Lecture 12: Graph processing

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- Looking behind the scenes of MapReduce: HDFS & Scheduling
- Algorithm design for MapReduce
- A high-level language for MapReduce: Pig Latin 1 & 2
- MapReduce is not a database, but HBase nearly is
- **Lets iterate a bit: Graph algorithms & Giraph**
- How does all of this work together? ZooKeeper/Yarn

Learning objectives

- **Explain** the drawbacks of MapReduce-base implementations of graph algorithms
(focus in the last lecture)
- **Explain and apply** the idea behind BSP
- **Discuss** the architecture of Pregel & Giraph
- **Implement** basic graph problems within the Giraph framework

Last time

Graphs

- Ubiquitous in modern society
  - Hyperlink structure of the Web
  - Social networks
    - Email flow
    - Friend patterns
  - Transportation networks

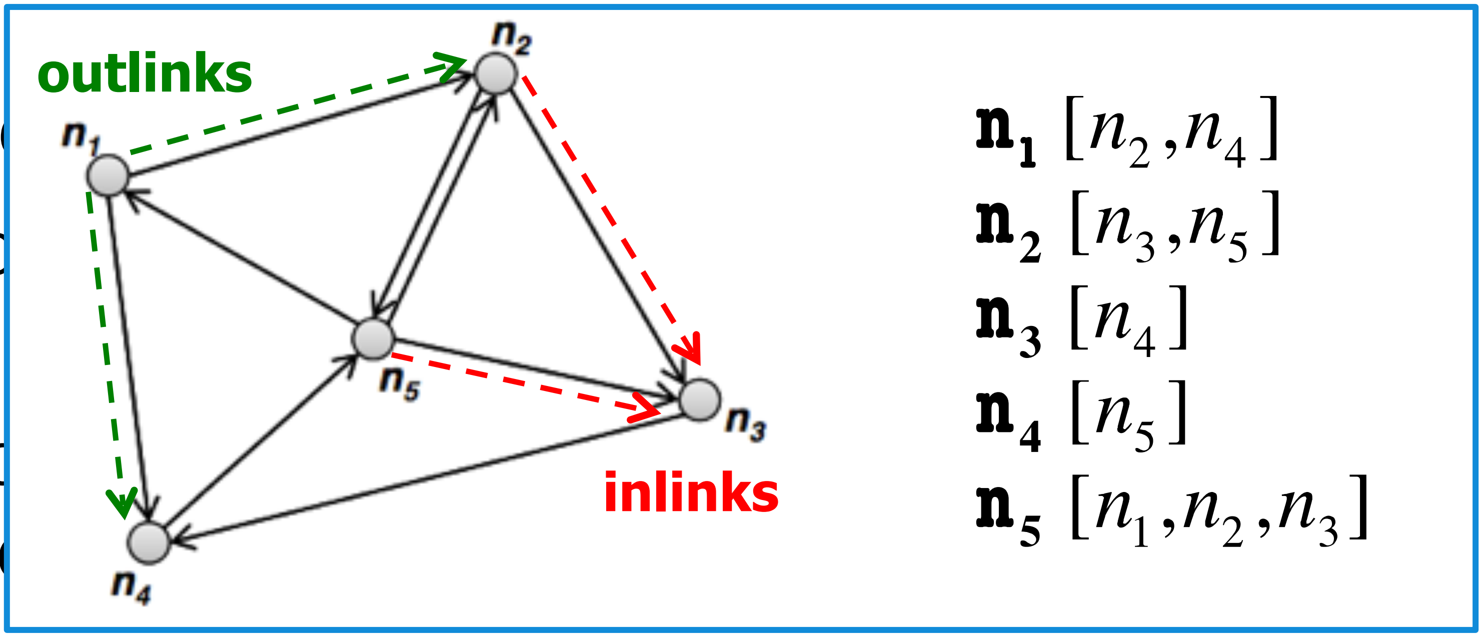
- Nodes and links can be annotated with **metadata**
  - Social network nodes: age, gender, interests
  - Social network edges: relationship type (friend, spouse, foe, etc.), relationship importance (weights)

Adjacency list

- A more **compressed** representation than adjacency matrices
  - On sparse graphs

- Only edges **that exist** are encoded in adjacency lists

- Two options to encode **un**
  - Encode each edge twic
    adjacency list)
  - Impose an order on no
    adjacency list of the no



- **Disadvantage**: some graph operations are more difficult compared to the adjacency matrix representation

Single-source shortest path

Standard solution: Dijkstra's algorithm

Task: find the shortest path from a **source node** to all other nodes in the graph

