-Reduce





Algorithm 5. GAS Model Semantics

```
Input: directed graph G=(V, E)
$a_v \leftarrow$ empty
for $v \in V$ do
    for $n \in N_v^{in}$ do
        $a_v \leftarrow sum(a_v, gather(S_v, S_{(v,n)}, S_n))$
    end for
    $S_v \leftarrow apply(S_v, a_v)$
    $S_{(v,n)} \leftarrow scatter(S_v, S_{(v,n)}, S_n)$
end for
```



Algorithm 6. PageRank Gather, Sum, Apply

```
double gather(double src, double edge, double trg):
    return $trg.value/trg.outNeighbors$
double sum(double rank1, double rank2):
    return $rank1 + rank2$
double apply(double sum, double currentRank):
    return $0.15 + 0.85 * sum$
```

A nearest neighbour search algorithm, implemented within a DGP-framework (Thijs, 2017)

Bi-partite directed network connecting scientific papers with cited references.

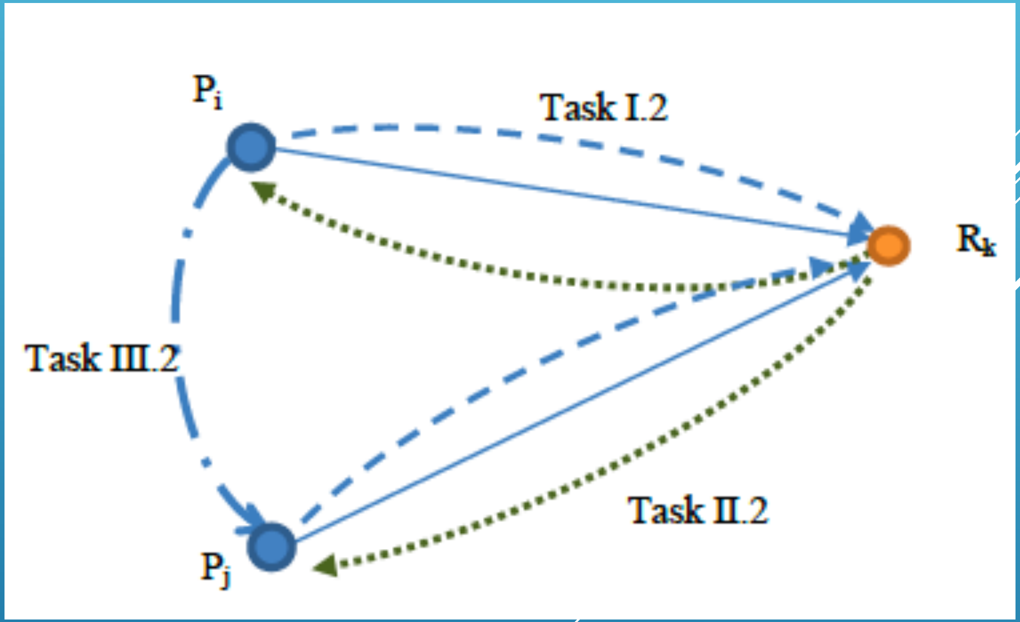
Goal is to detect similar papers based on the number of joint references normalized by the number of references of each publication.

