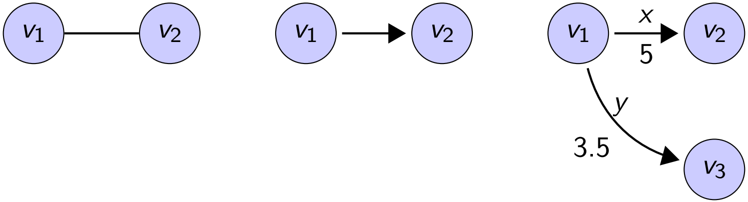
Vertex Centric & Bulk Synchronous Parallel Models and Solutions

Kaustubh Beedkar
kaustubh.beedkar@iitd.ac.in

CSE
IIT Delhi

Recap: Graphs 101

- **Graph** $G = (V, E)$
  - Abstract representation of a set of objects (**vertices** V)
  - Some pairs of these objects are connected by links (**edges** E)
  - Edges can be directed or undirected, can have labels, weights, ...
- Examples



Real word graph examples

Web around Wikipedia

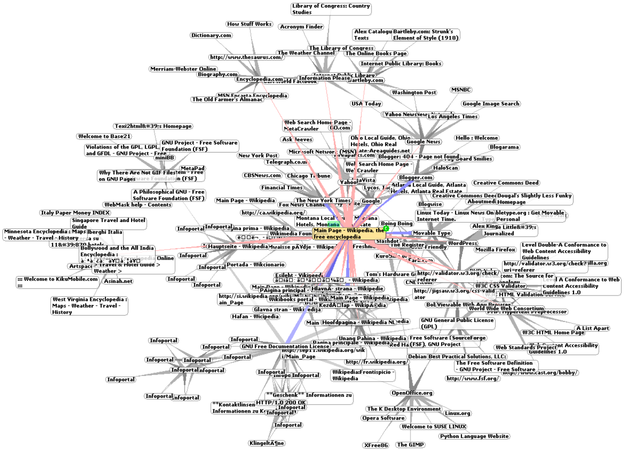


Image source

Real word graph examples

Social network

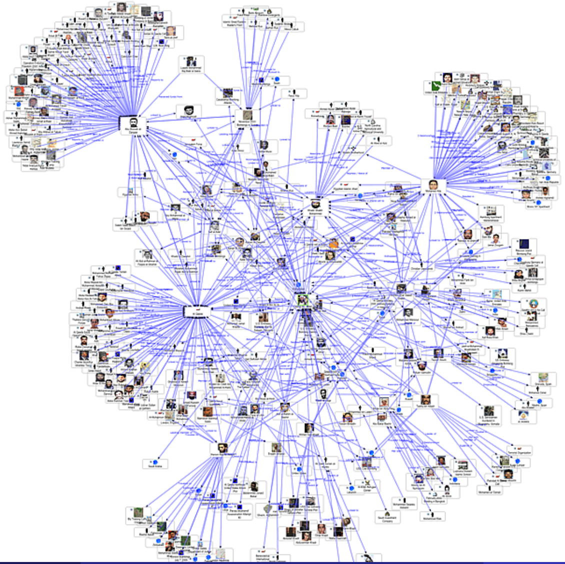


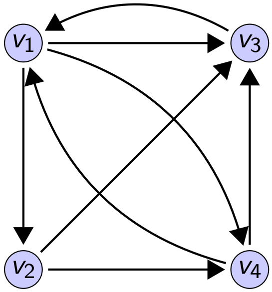
Image source

Graph algorithms

- PageRank
- Clustering
- Connected components
- Diameter finding/shortest path
- Graph pattern mining
- Machine learning/data mining (MLDM) algorithms
  - Belief propagation
  - Gaussian non-negative matrix factorization
  - ...

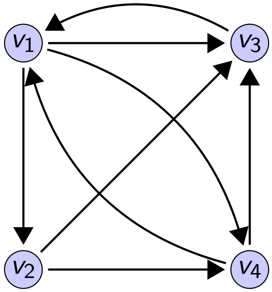
PageRank: example

- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u



PageRank: example

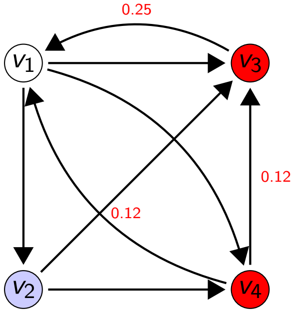
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k = 0$
$PR(v_1)$	0.25
$PR(v_2)$	0.25
$PR(v_3)$	0.25
$PR(v_4)$	0.25

