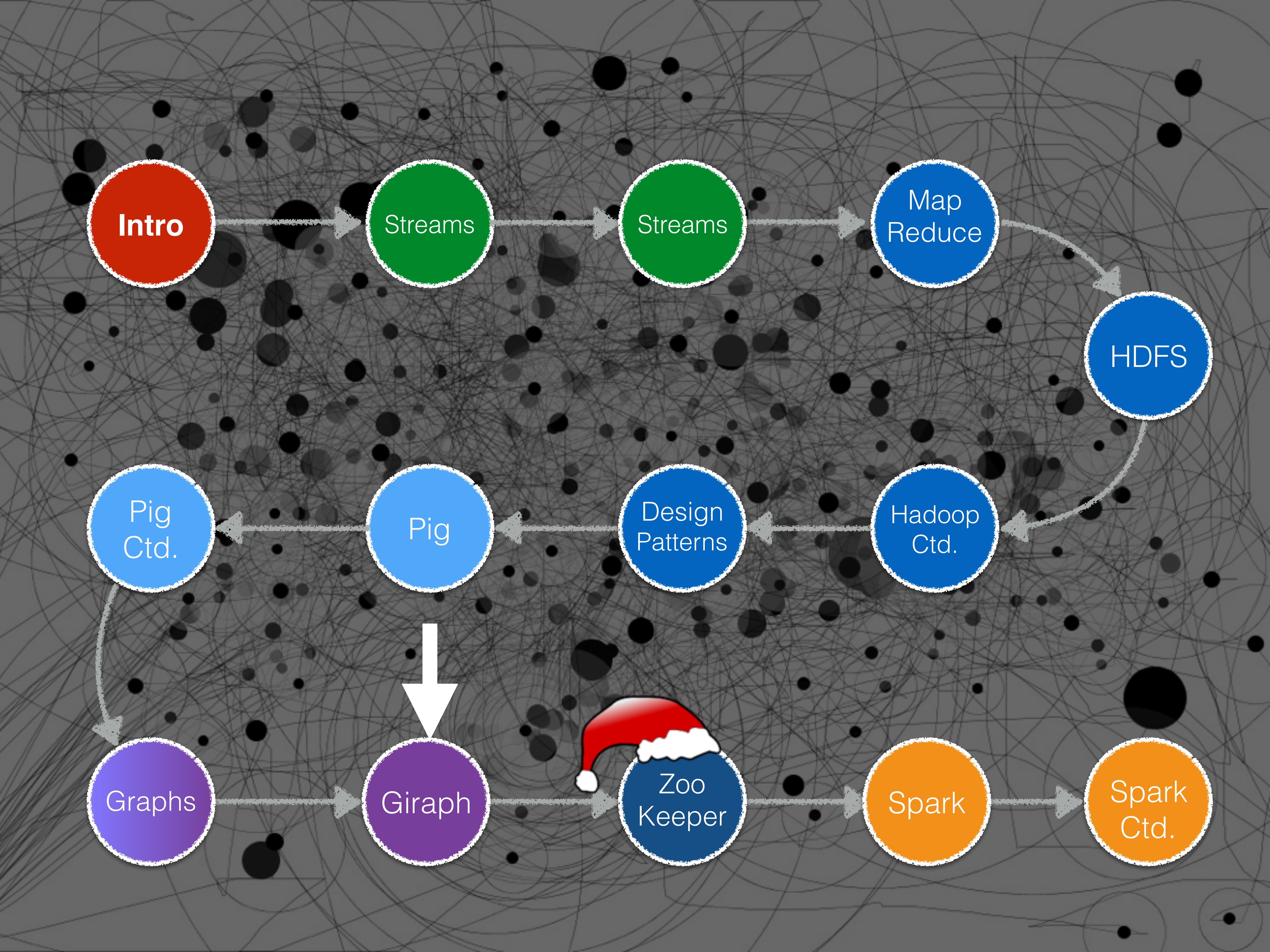
TI2736-B

Big Data Processing

Claudia Hauff

ti2736b-ewi@tudelft.nl

TU Delft


Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Ctd.
Design Patterns
Pig
Pig Ctd.
Graphs
Giraph
Zoo Keeper
Spark
Spark Ctd.

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

A little reminder

PageRank

Page et al., 1998

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

out-degree of node u

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

all nodes linking to v

PageRank in MapReduce

Pseudocode: simplified PageRank

```
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)
```


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

Efficient large-scale graph processing is challenging

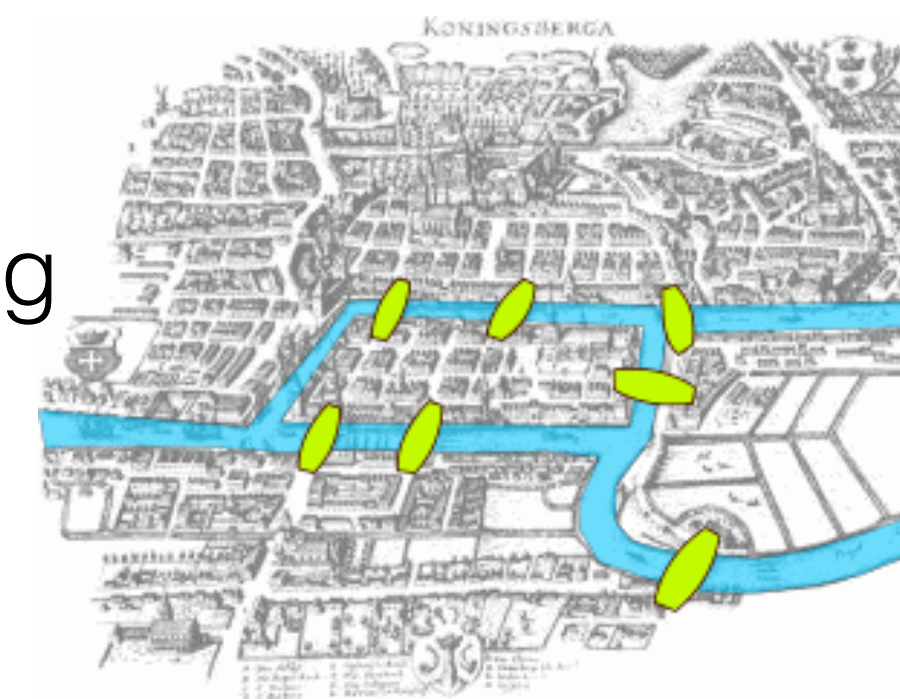
- **Poor locality** of memory access
- **Little work** per node (vertex)
- **Changing degree of parallelism** over the course of execution
- Distribution over many commodity machines due to poor locality is **error-prone** (failure likely)
- Needed: "***scalable general-purpose system for implementing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

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)