DISTRIBUTED GRAPH PROCESSING

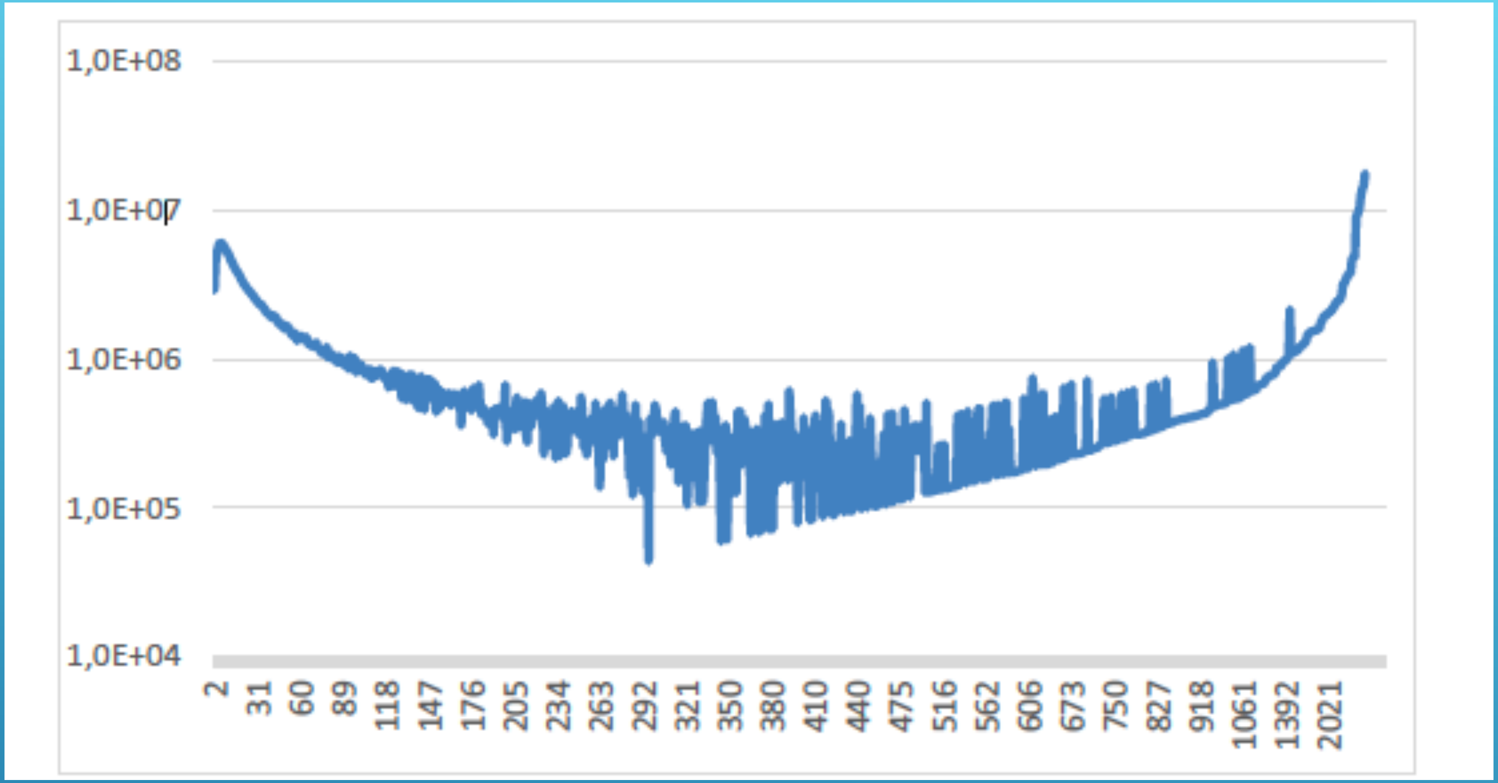
Implementation in a GAS approach with 3 distinct supersteps

1. Paper sends <u>degree and identifier</u> to cited reference
2. Cited reference sends back list with degrees and identifiers to its citing papers
3. Papers calculate similarity with other papers and sends message to make connection.