PageRank: example

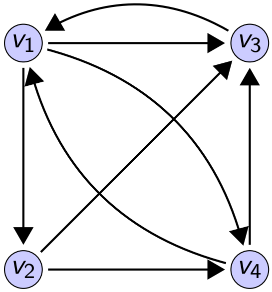
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$
$PR(v_1)$	0.25
$PR(v_2)$	0.25
$PR(v_3)$	0.25
$PR(v_4)$	0.25

PageRank: example

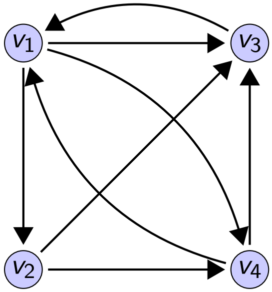
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$	$k=1$
$PR(v_1)$	0.25	0.37
$PR(v_2)$	0.25	0.08
$PR(v_3)$	0.25	0.33
$PR(v_4)$	0.25	0.20

PageRank: example

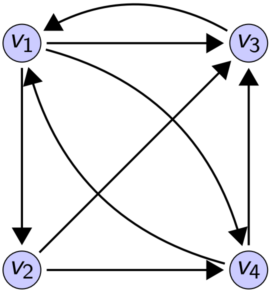
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$	$k=1$	$k=2$
$PR(v_1)$	0.25	0.37	0.43
$PR(v_2)$	0.25	0.08	0.12
$PR(v_3)$	0.25	0.33	0.27
$PR(v_4)$	0.25	0.20	0.16

PageRank: example

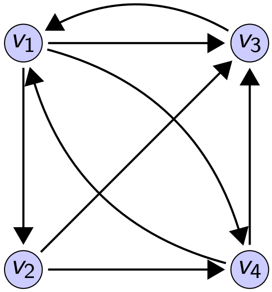
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$	$k=1$	$k=2$	$k=3$
$PR(v_1)$	0.25	0.37	0.43	0.35
$PR(v_2)$	0.25	0.08	0.12	0.14
$PR(v_3)$	0.25	0.33	0.27	0.29
$PR(v_4)$	0.25	0.20	0.16	0.20

PageRank: example

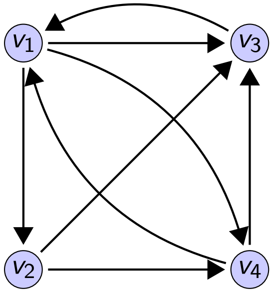
- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$
$PR(v_1)$	0.25	0.37	0.43	0.35	0.39
$PR(v_2)$	0.25	0.08	0.12	0.14	0.11
$PR(v_3)$	0.25	0.33	0.27	0.29	0.29
$PR(v_4)$	0.25	0.20	0.16	0.20	0.19

PageRank: example

- $PR_{k+1}(u) = \Sigma_{v \in B_u} \left(\frac{PR_k(v)}{|F_v|} \right)$
  - $PR(u)$: Page rank of node u
  - F_u : Out-neighbors of node u
  - B_u : In-neighbors of node u
- Iterative batch processing



	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
$PR(v_1)$	0.25	0.37	0.43	0.35	0.39	0.39	0.38
$PR(v_2)$	0.25	0.08	0.12	0.14	0.11	0.13	0.13
$PR(v_3)$	0.25	0.33	0.27	0.29	0.29	0.28	0.28
$PR(v_4)$	0.25	0.20	0.16	0.20	0.19	0.19	0.19

Real-world graphs are huge!