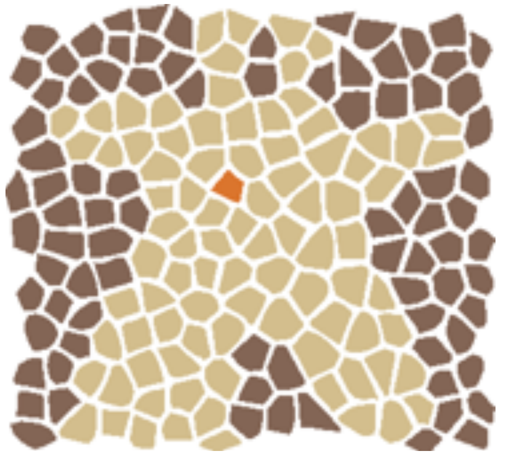


Graph processing in Hadoop

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

- **Advantage**: no additional libraries needed

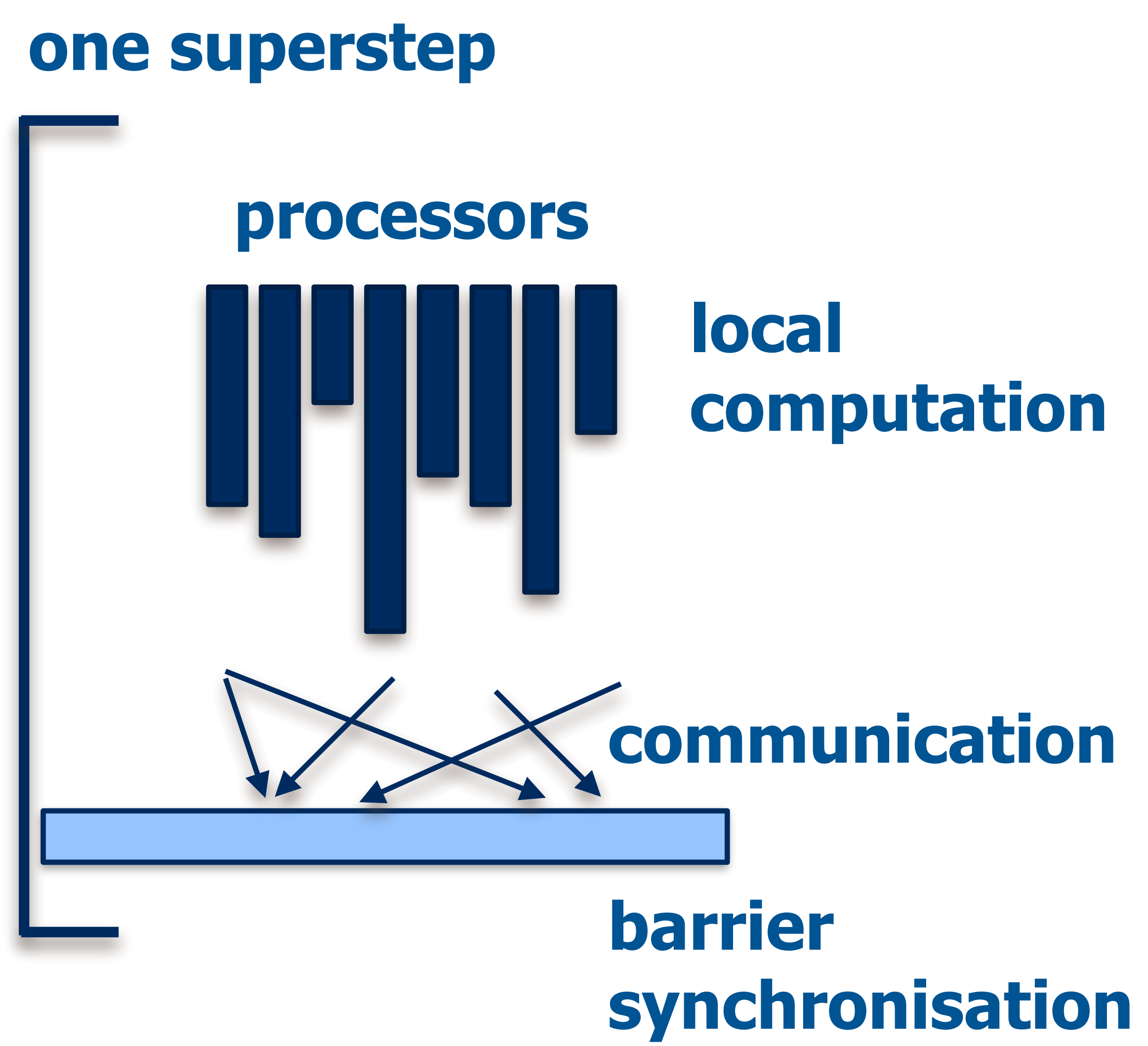
- Enter **Giraph**: an open-source implementation of yet another Google framework (Pregel)
    - Specifically created for iterative graph computations



