Anthony Ventresque
anthony.ventresque@ucd.ie

Graphs

School of Computer Science, UCD   Scoil na Ríomheolaíochta, UCD

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-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

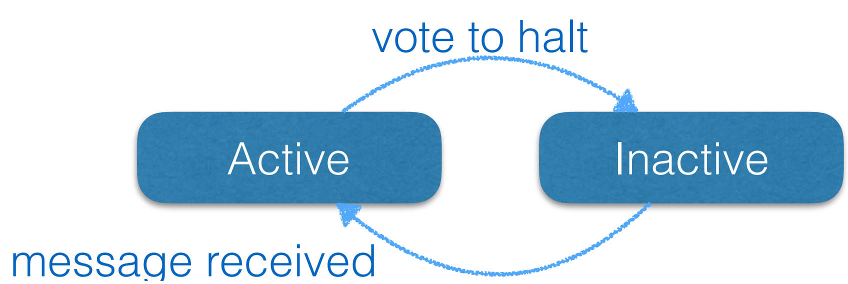
- Reminiscent of MapReduce
  - User (i.e. algorithm developer) focus on a local action
  - Each vertex is processed independently
- By design: well suited for a distributed implementation
  - All communication is from superstep S to $(S+1)$
  - No defined execution order within a superstep
  - Free of deadlocks and data races

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

Algorithm Termination

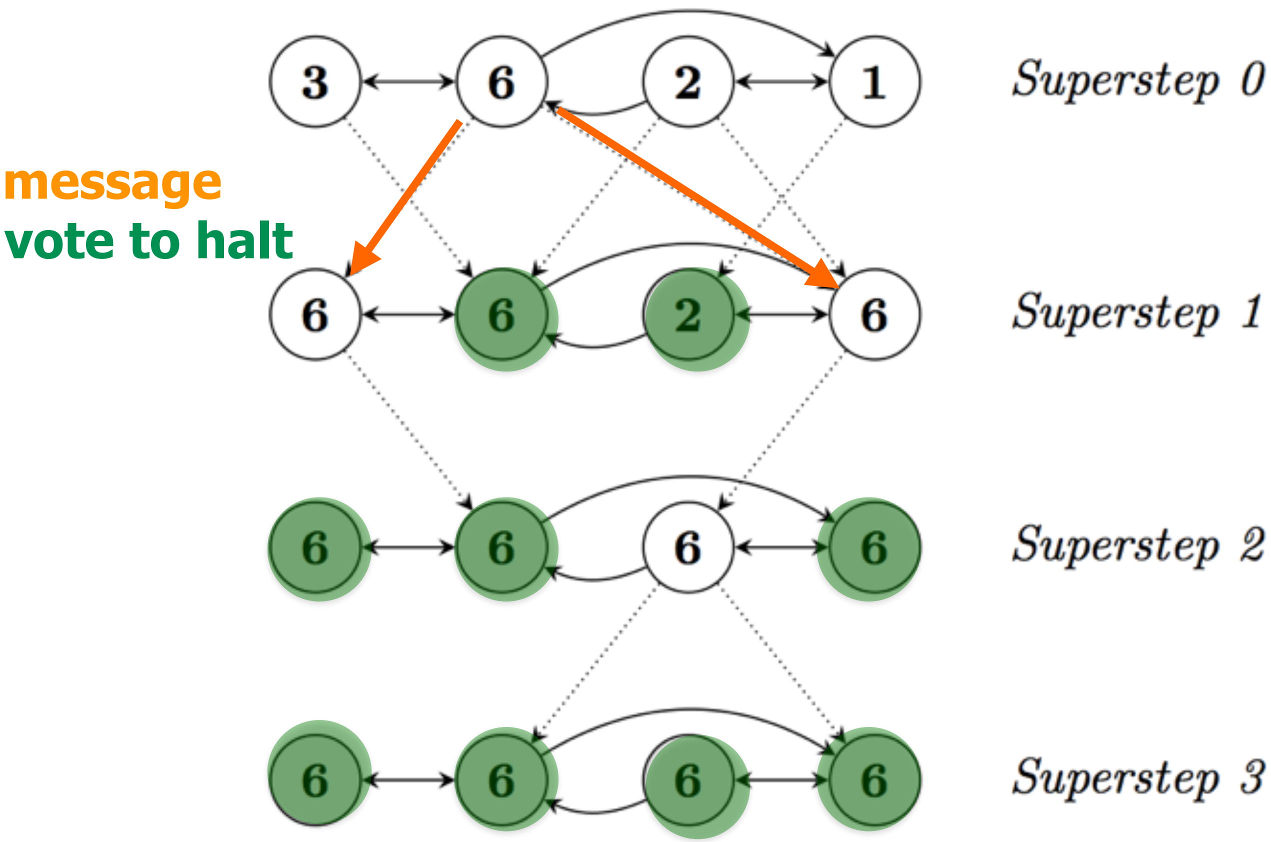
- In MapReduce: external driver program decides when to stop an iterative algorithm
- BSP-inspired Pregel:
  - Superstep 0: all vertices are active
  - All active vertices participate in the computation at each superstep
  - A vertex deactivates itself by voting to halt
  - No execution in subsequent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