Amount messages depend on degree



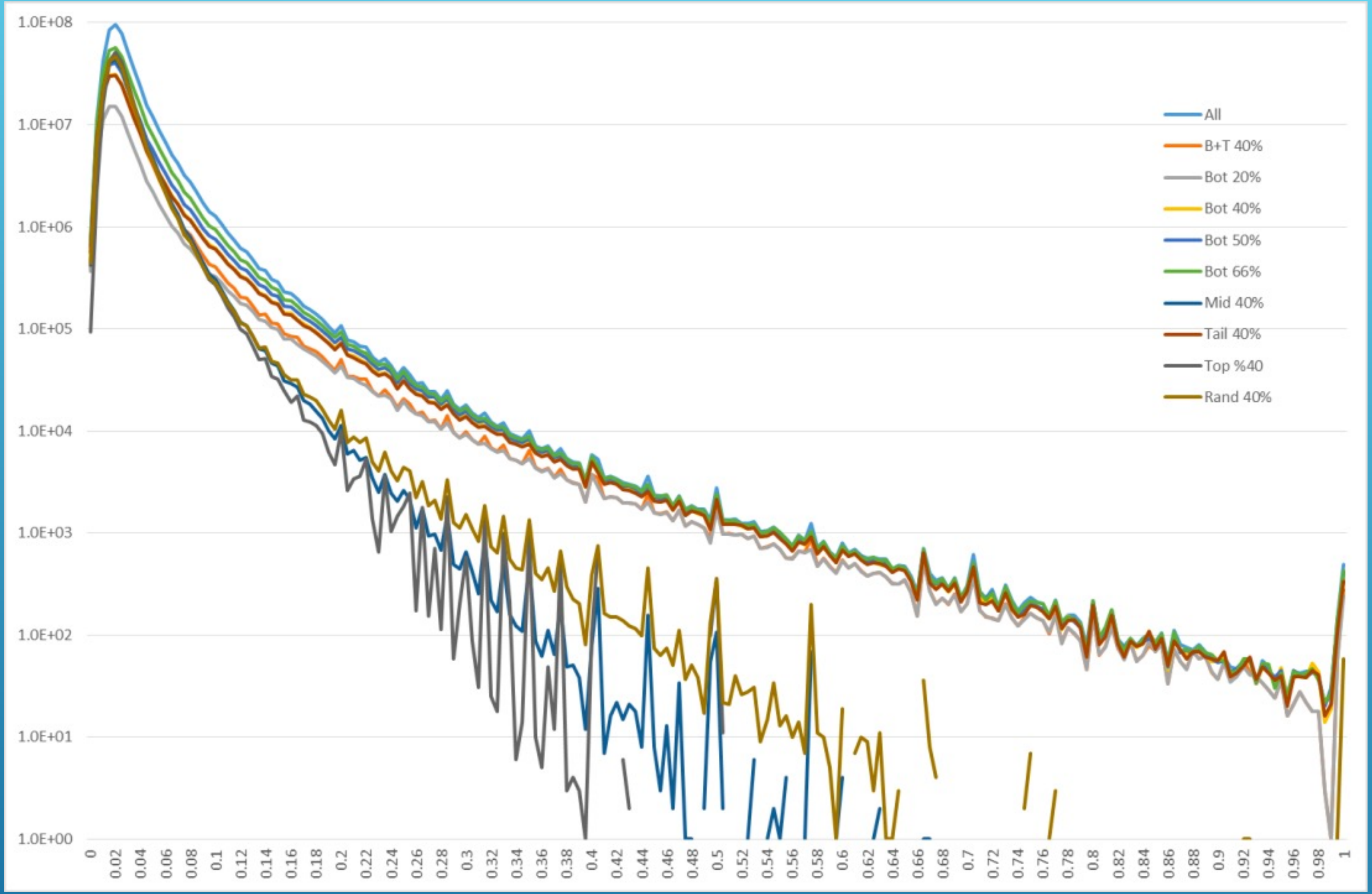
DISTRIBUTED GRAPH PROCESSING



Nodes can use specific functions to reduce information in message and to select recipients of message.

High degree nodes become less likely to receive message.
Identifiers are sorted and messages only contain list with lower ranked ID

DISTRIBUTED GRAPH PROCESSING



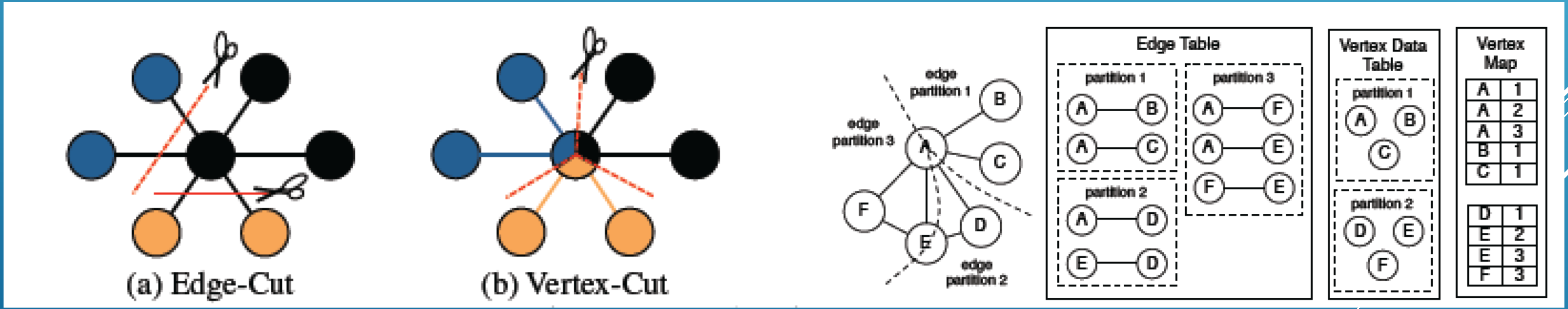
DISTRIBUTED GRAPH PROCESSING

DISTRIBUTED GRAPH PROCESSING: GRAPHX

GraphX:

Inspired by the GAS approach and introduces for Vertex Cuts which ensures equal distribution of edges across workers.

Combiners reduce message cost across instances/workers: combiner functions must be commutative and associative.