A bit of theory:
Bulk Synchronous Parallel
or BSP

Bulk Synchronous Parallel

- 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

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

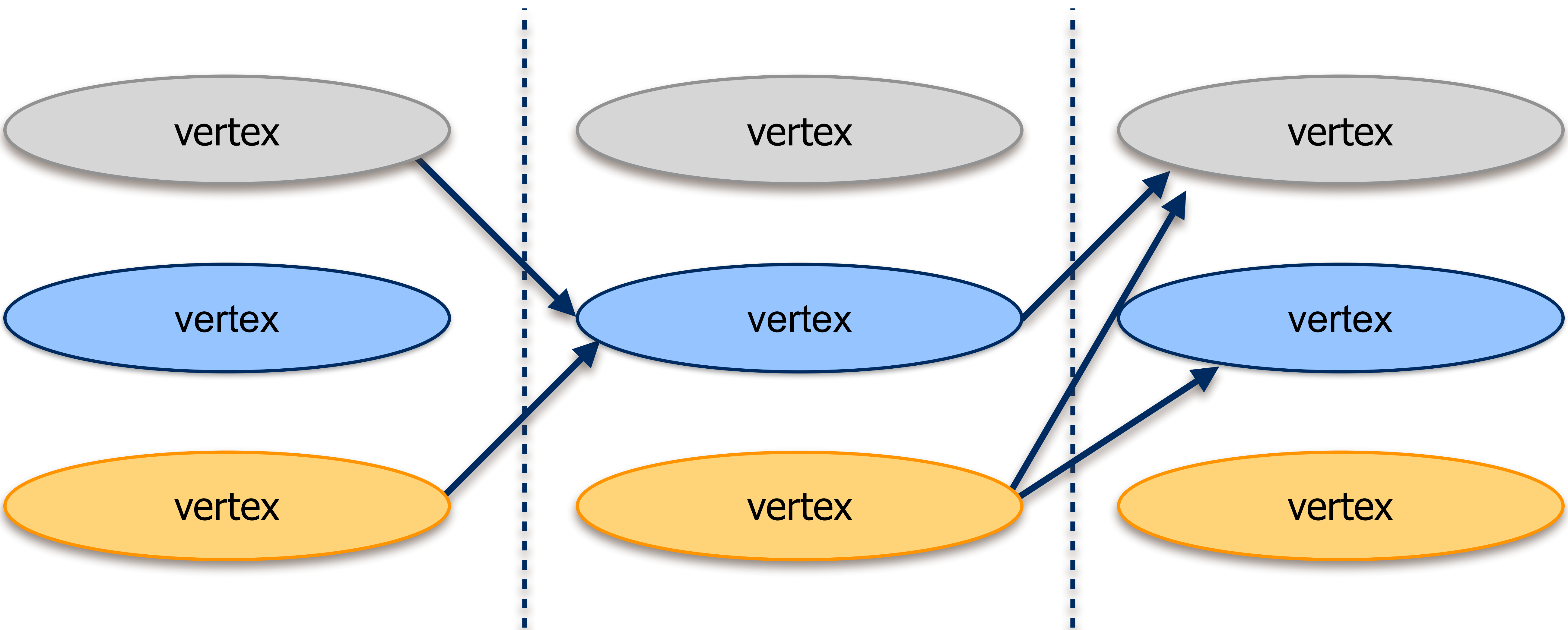
Bulk Synchronous Parallel

BSP & 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**
- 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

"We have not found any graph algorithms for which message passing is not sufficient"

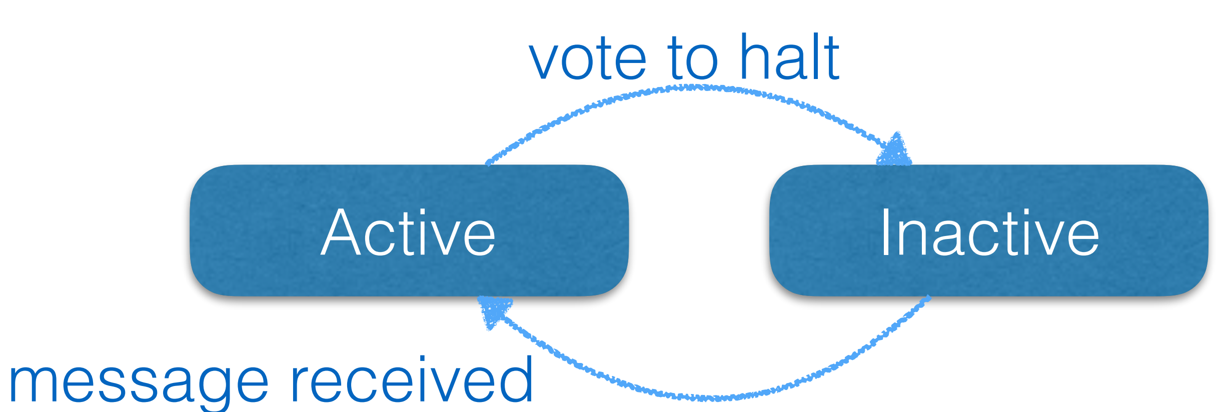
Pregel input

- **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

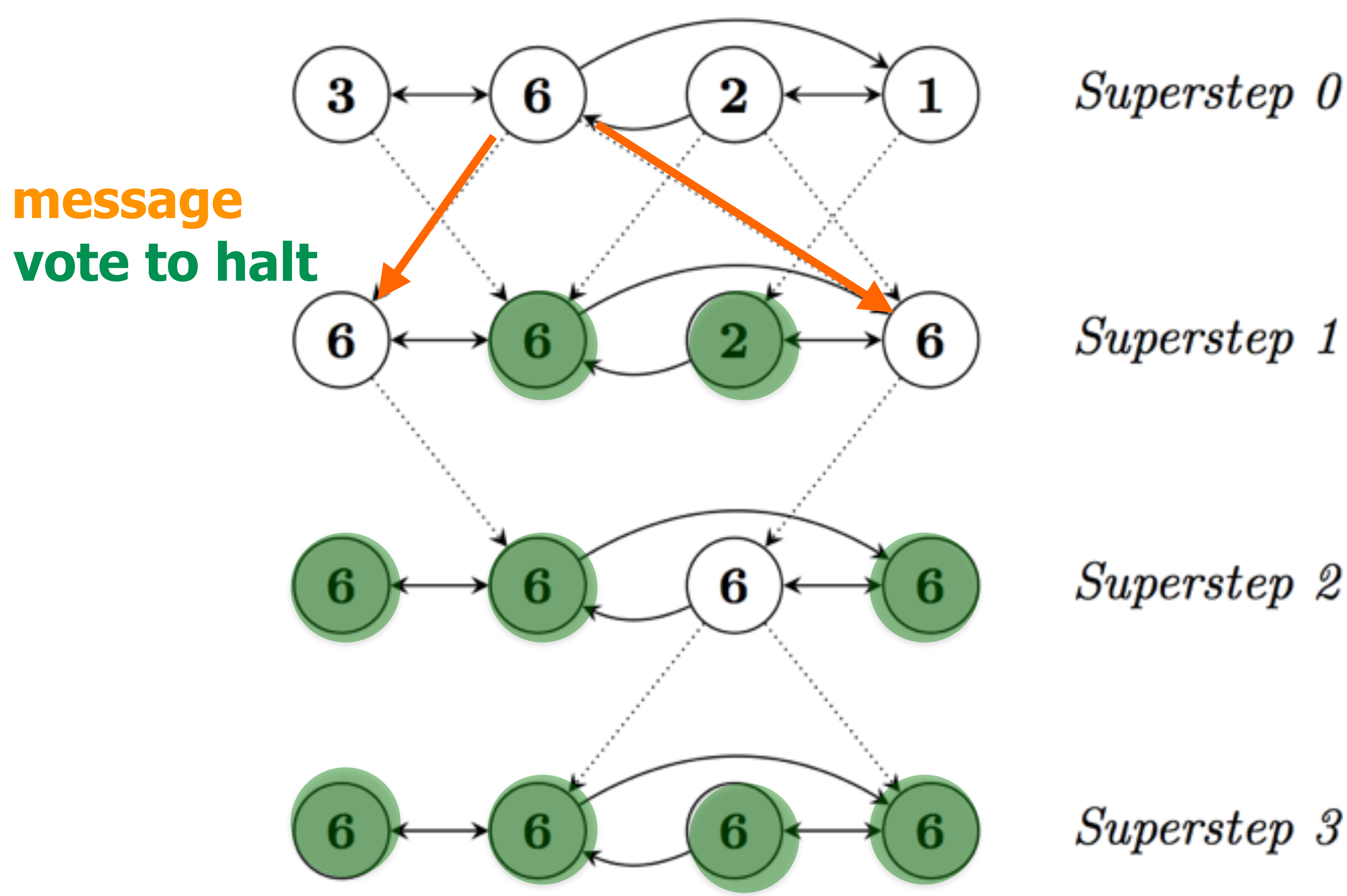
- In MapReduce: external driver program decides when to stop an iterative algorithm
- BSP-inspired Pregel:
  - Superstep 0: all vertices are active
  - All active vertices participate in the computation at each superstep
  - A vertex **deactivates itself** by voting to halt
  - No execution in subsequent 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