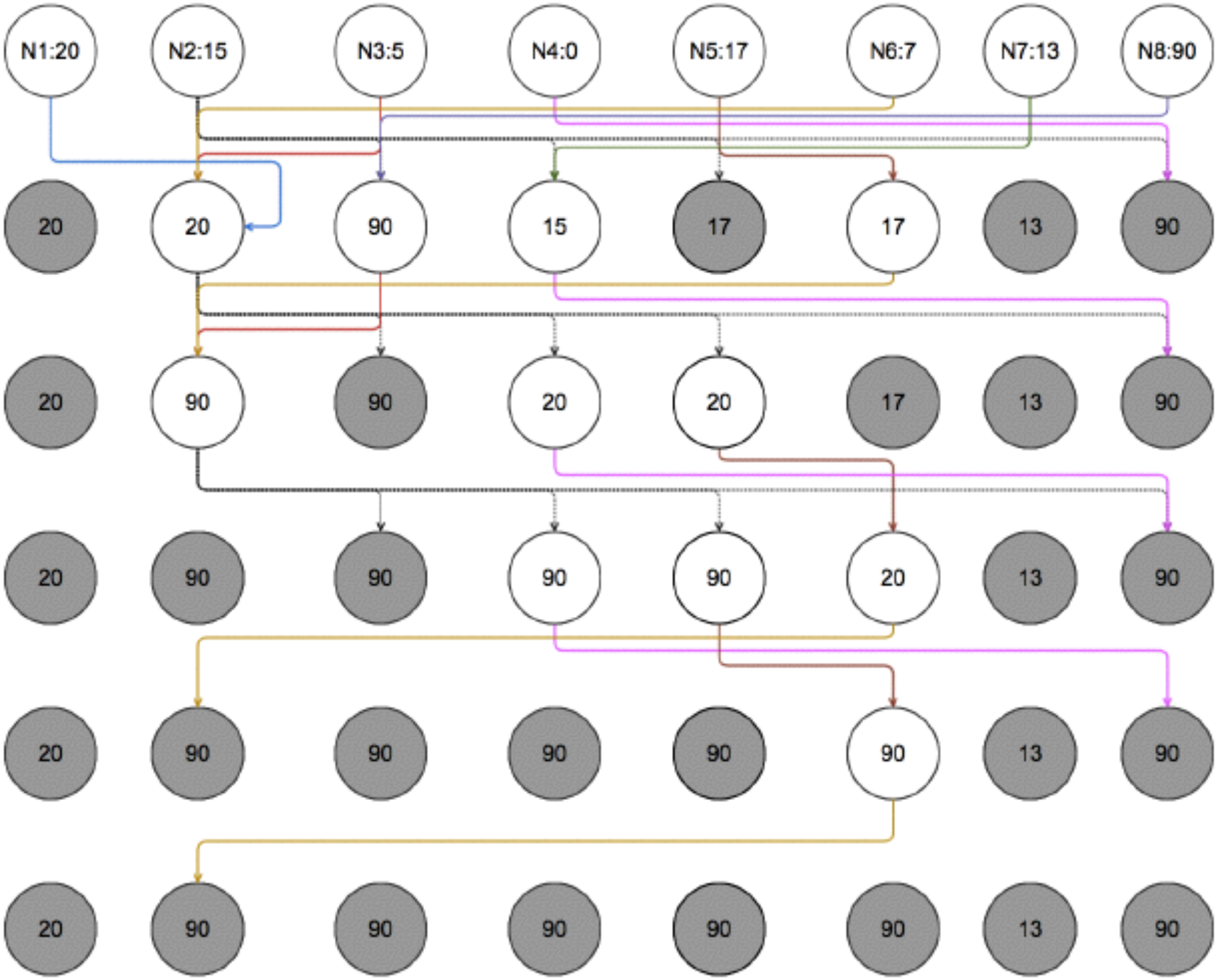
Example: Maximum Value

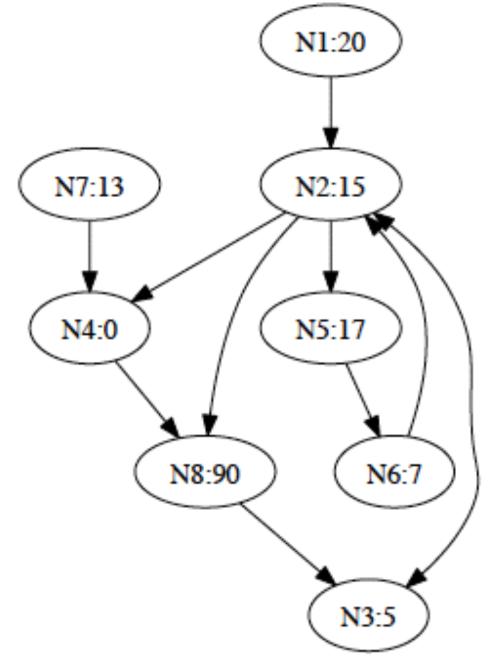


graph with four nodes and four directed edges

messages are usually send to vertices directly connected

Example (2): Maximum Value





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 previous superstep (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 EXAMPLE

Page Rank

```
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|

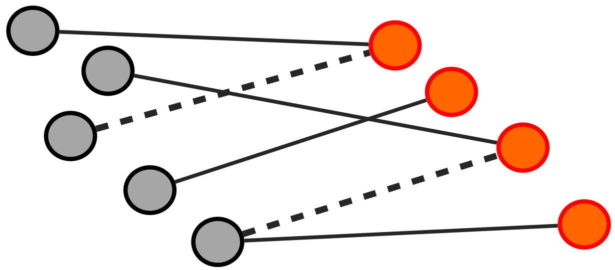
Single-source Shortest Paths