Input:
- directed connected graph in adjacency list format
- edge distances in w
- source s

```
DIJKSTRA(G, w, s)
    d[s] ← 0
    for all vertex v ∈ V do
        d[v] ← ∞
    Q ← {V}
    while Q ≠ ∅ do
        u ← EXTRACTMIN(Q)
        for all vertex v ∈ u.ADJACENCYLIST do
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
```

source node

starting distance: infinite for all nodes

Q is a **global priority queue** sorted by current distance

adapt distances

Source: **Data-Intensive Text Processing with MapReduce**

Single-source shortest path

In the MapReduce world: parallel BFS

Mapper: emit all distances, and the graph structure itself

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                          ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes
```

Reducer: update distances and emit the graph structure

```
class Reducer
    method Reduce(nid m, [d_1, d_2, ...])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d_1, d_2, ...] do
            if IsNode(d) then
                M ← d                           ▷ Recover graph structure
            else if d < d_min then              ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                      ▷ Update shortest distance
        Emit(nid m, node M)
```

Source: **Data-Intensive Text Processing with MapReduce**

Edges have unit weight.

PageRank

Page et al., 1998

- **Idea**: if page `px` links to page `py`, then the creator of `px` implicitly transfers some importance to page `py`
  - `yahoo.com` is an important page, many pages point to it
  - Pages linked to from `yahoo.com` are also likely to be important
- A page **distributes** "importance" through its outlinks
- Simple PageRank (iteratively):

out-degree of node u

$$PageRank_{i+1}(v) = \sum_{u \to v} \frac{PageRank_i(u)}{N_u}$$

all nodes linking to v

Graph processing notes

not discussed last time

- In **dense graphs**, MR running time would be dominated by the shuffling of the intermediate data across the network
  - **Worst case**: $O(n^2)$
  - **Impractical** for MR (commodity hardware)