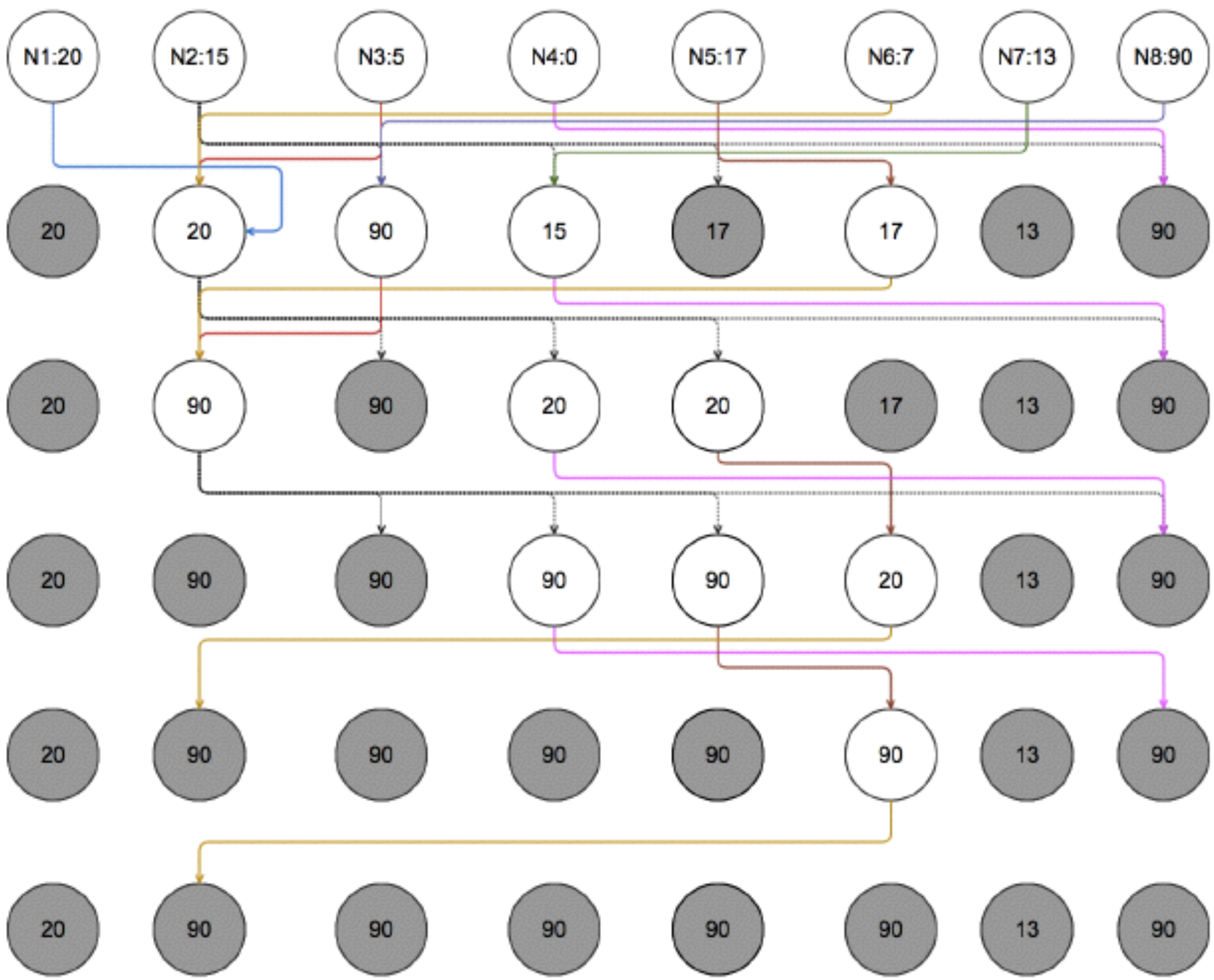
Example: maximum value

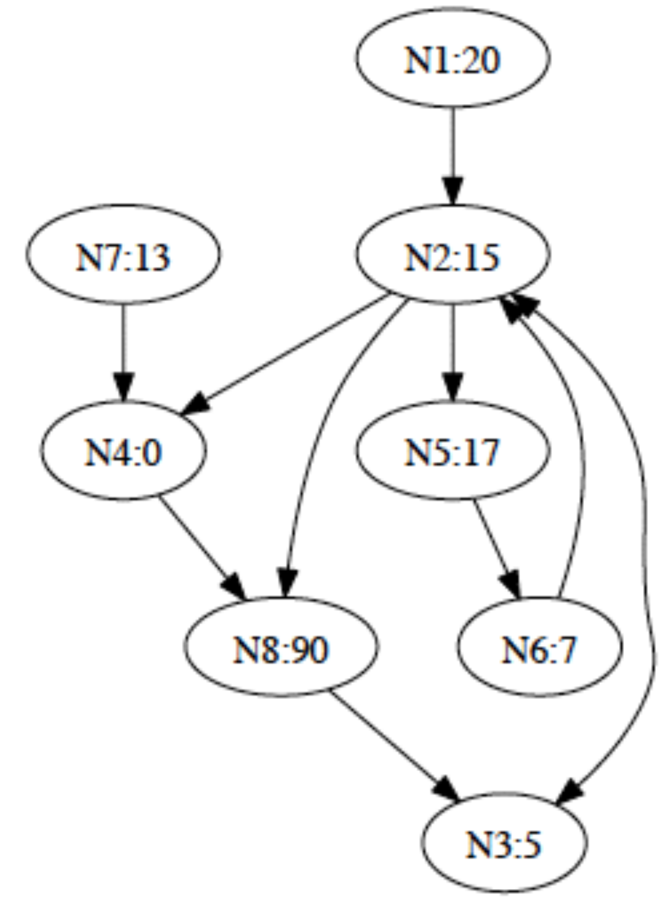


graph with four nodes and four directed edges

messages are usually send to vertices directly connected

Example II: maximum value





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 **every *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

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)

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

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: 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

- 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

Pregel Examples

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();
    }
  }
};
```

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

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

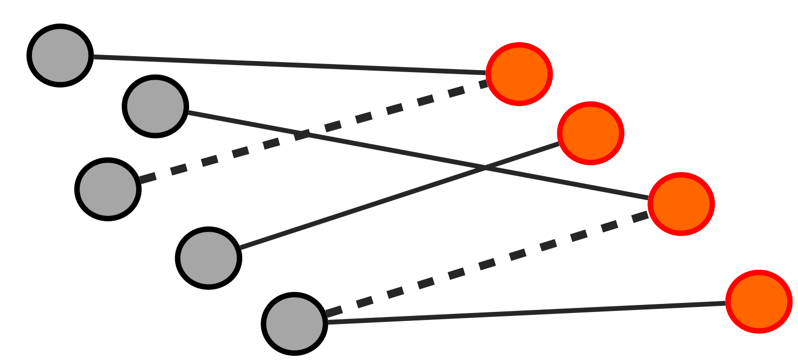
Single-source shortest paths