$\sim$1T indexed pages

$\sim$1B Users

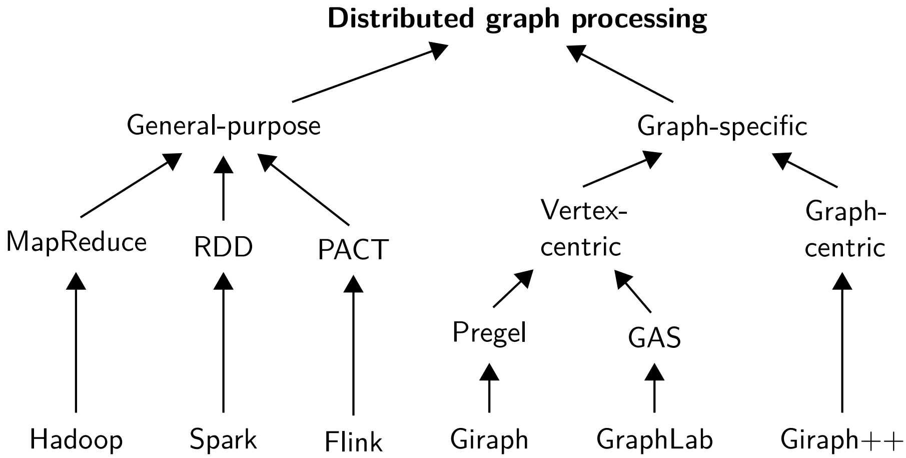
$\sim$450M active users

$>$100M Ratings, 230M Users, 17K Movies

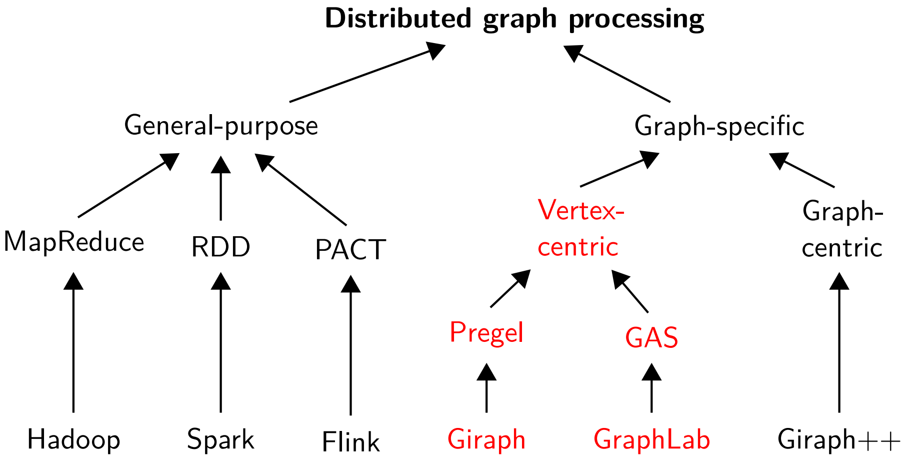
Large-scale graph processing challenges

- Each vertex depends on its neighbors, recursive
- Recursive problems are nicely solved iteratively
- Challenges
  - partitioning
  - recursive joins
  - graph data is often unstructured
  - and highly irregular
  - poor locality of memory access

Systems for distributed graph processing

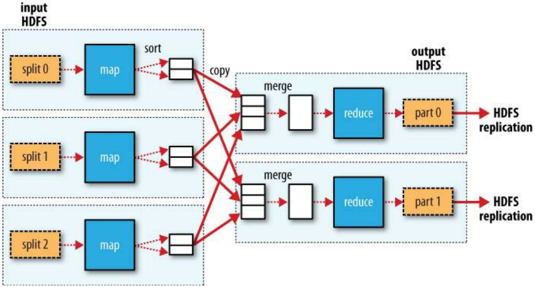


Systems for distributed graph processing



- In this lecture

MapReduce for large-scale graph processing



... Repeat N times ...

- MapReduce drawbacks
  - Each job is executed N times
  - Overhead of job bootstrap
  - Mappers send values and structure
  - Extensive IO at input, shuffle & sort, output
  - Disk I/O and Job scheduling quickly dominate the runtime

Outline

1. Background
2. Vertex-centric approaches
   - Pregel/Giraph
   - GraphLab
   - PowerGraph
3. Discussion
   - Dataflow engines
   - Summary

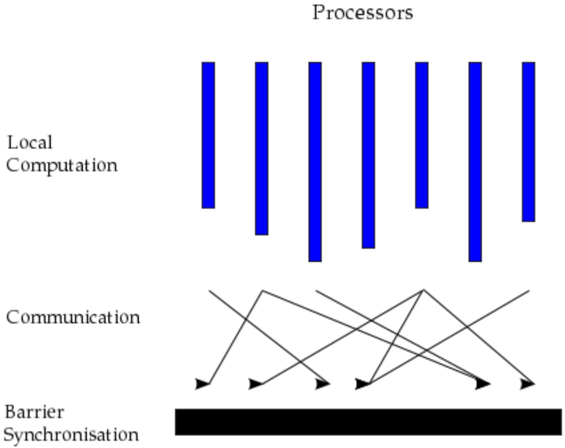
Outline

Google Pregel

- A system for large-scale graph processing
- Bulk Synchronous Parallel (BSP) as execution model
- Intuitive API that let's you "think like a vertex"
- Fault tolerance by checkpointing

Bulk Synchronous Parallel (BSP)

- Originally developed by Leslie Valiant
- Key idea:



- Processors
- Have some local memory
- Can perform some (local) computation

- Processors can communicate pairwise
- Communication can overlap with another node's computation

Bulk Synchronous Parallel (BSP)

- A BSP program proceeds in a sequence of **supersteps**
  1. Superstep 1
     - Get required data
     - Compute — Yes or No?
     - Exchange messages — Yes or No?