```
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

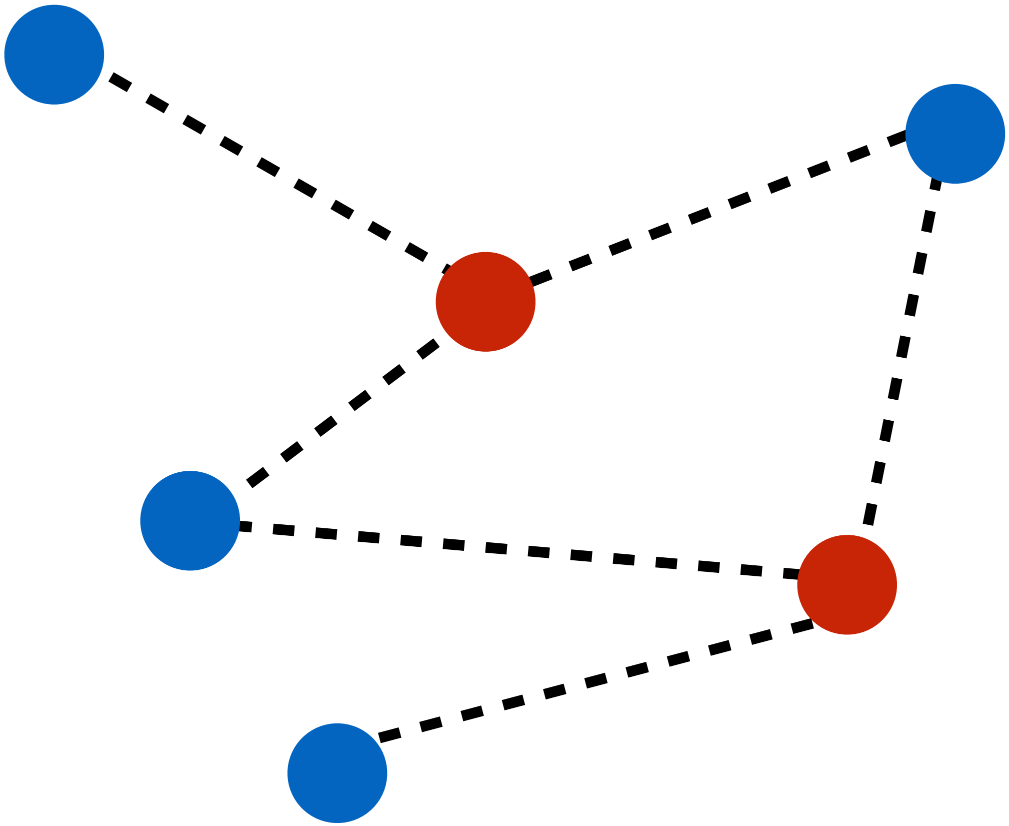
- Input: two distinct sets of vertices with only edges between them
- Output: subset of edges with no common endpoints
- Maximal matching: no more edges can be added without violating the no-common-endpoints condition
- Vertex values: tuple of Left/Right flag (is the vertex a “left” or “right” one) and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