- Often, combiners and in-mapper combining patterns can be used to speed up the process
- **Data localization** can be **difficult**
  - Combiners are only useful if there is something to aggregate (e.g. for PR several nodes pointing to the same target in a single MAPPER)
  - Heuristics: e.g. pages from the same domain to the same MAPPER

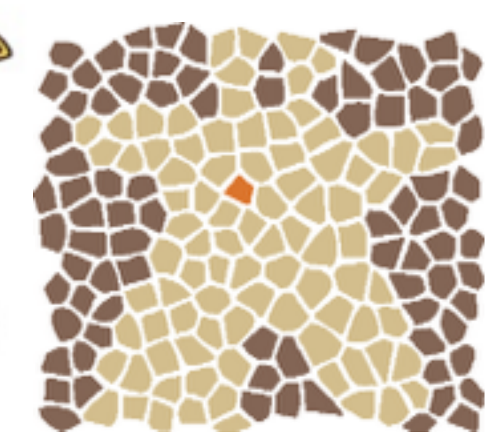
not discussed last time

Graph processing in Hadoop

- Disadvantage: iterative algorithms are slow
  - Lots of reading/writing to and from disk

- Advantage: no additional libraries needed

- Enter **Pregel** / **Giraph**:
  - Specifically created for iterative graph computations
  - More details in this lecture



Now: Issues and Solutions

Efficient large-scale graph processing is challenging

- **Poor locality** of memory access
- **Little work** per node (vertex)
- **Changing degree of parallelism** over the course of execution
- Distribution over many commodity machines due to poor locality is **error-prone** (failure likely)
- Needed: **"*scalable general-purpose system for implementing arbitrary graph algorithms [in batch mode] over arbitrary graph representations in a large-scale distributed environment*"**

Processing large graphs: existing options (until 2010)

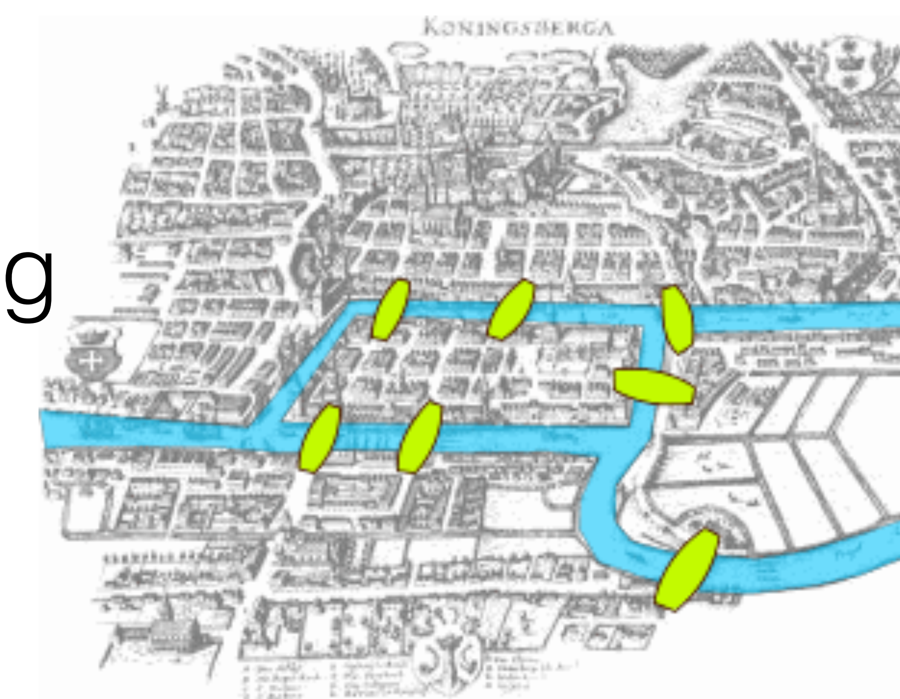
- **Custom distributed infrastructure**
  - Problem: each algorithm requires new implementation effort
- Relying on the **MapReduce framework**
  - Problem: performance and usability issues
  - Remember: the whole graph is read/written in every job
- **Single-processor graph algorithm library** (e.g. LEDA)
  - Problem: does not scale
- **Existing parallel graph systems**
  - Problem: do not address fault tolerance & related issues appearing in large distributed setups

Enter Pregel (2010)

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn,
Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

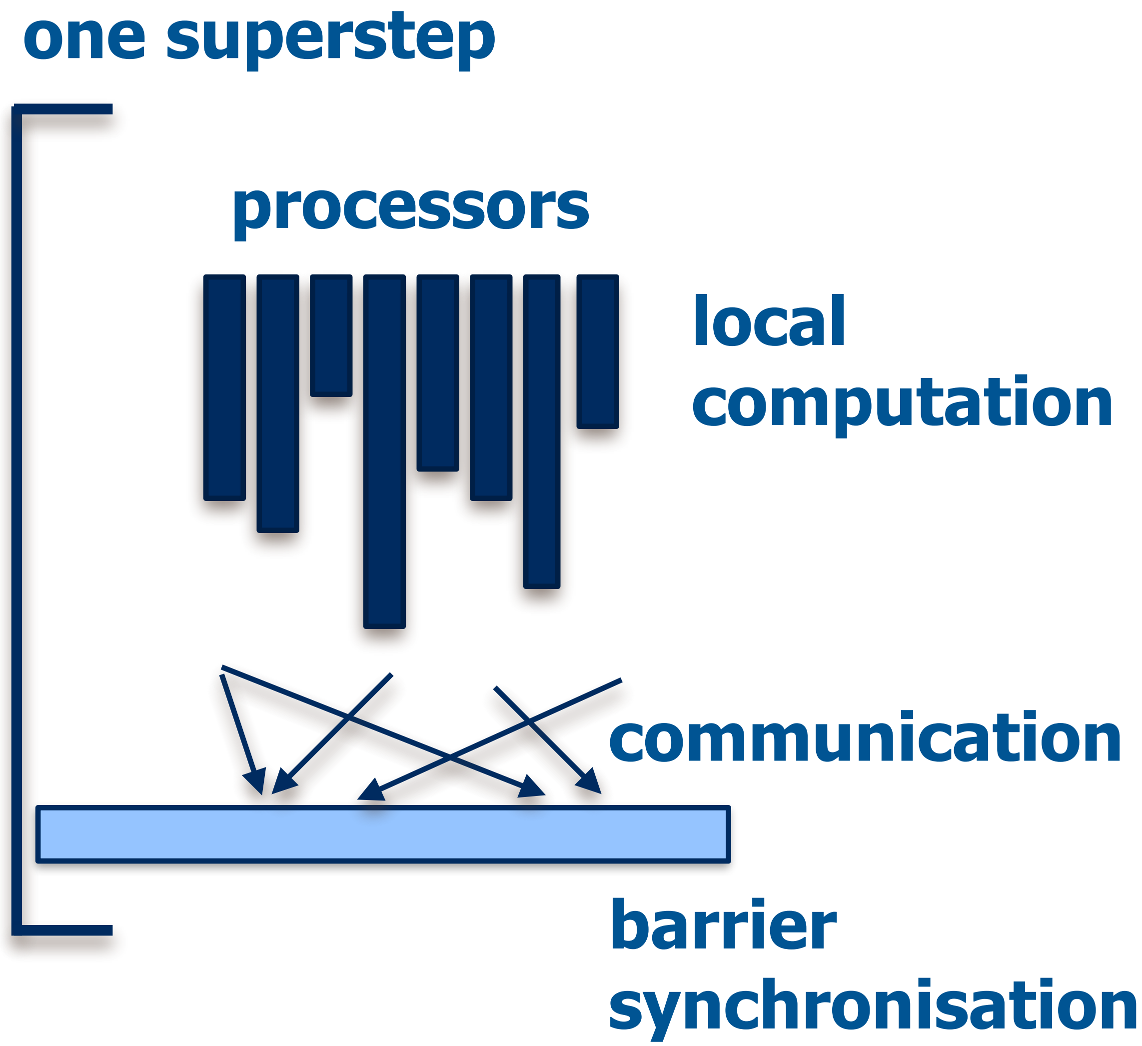
- “We built a scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms”
- Pregel river runs through Königsberg (Euler’s seven bridges problem)