```
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

- **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

Randomized maximal matching

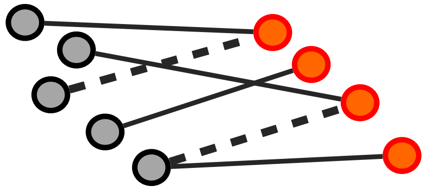
1. Each *left* vertex not yet matched sends a **message** to each neighbour to request a match; vote to halt

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 **randomly** chooses one of the grants it received and sends acceptance back

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

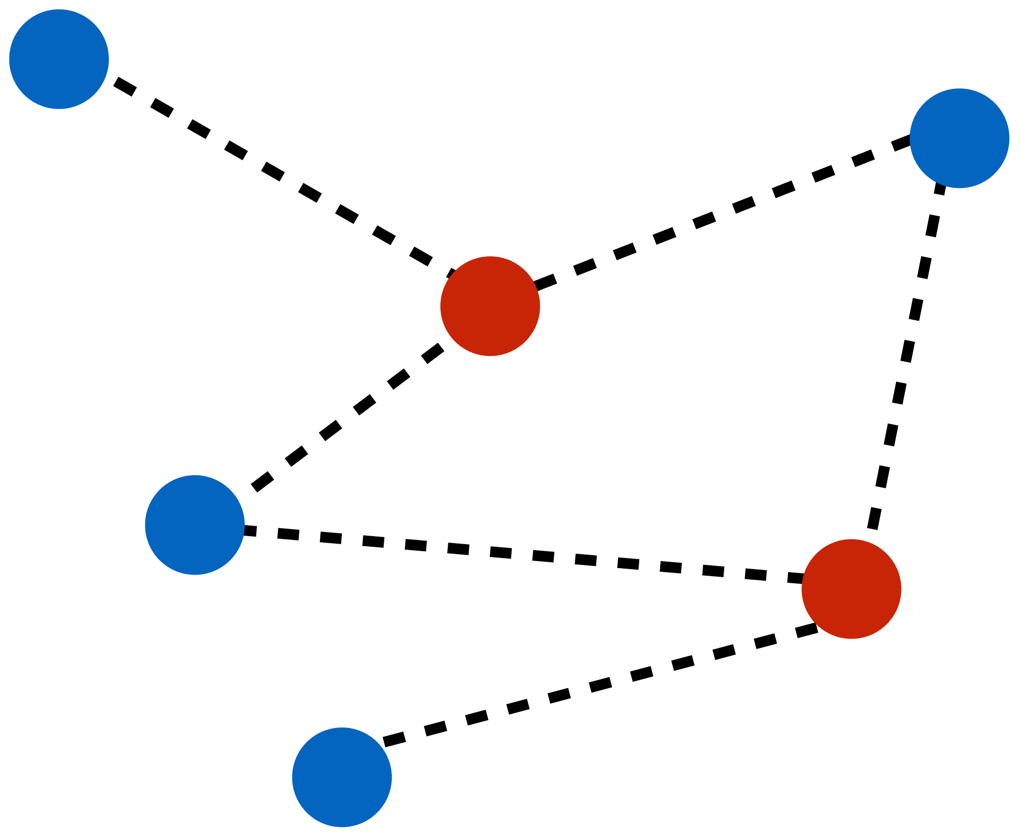
cycles of 4 phases



a 3-way handshake

Bipartite matching

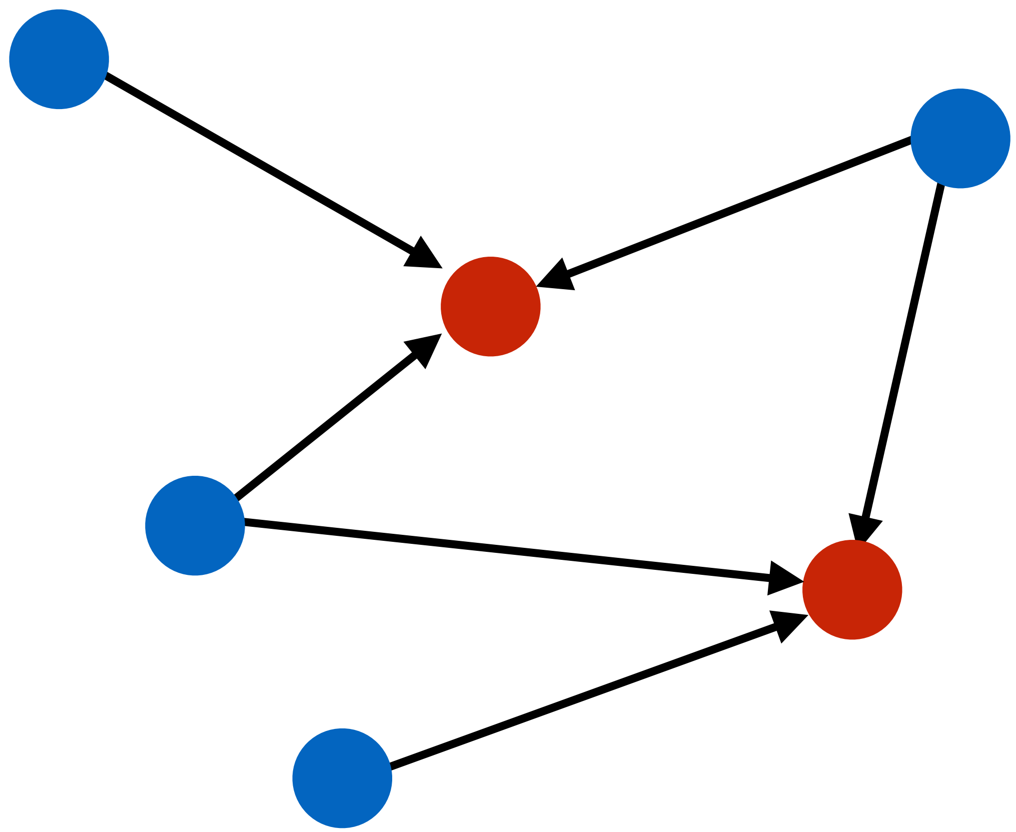
Randomized maximal matching



(blue, red)