     Synchronize
  2. Superstep 2
     - Get required data
     - Compute — Yes or No?
     - Exchange messages — Yes or No?

     Synchronize
  3. **Superstep 3**

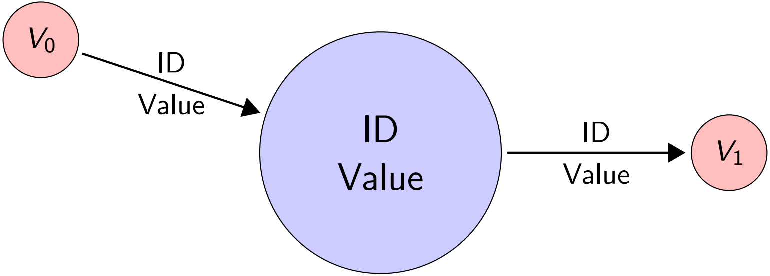
     ...

     Synchronize

     ...

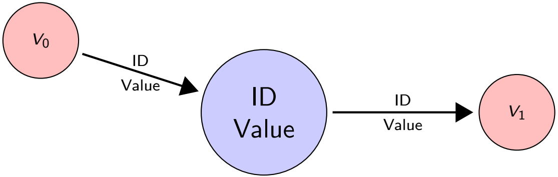
BSP as a programming model for graphs

- Assumptions
  - Vertex synonymous to processor
  - Vertex can send/receive messages to/from its neighboring vertices
  - Each vertex has an ID and a value
  - Each edge (may) also have an ID and a value
  - Each vertex knows which vertex it is connected to
- Think of computation as a **Vertex Centric Task**



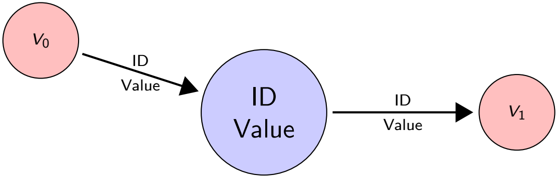
Think like a Vertex

- What can a Vertex do?



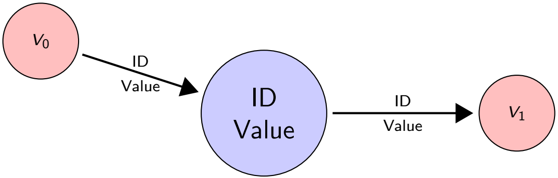
Think like a Vertex

- What can a Vertex do?
  - Get its ID



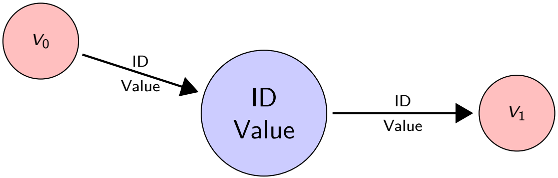
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value



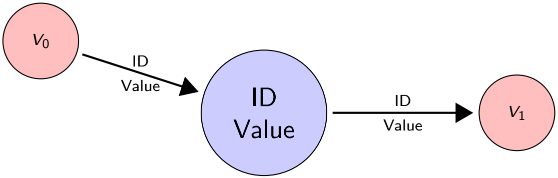
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges



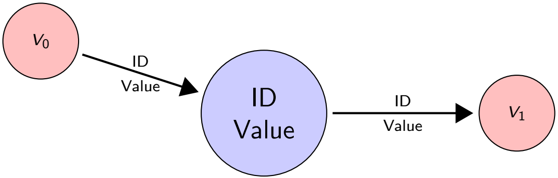
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value



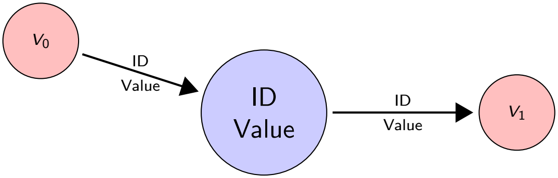
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value
    - Using edge ID