A bit of theory: BSP

Bulk Synchronous Parallel (BSP)

- General model for the design of parallel algorithms
- Developed by Leslie Valiant in the 1980s/90s
- BSP computer: processors with fast local memory are connected by a communication network
- BSP computation is a series of "**supersteps**"



- No message passing in MR
- Avoids MR's costly disk and network operations

Bulk Synchronous Parallel (BSP)

- **Supersteps** consist of **three phases**

Local computation: every processor performs computations using data stored in local memory - independent of what happens at other processors; a processor can contain several processes (threads)

Communication: exchange of data between processes (put and get); one-sided communication

Barrier synchronisation: all processes wait until everyone has finished the communication step

- Local computation and communication phases are **not** strictly ordered in time

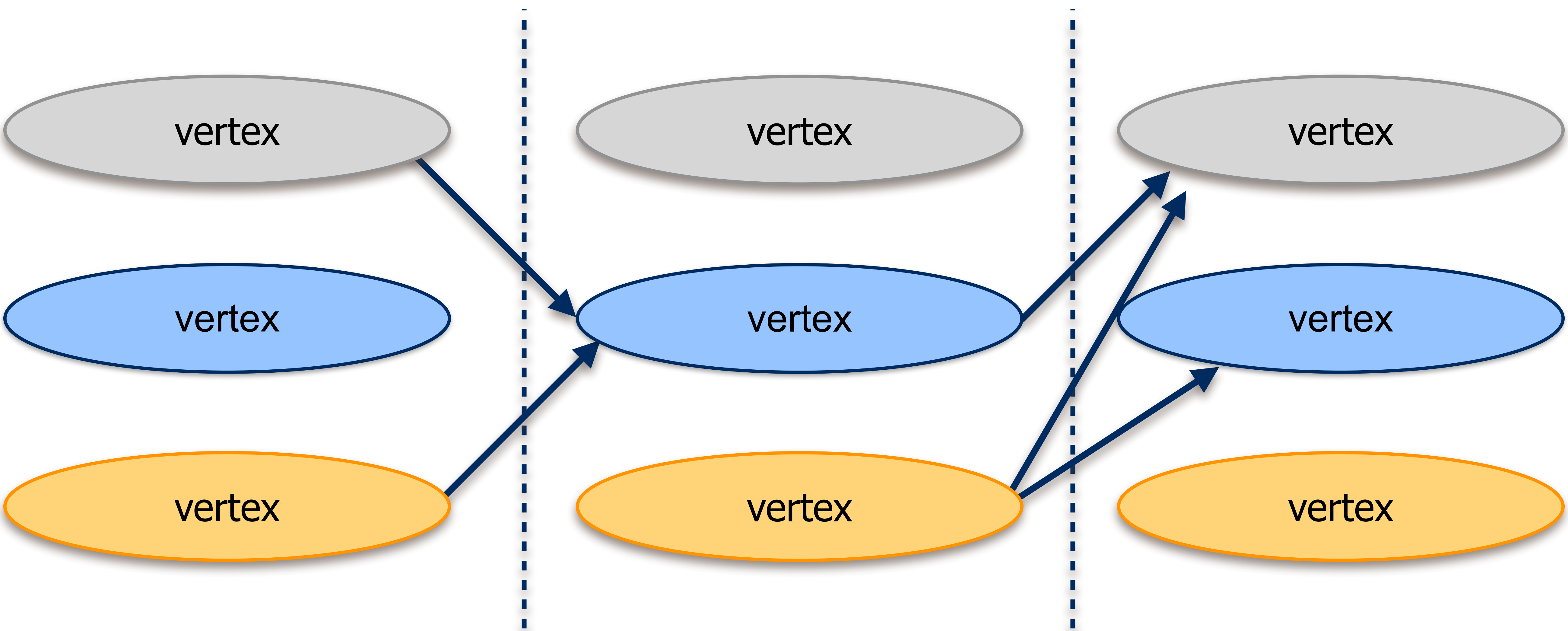
Bulk Synchronous Parallel (BSP)

BSP & graphs: "**Think like a vertex!**"

In BSP, algorithms are implemented from the viewpoint of a **single vertex** in the input graph performing a **single iteration** of the computation.

Think like a vertex

Each vertex has an **id**, a **value**, a **list of adjacent neighbour ids** and corresponding **edge values**.



Pregel

A high-level view

- Pregel computations consist of a **sequence of iterations** (supersteps)
- In a superstep, the framework invokes a **user-defined function for each vertex** (conceptually in parallel)
- Function specifies **behaviour at a single vertex** V and a single superstep S
  - it can **read messages** sent to V in superstep $(S\text{-}1)$
  - it can **send messages** to other vertices that will be read in superstep $(S+1)$
  - it can modify the **state** of V and **its outgoing edges**

Vertex-centric approach

- Reminiscent of MapReduce
  - User (i.e. algorithm developer) focus on a **local action**
  - Each vertex is processed **independently**
  - System composes these actions to lift computation to a large dataset