Bipartite Matching

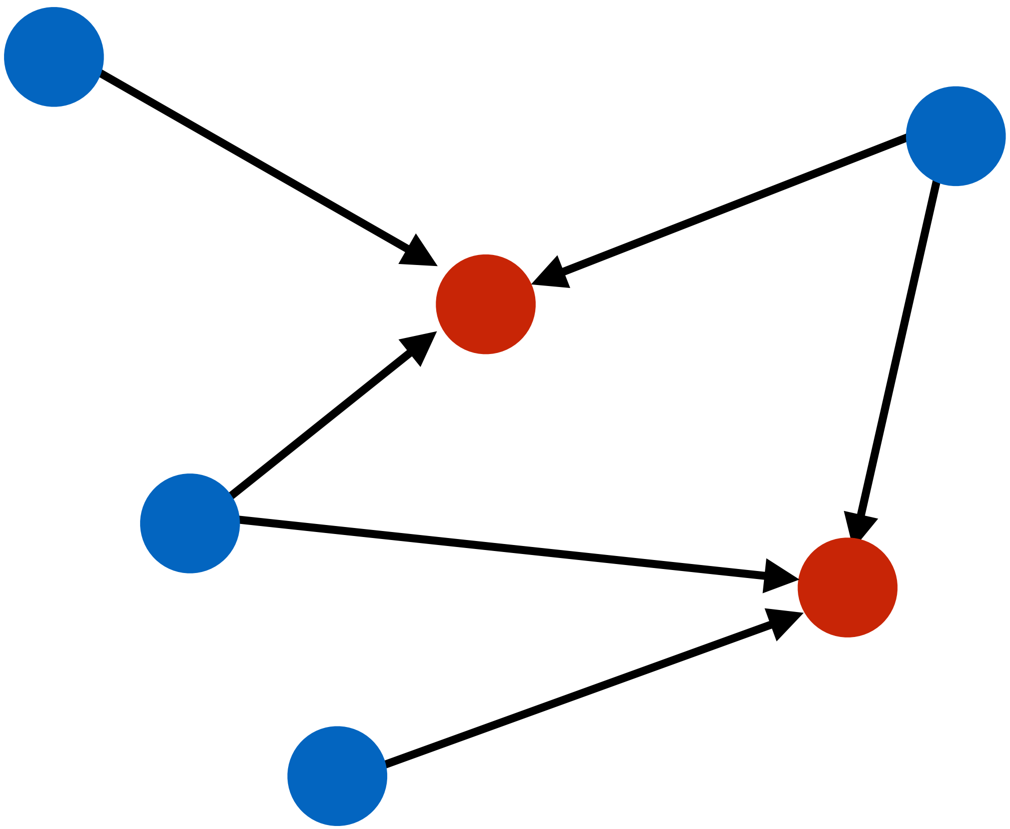
- Randomised maximal matching:
  1. Each *left* vertex not yet matched sends a message to each neighbour to request a match; vote to halt
  2. Each *right* vertex not yet matched randomly chooses one of the messages it receives, grants the request and informs all requesters about decision; vote to halt
  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