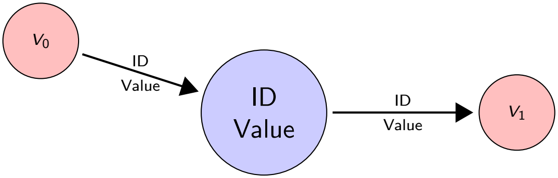
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value
    - Using edge ID
    - Using the ID of the target vertex



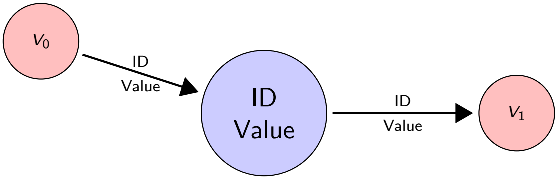
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value
    - Using edge ID
    - Using the ID of the target vertex
  - Gets values of all edges connected to a vertex



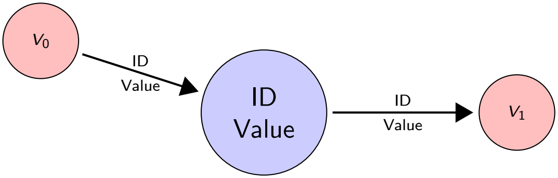
Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value
    - Using edge ID
    - Using the ID of the target vertex
  - Gets values of all edges connected to a vertex
  - Add/remove a specific edge

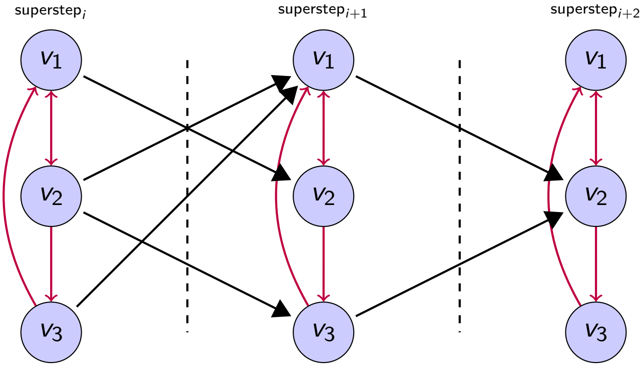


Think like a Vertex

- What can a Vertex do?
  - Get its ID
  - Get/set its value
  - Get/count its edges
  - Get/set a specific edge's value
    - Using edge ID
    - Using the ID of the target vertex
  - Gets values of all edges connected to a vertex
  - Add/remove a specific edge
  - Start/Stop computing

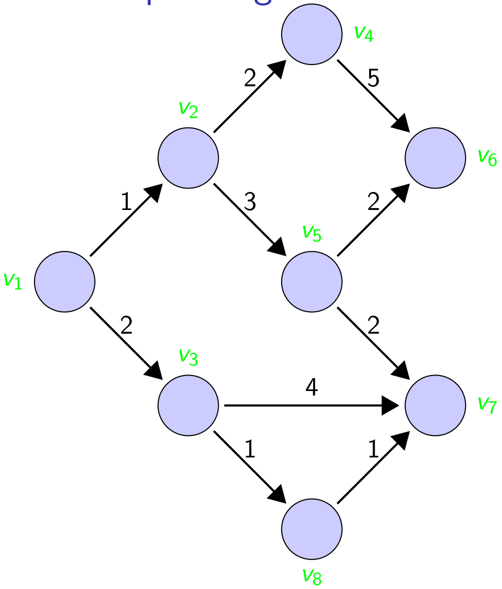


Vertex-centric BSP



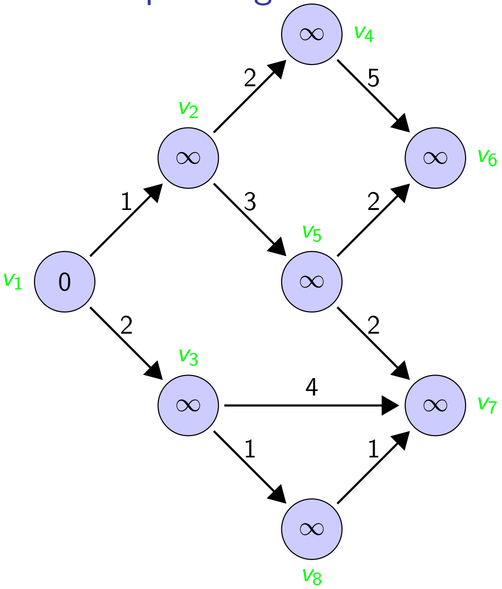
- Each vertex has an id/value, list of adj. neighbor ids, edge values
- Each vertex in invoked in each superstep
- Can re-compute its value and send messages to other vertices, which are delivered over superstep barriers
- Adv. features: termination votes, combiners, aggegrators, topology mutation