Bipartite matching

Randomized maximal matching

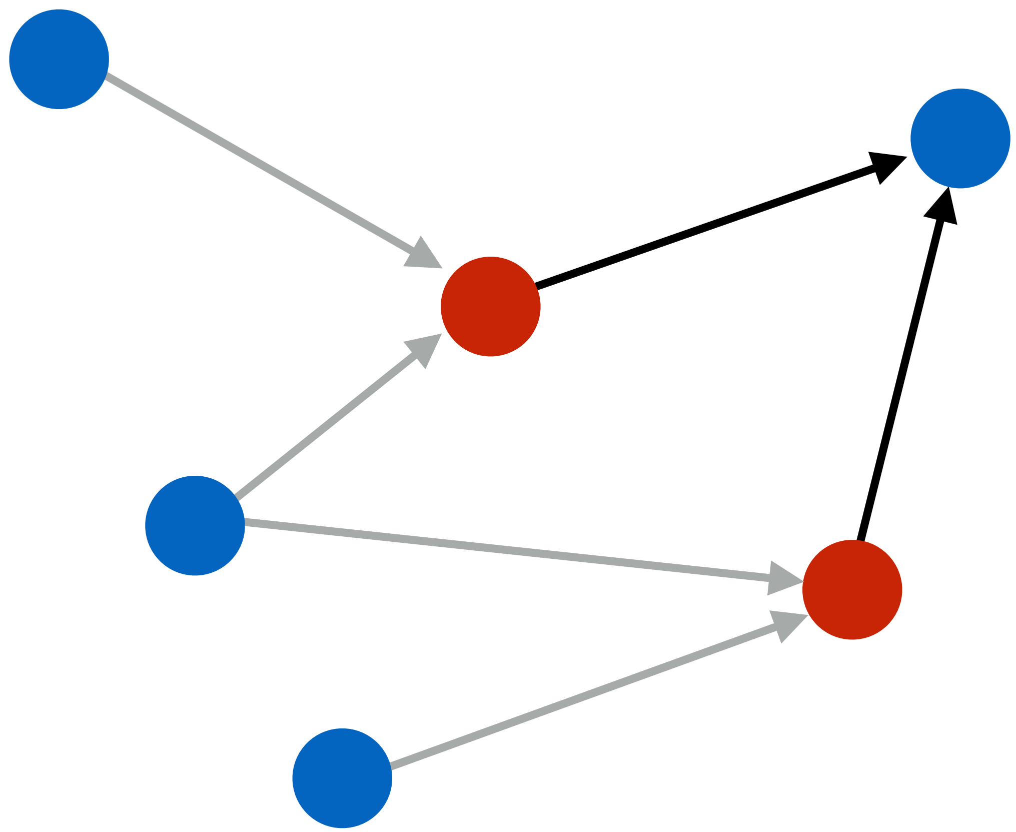


(blue, red)

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

Bipartite matching

Randomized maximal matching

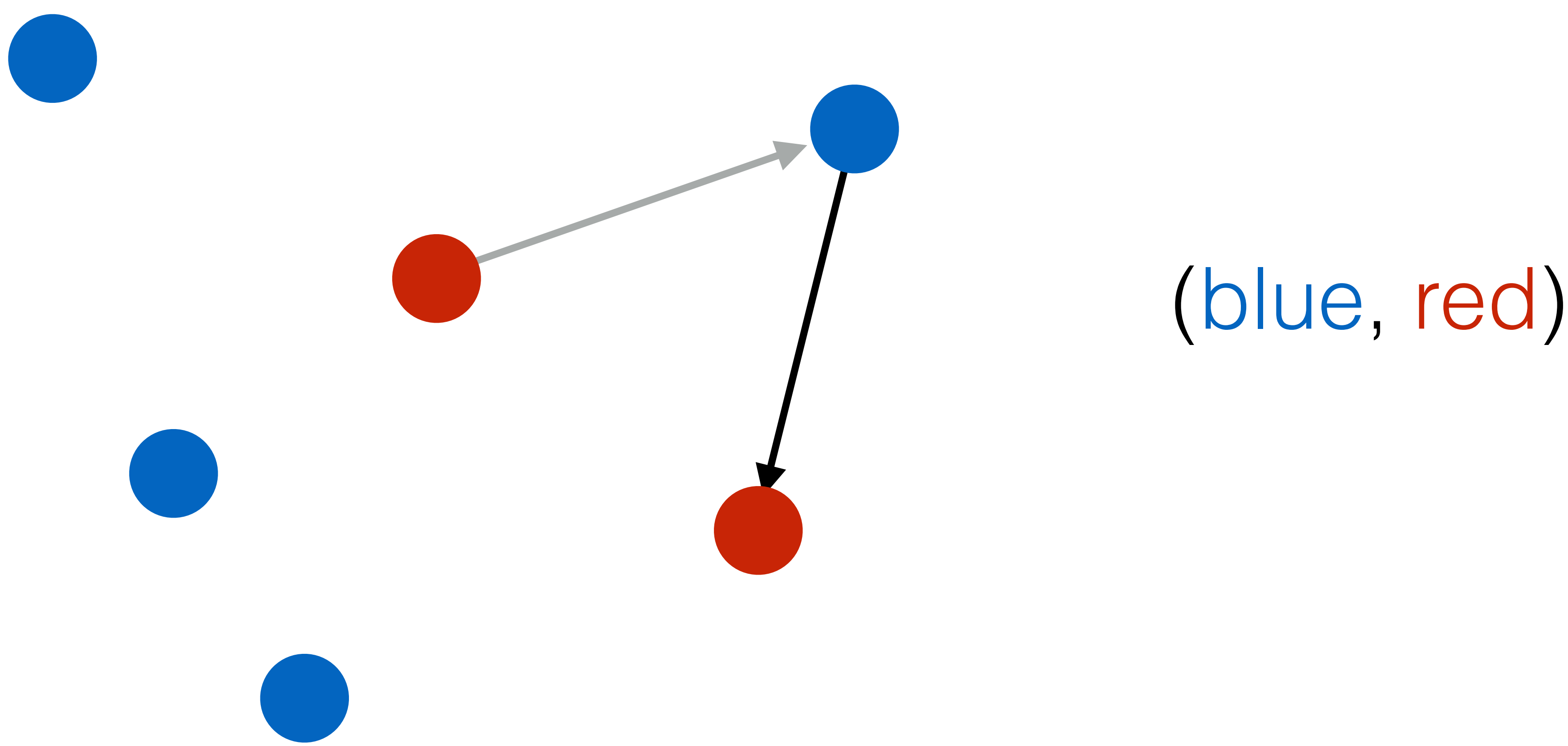


(blue, red)