Bipartite Matching



(blue, red)

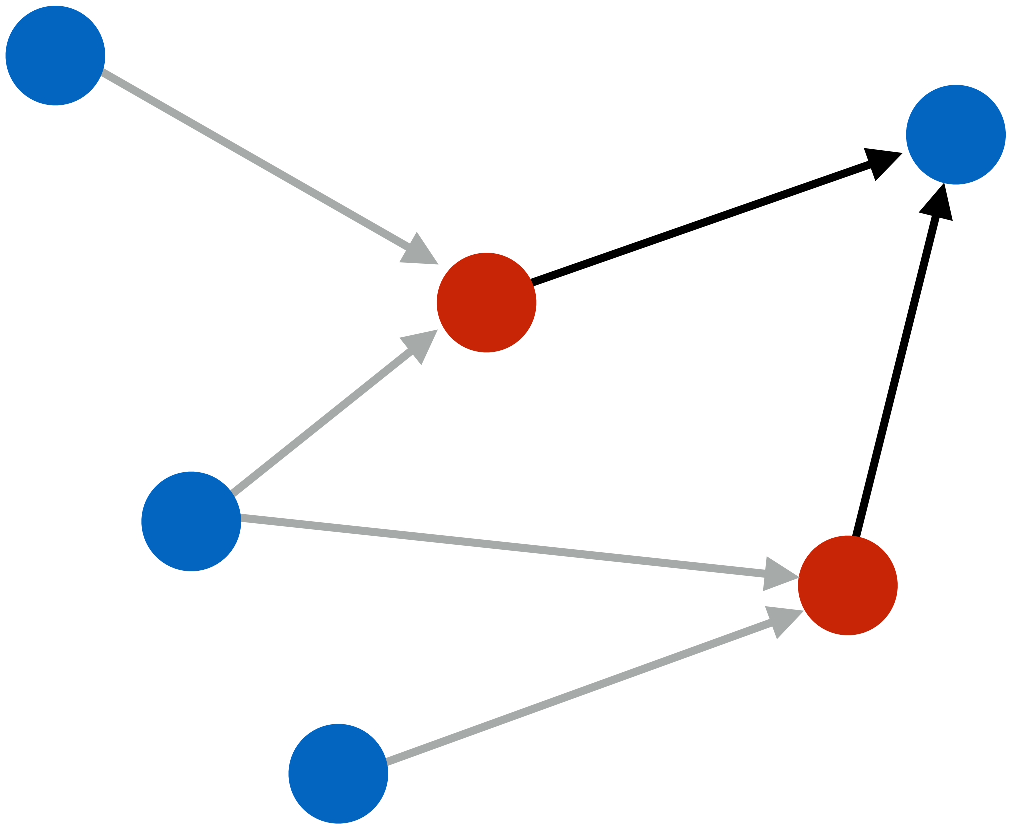
Bipartite Matching



(blue, red)