- By design: well suited for a **distributed** implementation
  - All communication is from superstep S to $(S+1)$
  - **No defined** execution **order** within a superstep
  - Free of deadlocks and data races

Pregel input

- **Directed** graph
- Each vertex is associated with a modifiable, user-defined value
- The directed edges are **associated** with their **source vertices**
- Each directed edge consists of a modifiable, user-defined value and a target vertex identifier

Edges are **not** first-class citizens in this model.

Algorithm termination

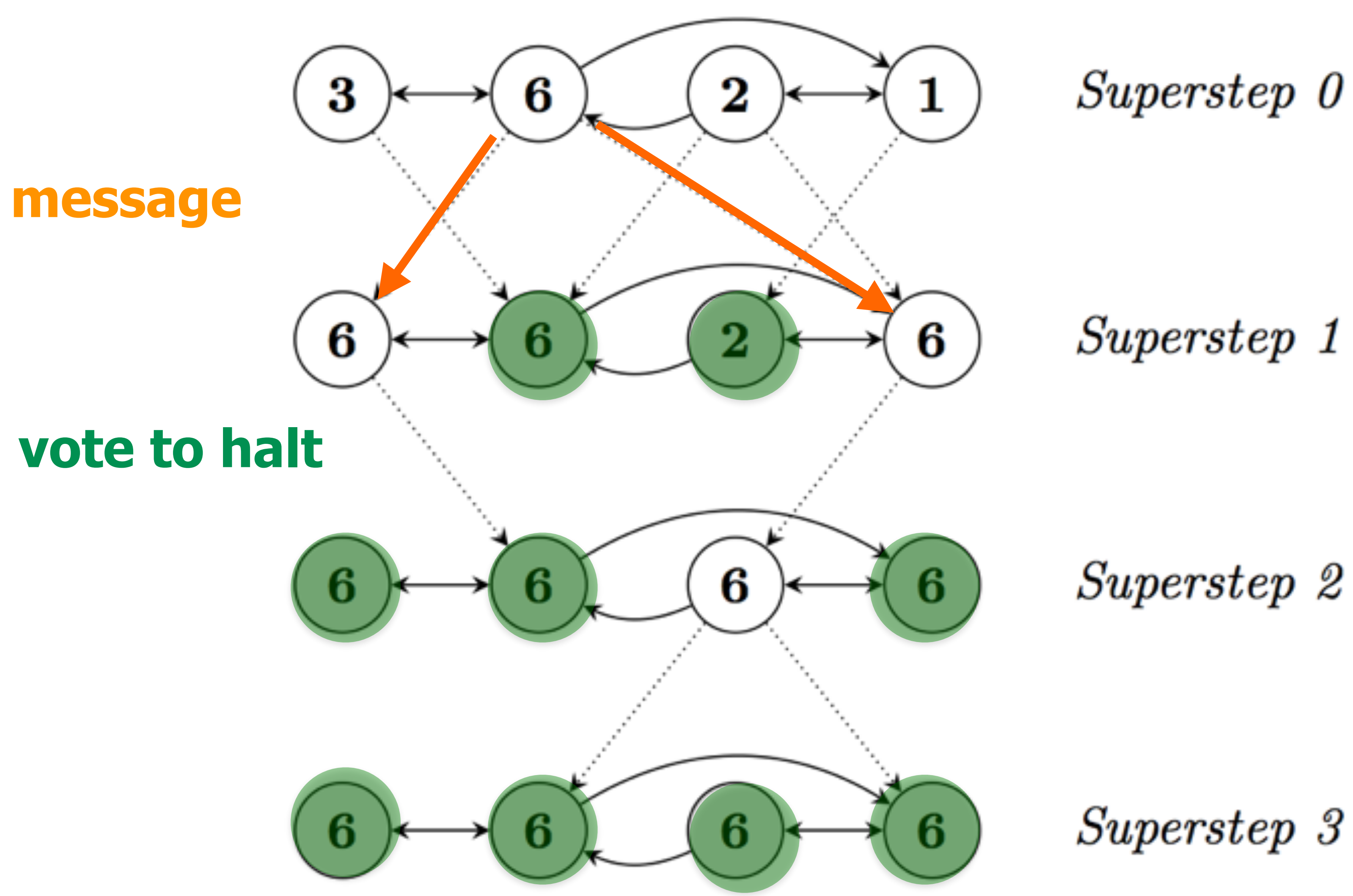
- In MapReduce: external driver program decides when to stop an iterative algorithm
- BSP-inspired Pregel:
  - Superstep 0: all vertices are active
  - All active vertices participate in the computation at each superstep
  - A vertex **deactivates itself** by voting to halt
  - No execution in subsequent supersteps
  - Vertex can be **reactivated** by receiving a message
- Termination criterion: **all vertices have voted to halt** & no more messages are in transit

Pregel's output

- A set of values output by the vertices
- Often: a directed graph *isomorphic* to the input (i.e. no change)
- Other outputs are possible as vertices/edges can be added/removed during supersteps
  - Clustering: generate a small set of disconnected vertices selected from a large graph
  - Graph mining algorithm might output aggregated statistics mined from the graph

In Hadoop anything can be emitted as output.

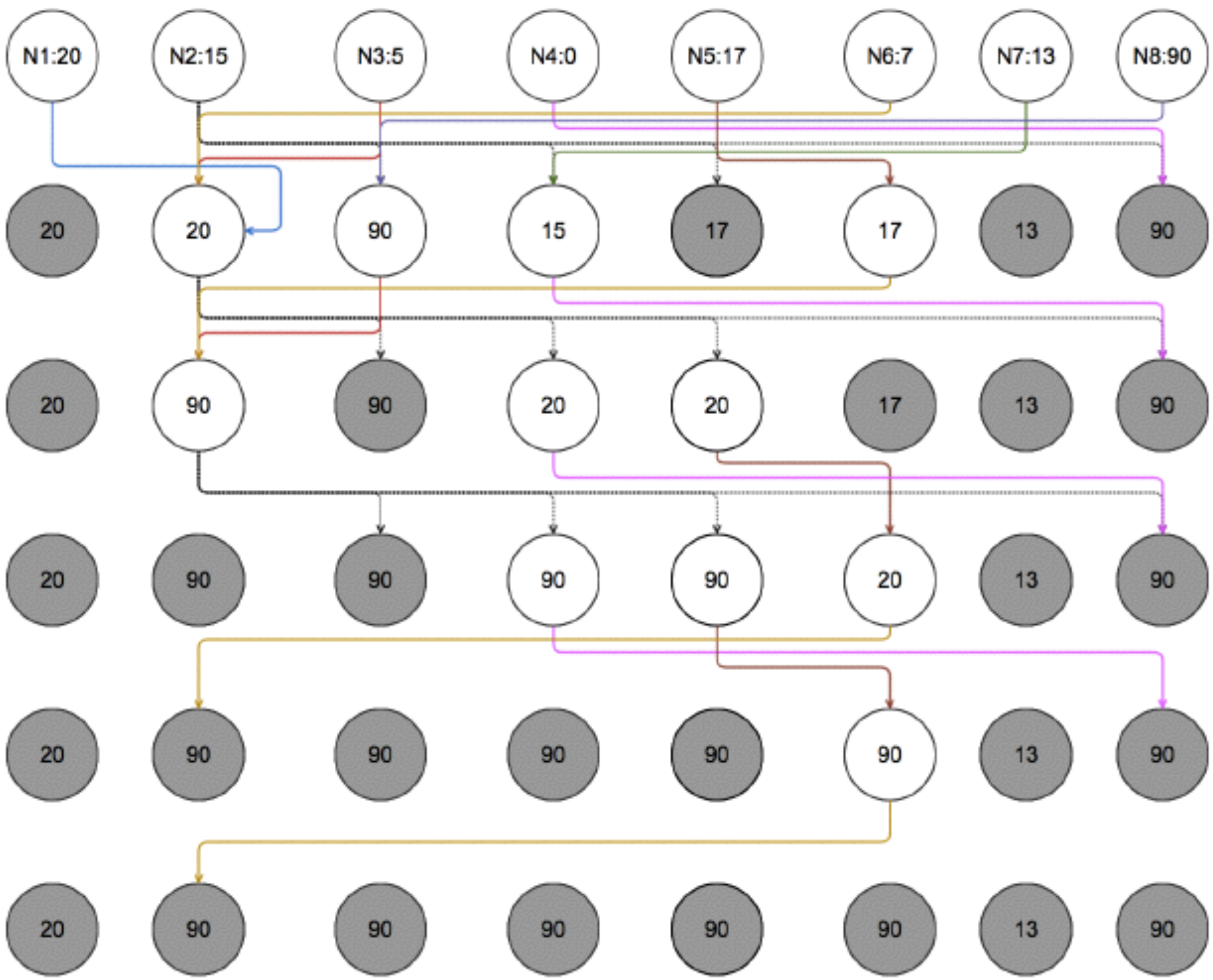
Example: maximum value

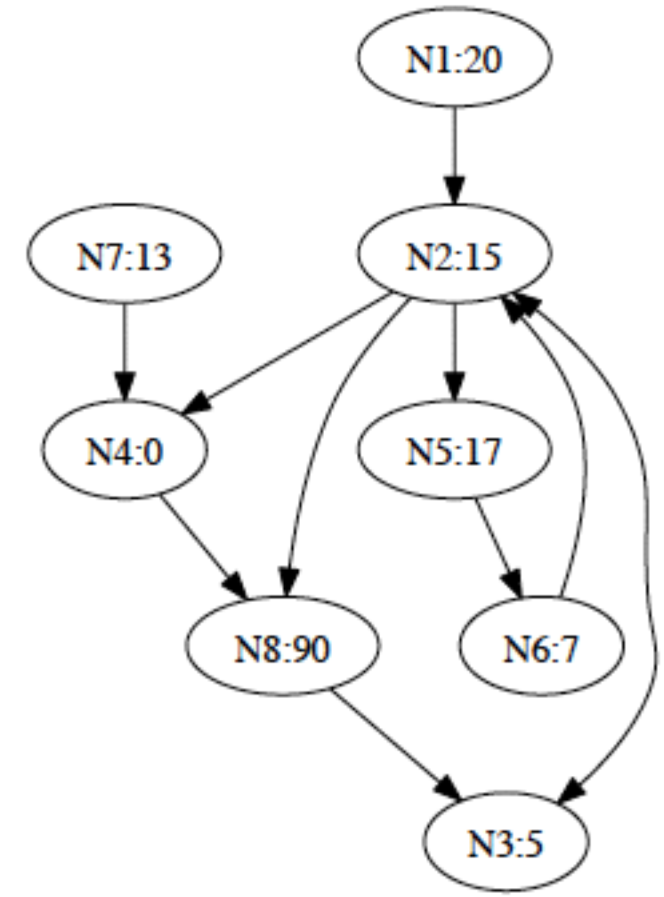


graph with four nodes and four directed edges

messages are usually send to vertices directly connected

Example II: maximum value





(one of last year's assignments)

Limiting the graph state to a single value per vertex/edge simplifies the main computation cycle, graph distribution, failure recovery.

Pregel API

- All vertices have an associated value of a particular specified type (similarly for edge and message types)
- User provides the content of a **`compute()`** method which is executed by each *active* vertex in every superstep