Example: Dijkstra's single source shortest path algorithm



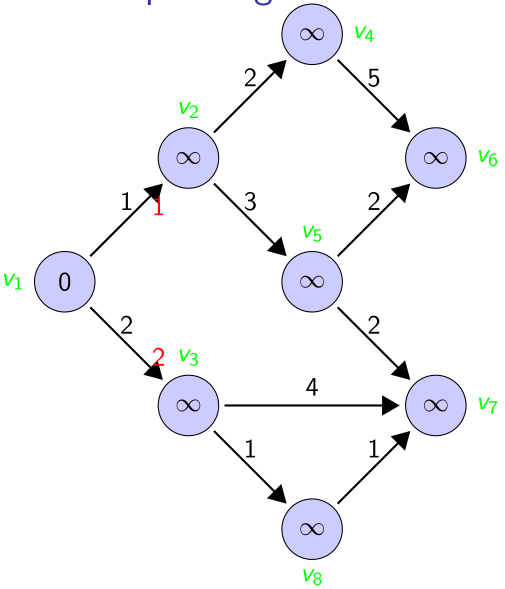
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize



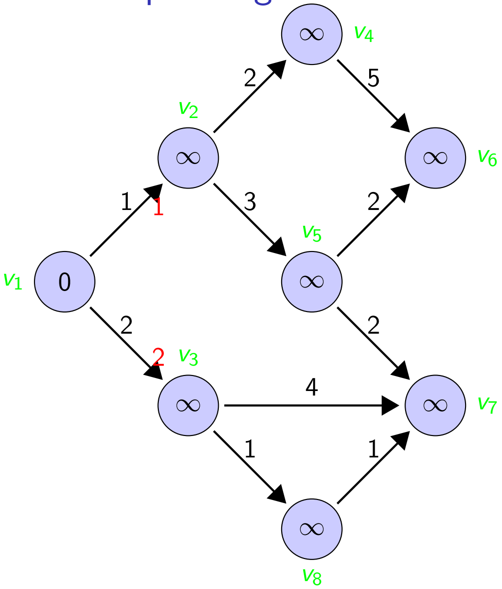
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate



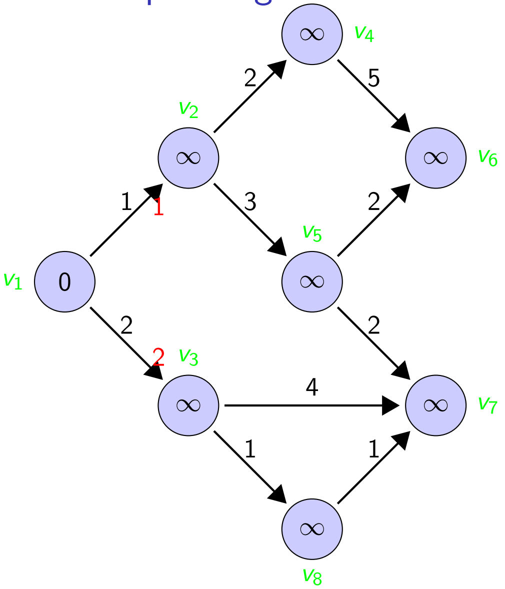
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt



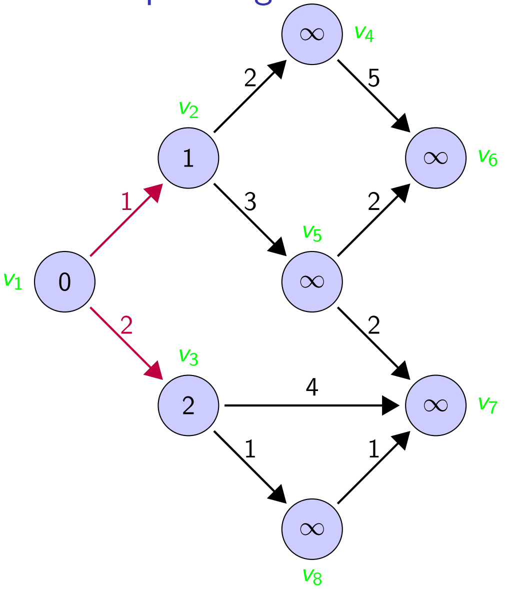
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages



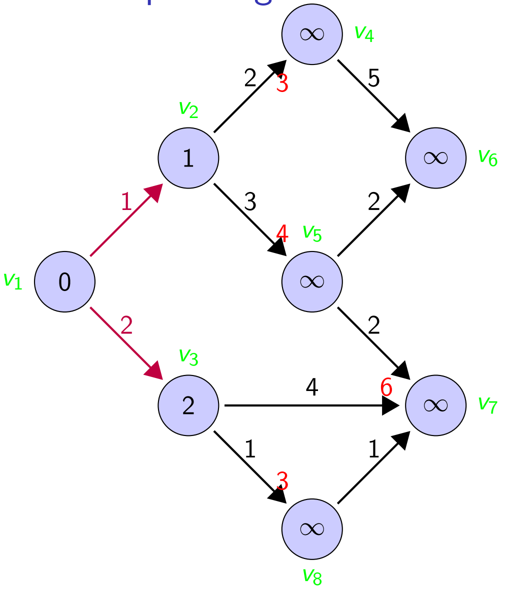
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update



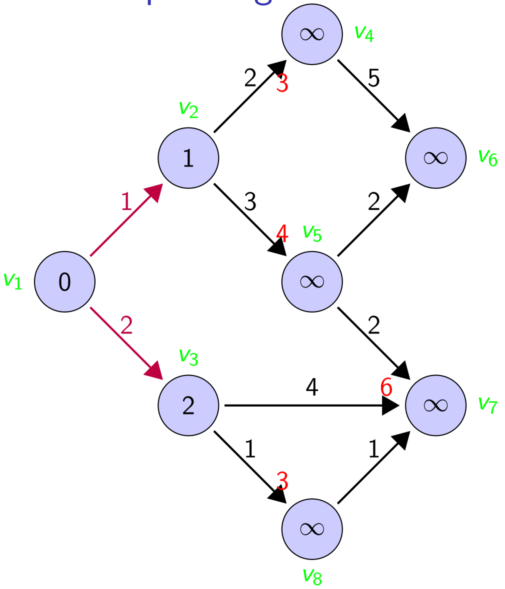
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate



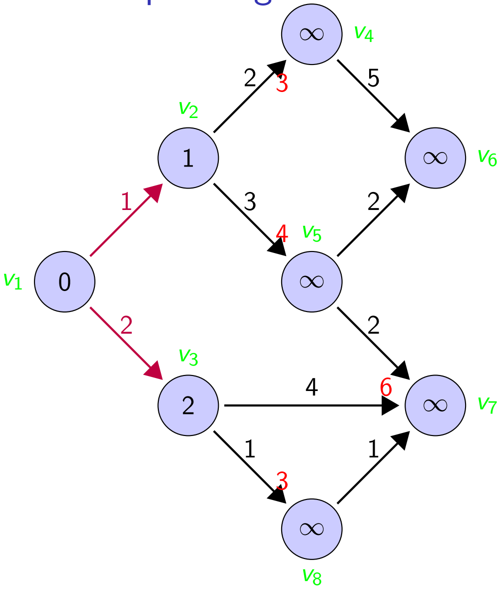
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt



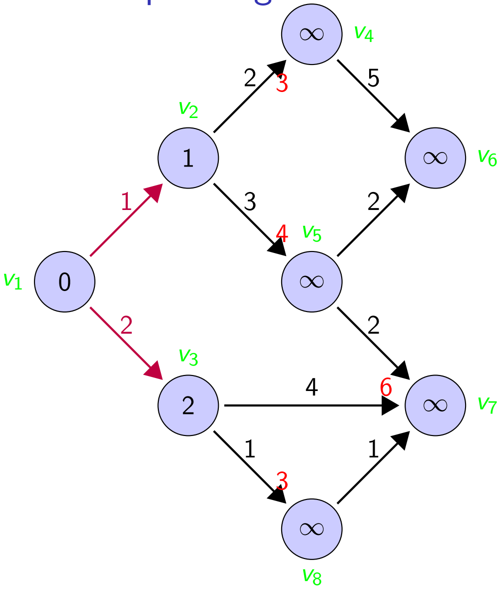
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2