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

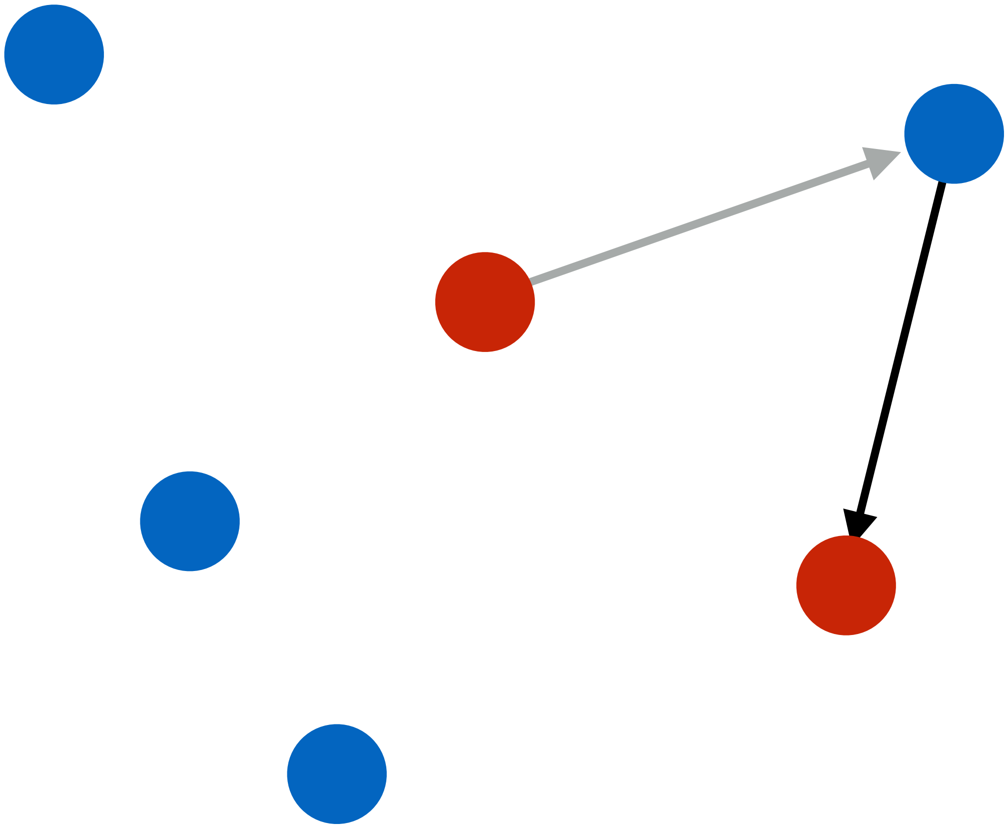
Bipartite Matching