  - **`compute()`** can access information about the current vertex (its value), its edges, received messages sent in the previous superstep
  - **`compute()`** can change the vertex value, the edge value(s) and send new messages to be read in next superstep
- Values associated with the vertex and its edges are the only per-vertex state that persists across supersteps

Message passing

- Vertices **communicate** via messages
- Message consists of a message value and the name of the destination vertex
- Every vertex can send **any number of messages** in a superstep to any other vertex with **known** id
- All messages sent to vertex V in superstep S are available to V in superstep $S+1$
  - Messages can be PageRank scores to be distributed
  - Message to non-existing vertex can create it

Combiners

- Message sending incurs overhead
  - Especially to a vertex on a different machine

- Messages for a single vertex may be combined
  - Example: messages contain integer values & overall goal is the sum of all integers aimed at the target vertex

Aggregators

- Mechanism for **global communication**, monitoring and data

- Each vertex can provide a value to an aggregator in superstep S
  - The system combines those values using a reduction operator (e.g. min, max, sum)
  - The resulting value is made available to all vertices in superstep S+1

Aggregators

- Usage scenario: statistics
    - Sum aggregator applied to the out-degree of each vertex yields the total number of edges in the graph
    - Lost PageRank mass can be redistributed after every superstep

Aggregators

- Usage scenario B: global coordination
  - One branch of **`compute()`** can be executed in each superstep until an and aggregator determines that all vertices fulfil a particular condition, then another branch is executed
  - Min/max aggregator applied to vertex IDs can select one vertex for a distinguished role in the algorithm
- Aggregators should be **commutative** and **associative** (ordering of input does not play a role)
- *Sticky* aggregator: uses input values from all supersteps