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

Bipartite matching

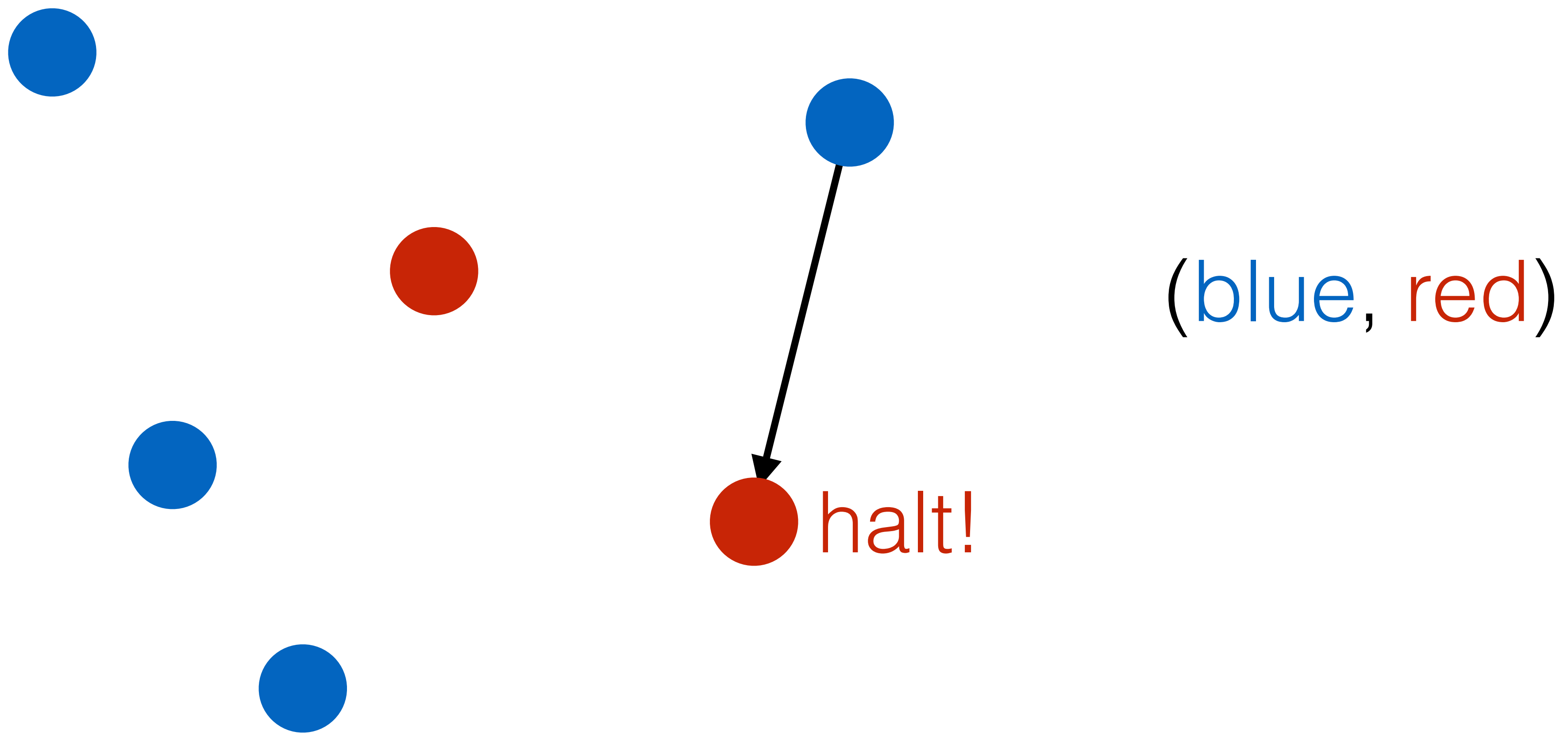
Randomized maximal matching



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

Bipartite matching

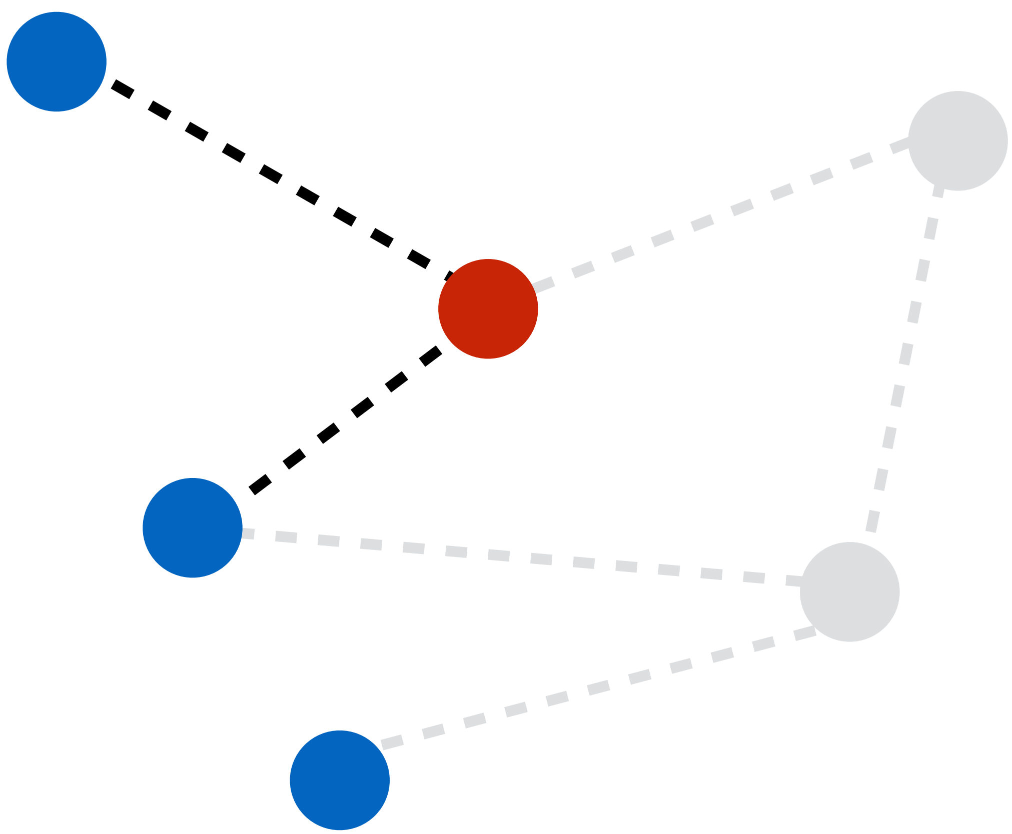
Randomized maximal matching



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

Bipartite matching

Randomized maximal matching



(blue, red)

another cycle begins …

Soft clustering

- Cluster in social graphs: a group of people that interact frequently with each other and less frequently with others
  - A person may can belong to more than one cluster

- **Input**: weighted, undirected graph

- **Output**: C_{max} clusters each with at most V_{max} vertices

- Also called “semi-clustering”