(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

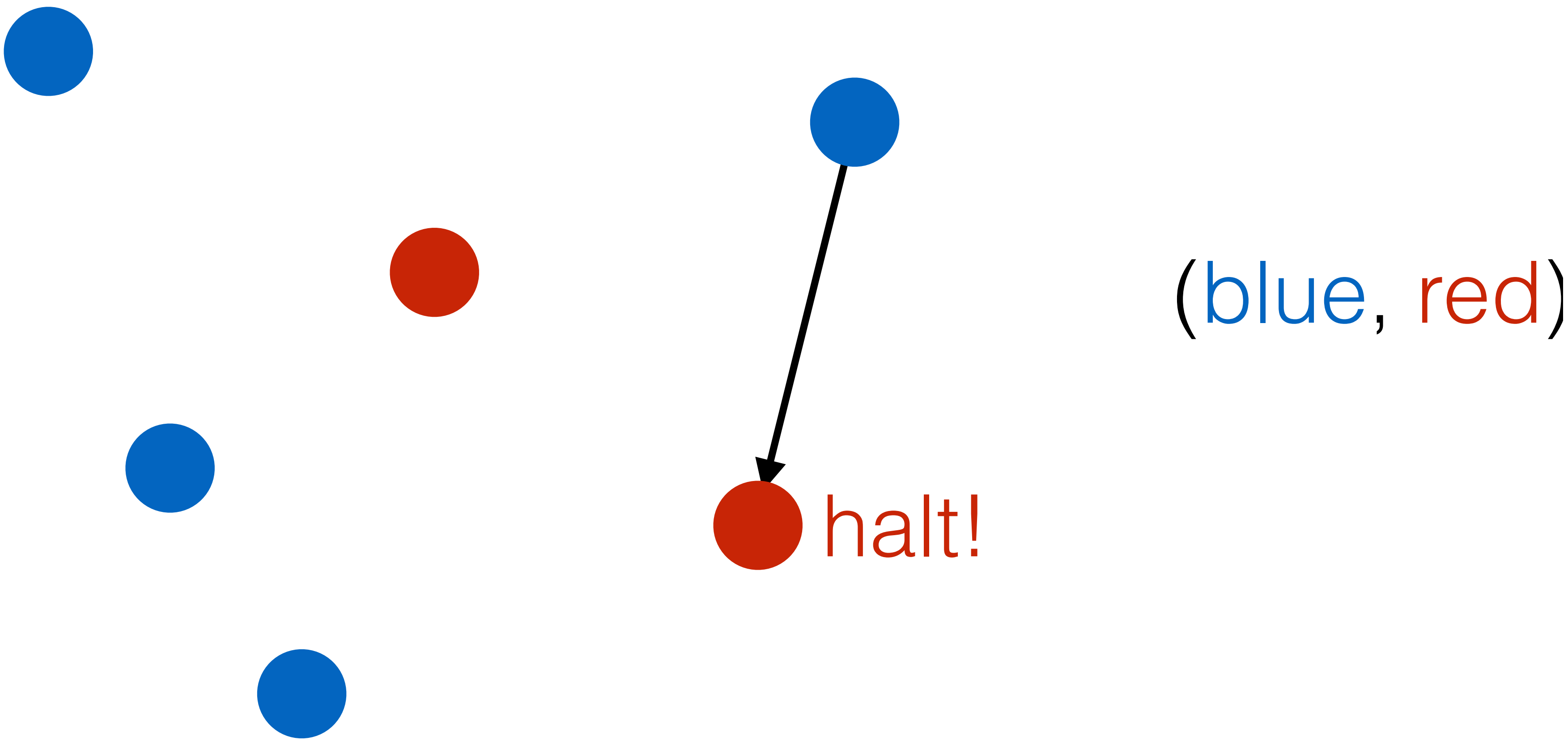
Bipartite Matching



(blue, red)