Topology mutations

- Some graph algorithms change a graph's topology
  - Example: minimum spanning tree algorithm might remove all but the tree edges
- Requests to add/remove vertices and edges are issued within `compute()`
- Multiple vertices may issue conflicting requests in the same superstep
  - Resolved through simple ordering rules

Graph partitioning

- MapReduce framework: entire graph is read/written in each iteration
- In Pregel:
  - Graph is divided into partitions, each consisting of a set of vertices and all those vertices outgoing edges
  - Assignment of a vertex to a partition depends on the vertex ID

Fault tolerance

- Achieved through **checkpointing**
- At the beginning of some supersteps the master instructs the workers to save the state of their partitions to persistent storage
- Worker failure detected through ping messages the master issues to workers
- If a worker is corrupt, the master reassigns graph partitions to the workers being alive; they reload their partition state from the most recently available checkpoint

Worker implementation

- Each worker maintains the state of its portion of the graph **in memory**
  - Map from vertexID to the state of each vertex: current value, list of outgoing edges, a queue of incoming messages, flag [active/inactive]
- In a superstep, a worker loops through all its vertices
- Messages:
  - Destination vertex on a different worker: messages are buffered for delivery; sent as single network message
  - Destination vertex on the same worker: message is placed directly into the incoming message queue

Master implementation

- Master is responsible for coordinating the worker activities
- Each worker has a unique id
- Master maintains list of workers currently alive
  - Worker id, addressing information, portion of the graph assigned
  - Size of this data structure proportional to the number of partitions, not the number of vertices/ edges (thus, large graphs can be stored)

Examples

PageRank in 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();
    }
  }
};
```