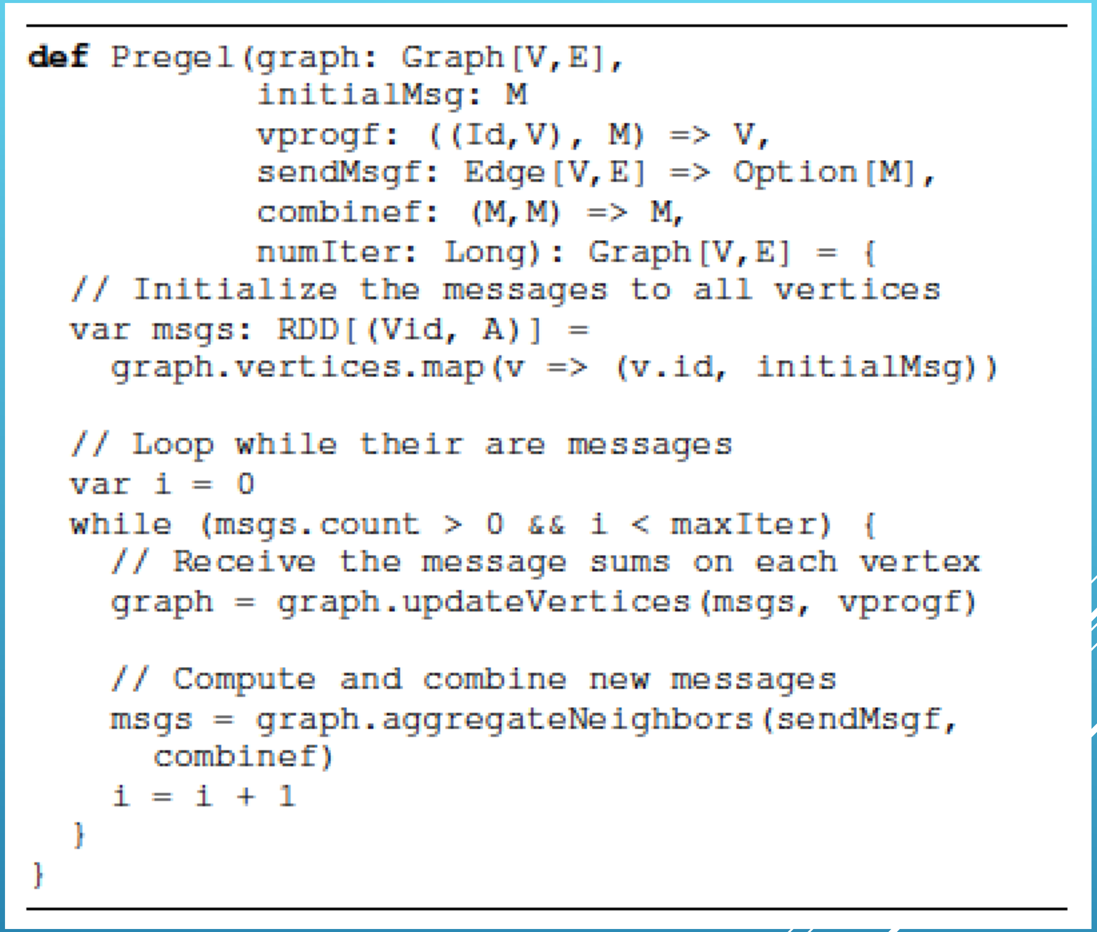
Review GraphX

- Does their Pregel implementation follow the genuine Pregel model?
- How many functions are required for the implementation of the Pregel function.

DISTRIBUTED GRAPH PROCESSING

Pregel in GraphX: Each Superstep

1. Aggregate received Messages
2. Calculate new attributes
3. Send messages to neighbours



```
def Pregel(graph: Graph[V,E],
           initialMsg: M
           vprogf: ((Id,V), M) => V,
           sendMsgf: Edge[V,E] => Option[M],
           combinef: (M,M) => M,
           numIter: Long): Graph[V,E] = {
  // Initialize the messages to all vertices
  var msgs: RDD[(Vid, A)] =
    graph.vertices.map(v => (v.id, initialMsg))

  // Loop while their are messages
  var i = 0
  while (msgs.count > 0 && i < maxIter) {
    // Receive the message sums on each vertex
    graph = graph.updateVertices(msgs, vprogf)

    // Compute and combine new messages
    msgs = graph.aggregateNeighbors(sendMsgf,
      combinef)
    i = i + 1
  }
}
```