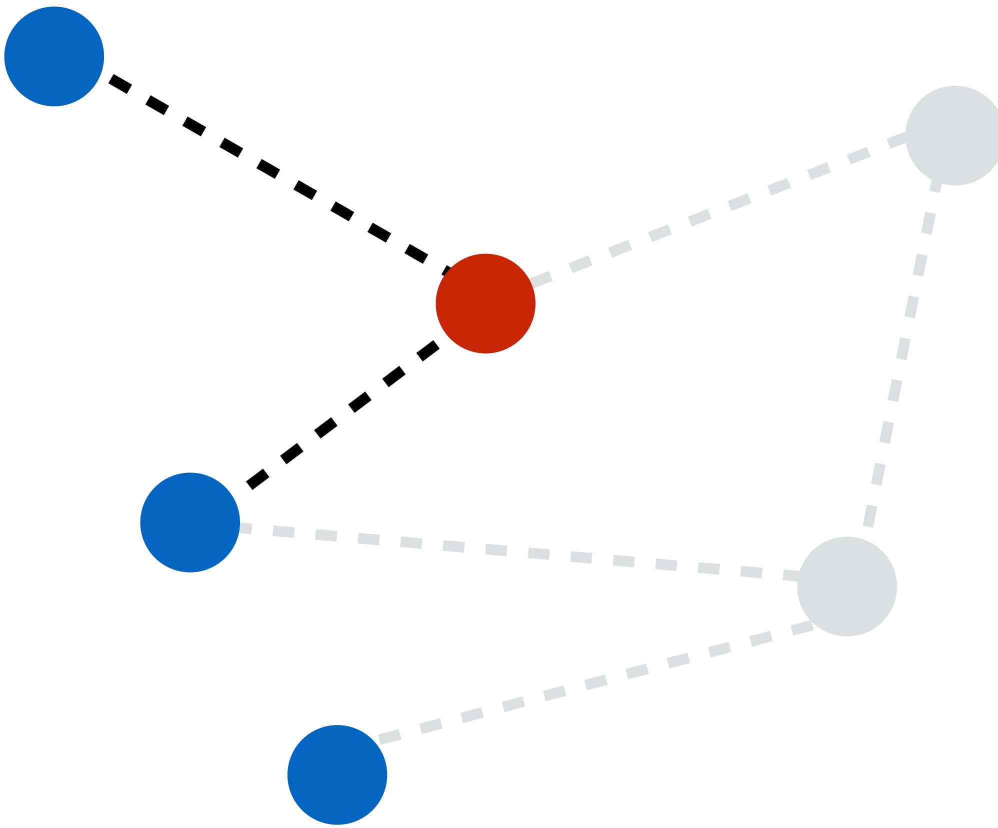
3. Each *left* vertex not yet matched randomly chooses one of the grants it received and sends acceptance back

Bipartite Matching



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

Bipartite 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: *Cmax* clusters each with at most *Vmax* vertices
- Also called “semi-clustering”

Soft Clustering

Cluster score

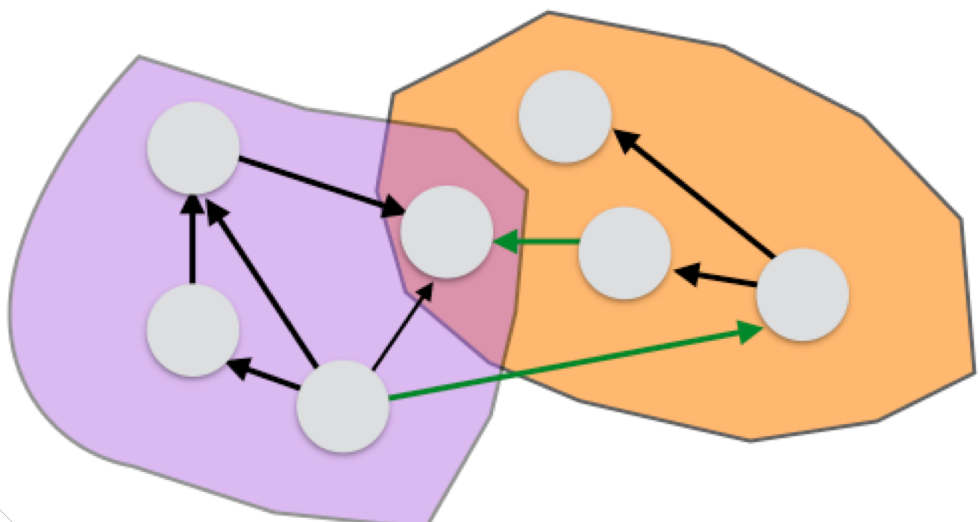
sum of weights of internal edges

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 *Cmax* 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 *c1..ck* 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 *c1..ck*, *d1..dk* 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 *c1..ck*, *d1..dk* 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

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)
- 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)

Giraph in action: maximum value in a graph

```
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

reactivation only after incoming message

at start or after change, message connected vertices

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