vertex type: double
message type: double
edge value: void

superstep 0:
initialisation
with PR=1/|G|

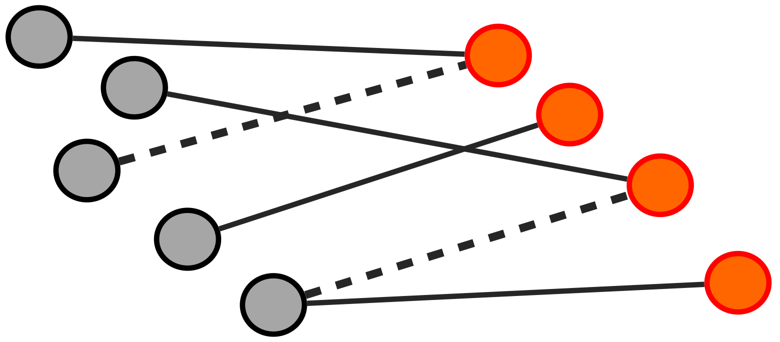
Single-source shortest paths in Pregel

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

superstep 0: initialisation with INF

Bipartite matching in Pregel

- **Input**: two distinct sets of vertices with only edges between them
- **Output**: subset of edges with no common endpoints
- **Maximal matching**: no more edges can be added without violating the no-common-endpoints condition
- Vertex values: tuple of Left/Right flag (is the vertex a “left” or “right” one) and name of matched vertex once known



Bipartite matching in Pregel

Randomized maximal matching

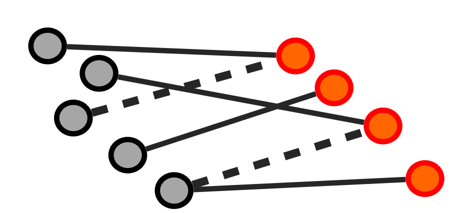
1. Each *left* vertex not yet matched sends a **message** to each neighbour to request a match; vote to halt

2. Each *right* vertex not yet matched **randomly** chooses one of the messages it receives, grants the request and **informs all requesters** about decision; vote to halt

3. Each *left* vertex not yet matched chooses one of the grants it received and sends acceptance back

4. Unmatched *right* vertex receives at most one acceptance message; votes to halt

a 3-way handshake



Some experimental results of Pregel

- Single-source shortest path on a *binary* tree with **1 billion vertices**
  - 50 worker tasks: 174 seconds
  - 800 worker tasks: 17 seconds

- Single-source shortest path on a *binary* tree with 800 worker tasks
  - 1 billion vertices: 17 seconds
  - **50 billion vertices**: 700 seconds

- Single-source shortest path on a **random graph** with mean out degree **127**, 800 worker tasks
  - 1 billion vertices (127 billion edges): ~10 minutes

300 multi-core commodity PCs

Giraph

Pregel is not open source source but Giraph is

- **Giraph**: a loose open-source implementation of Pregel

- Employs **Hadoop's MAP phase** to run computations

- Employs Zookeeper (service that provides distributed synchronisation) to enforce barrier waits

- Active contributions from Twitter, Facebook, LinkedIn and HortonWorks

- Differences to Pregel: edge-oriented input, out-of-core computations, master computation…

Giraph

- Hadoop Mappers are used to host Giraph Master and Worker tasks
  - No Reducers (no shuffle/sort phase)
- **Input graph is loaded just once**, data locality is exploited when possible
  - Graph partitioning by default according to hash(vertexID)
- The computations on data are performed **in memory,** with very few disk spills
- Only **messages are passed through the network** (not the entire graph structure)

Giraph in action: maximum value in a graph

Remember: Think like a vertex!

```java
package org.apache.giraph.examples;

public class MaxComputation extends BasicComputation<IntWritable, IntWritable,
NullWritable, IntWritable> {

 @Override
 public void compute(Vertex<IntWritable, IntWritable, NullWritable> vertex,
                     Iterable<IntWritable> messages) throws IOException {

   boolean changed = false;
   for (IntWritable message : messages) {
     if (vertex.getValue().get() < message.get()) {
       vertex.setValue(message);
       changed = true;
     }
   }
   if (getSuperstep() == 0 || changed) {
     sendMessageToAllEdges(vertex, vertex.getValue());
   }
   vertex.voteToHalt();
 }
}
```

vertex id, vertex data
edge data, message type

process messages
from previous superstep

maximum changes

at start or after change,
message connected vertices

reactivation only
after incoming message

Summary

- Reminder of MapReduce-based graph algorithm implementations
- Pregel
- BSP
- Giraph
- Examples of implemented graph algorithms

References

- Malewicz, Grzegorz, et al. **"Pregel: a system for large-scale graph processing."** Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.
- Apache Giraph: http://giraph.apache.org/
- Giraph example code: http://bit.ly/1bSohxy

THE END