Soft clustering

Cluster score

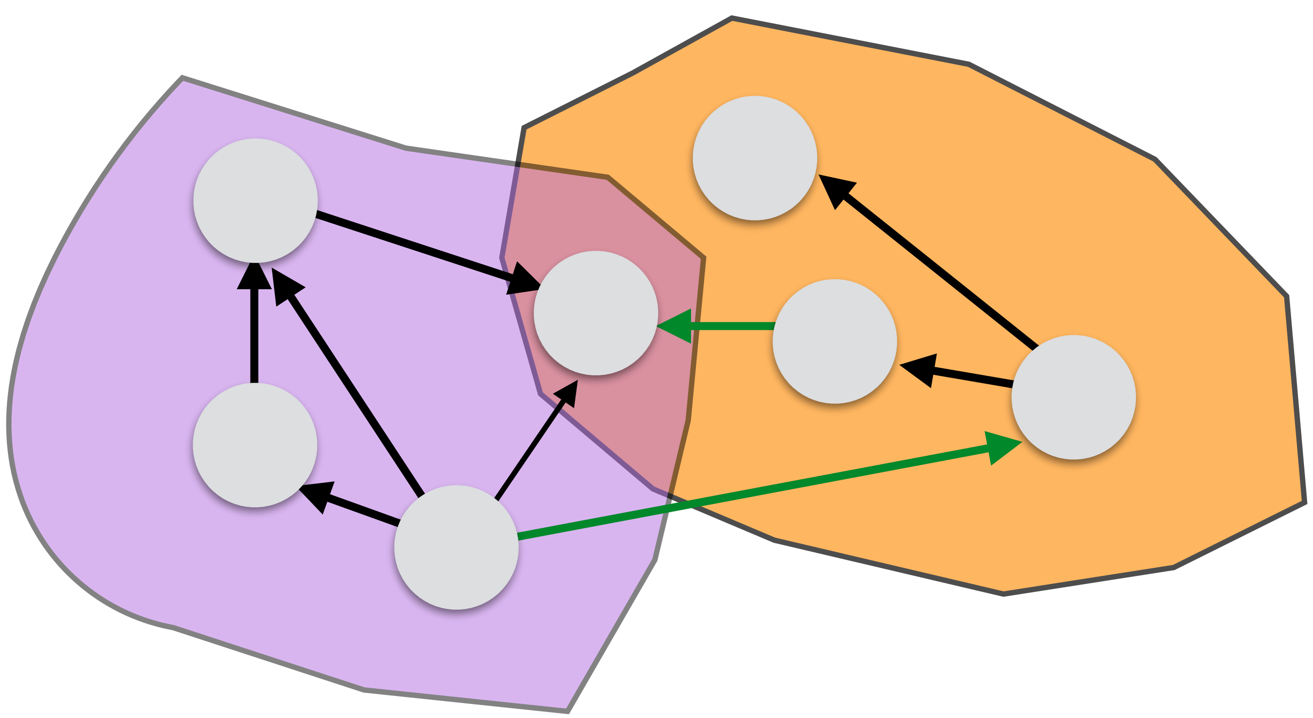
sum of weights of internal edges

user-specified param in [0,1]

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

#vertices in semi-cluster

sum of weights of boundary edges



Soft clustering

- Each vertex V maintains **a list** of at most C_{max} semi-clusters, sorted by score
- **Superstep 0**: V enters itself in the list as semi-cluster of size 1 and score 1; V publishes itself to all direct neighbours
- **Supersteps *S=1 … [until no more changes]***:
  - V iterates over the semi-clusters $c_1..c_k$ **sent to it** at S-1
  - If a semi-cluster c does not already contain V and its size is below the maximum, add V to form d
  - Semi-clusters $c_1..c_k$, $d_1..d_k$ are sorted by their cluster scores and **the best ones** are sent to V 's neighbours
  - V updates its semi-cluster list with those from $c_1..c_k$, $d_1..d_k$ that contain V

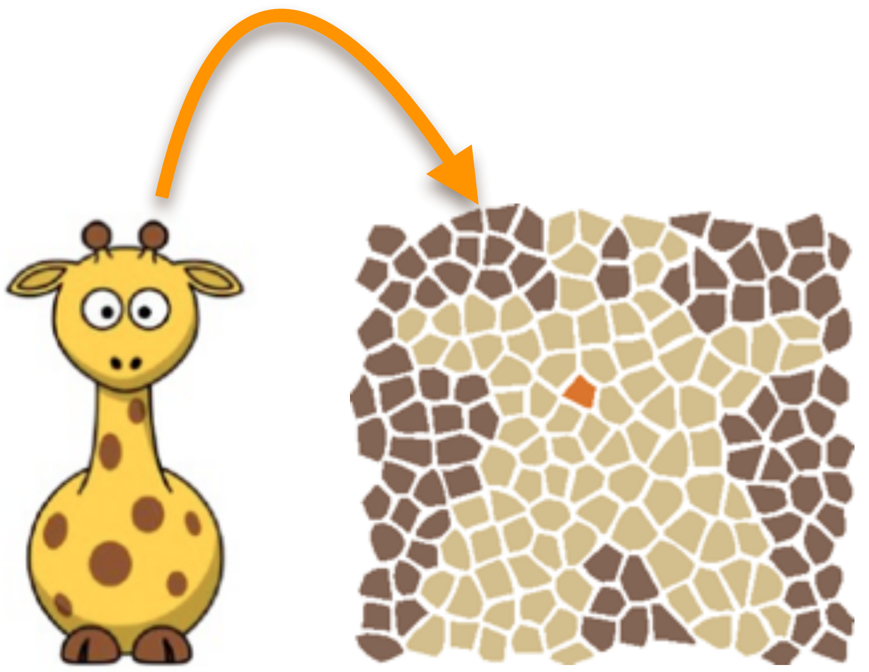
Some experimental results of Pregel

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

```
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

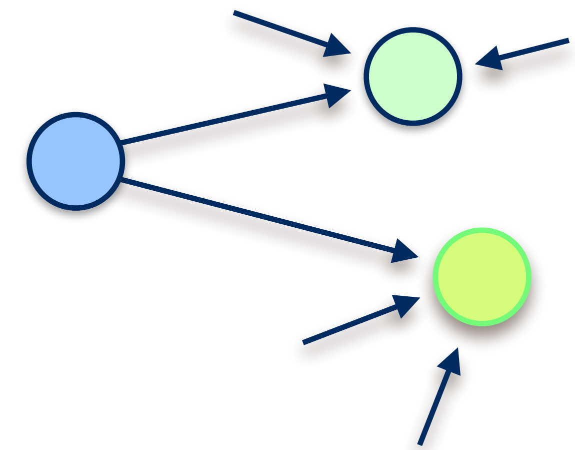
process messages from previous superstep

maximum changes

at start or after change, message connected vertices

reactivation only after incoming message

Giraph in action: indegree count



```
public class SimpleInDegreeCountComputation extends
BasicComputation<LongWritable, LongWritable, DoubleWritable, DoubleWritable> {
  @Override
  public void compute(Vertex<LongWritable, LongWritable, DoubleWritable>
                      vertex,
                      Iterable<DoubleWritable> messages) throws IOException {
    if (getSuperstep() == 0) {
      Iterable<Edge<LongWritable, DoubleWritable>> edges = vertex.getEdges();
      for (Edge<LongWritable, DoubleWritable> edge : edges) {
        sendMessage(edge.getTargetVertexId(), new DoubleWritable(1.0));
      }
    } else {
      long sum = 0;
      for (DoubleWritable message : messages) {
        sum++;
      }
      LongWritable vertexValue = vertex.getValue();
      vertexValue.set(sum);
      vertex.setValue(vertexValue);
      vertex.voteToHalt();
    }
  }
}
```

send out the inlink messages

count them up

stop

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