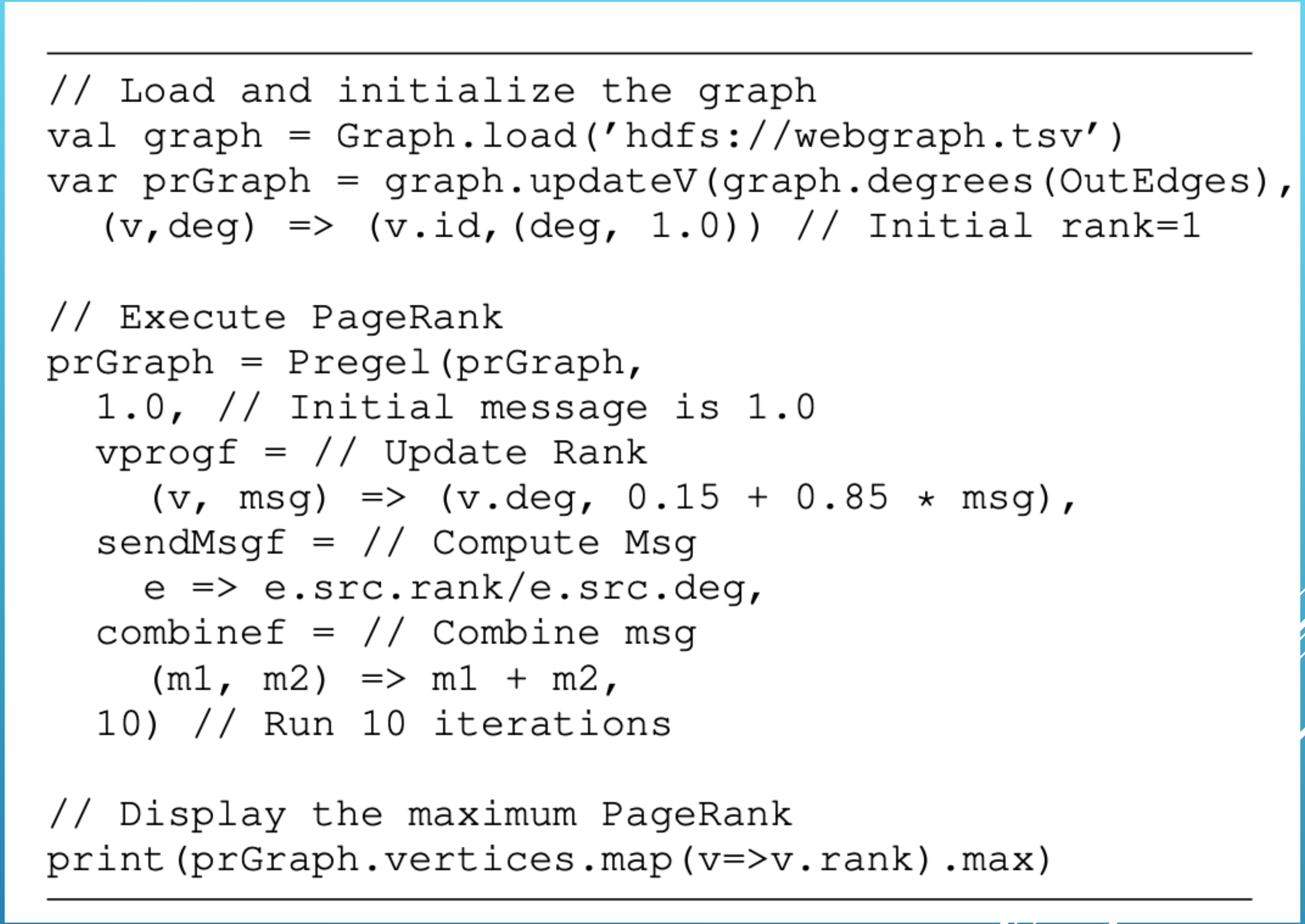
DISTRIBUTED GRAPH PROCESSING: GRAPHX

DISTRIBUTED GRAPH PROCESSING: GRAPHX

Page Rank in GraphX: Each Superstep

- Initial message

1. Combine Messages
2. Update Rank
3. Send Messages



```
// Load and initialize the graph
val graph = Graph.load('hdfs://webgraph.tsv')
var prGraph = graph.updateV(graph.degrees(OutEdges),
   (v,deg) => (v.id,(deg, 1.0)) // Initial rank=1

// Execute PageRank
prGraph = Pregel(prGraph,
  1.0, // Initial message is 1.0
  vprogf = // Update Rank
    (v, msg) => (v.deg, 0.15 + 0.85 * msg),
  sendMsgf = // Compute Msg
    e => e.src.rank/e.src.deg,
  combinef = // Combine msg
    (m1, m2) => m1 + m2,
  10) // Run 10 iterations

// Display the maximum PageRank
print(prGraph.vertices.map(v=>v.rank).max)
```