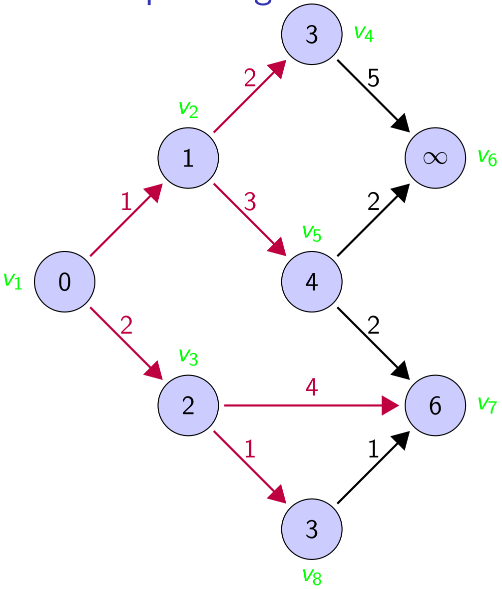
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages



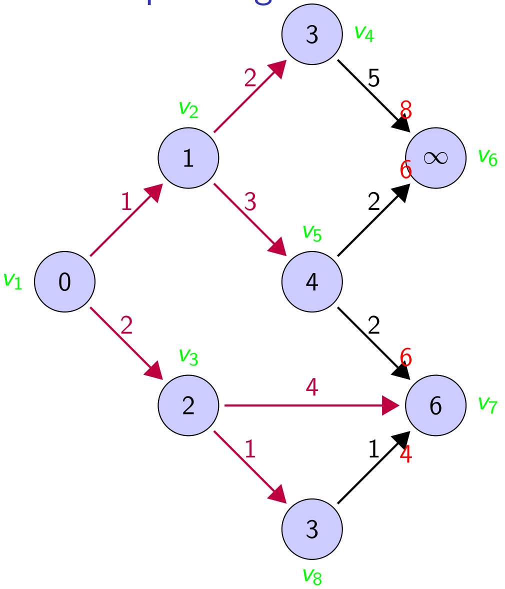
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update



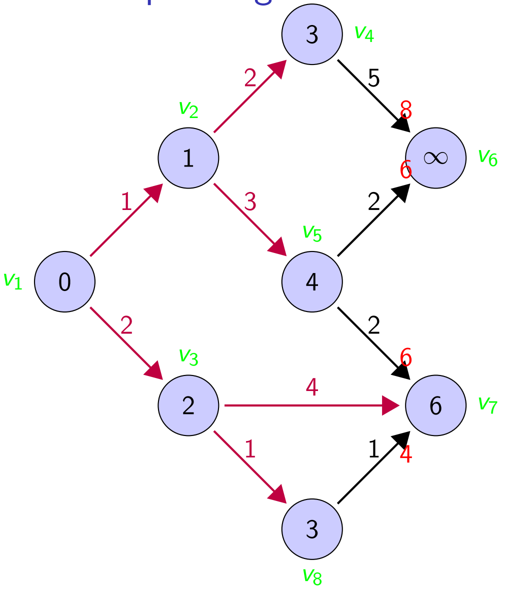
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update
  - Propagate



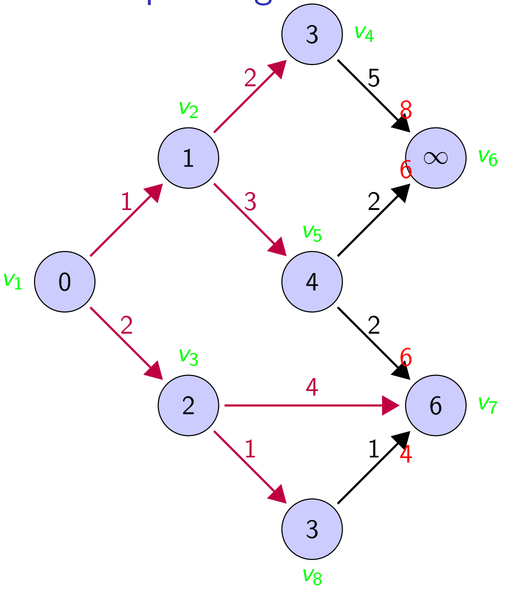
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update
  - Propagate
  - Halt



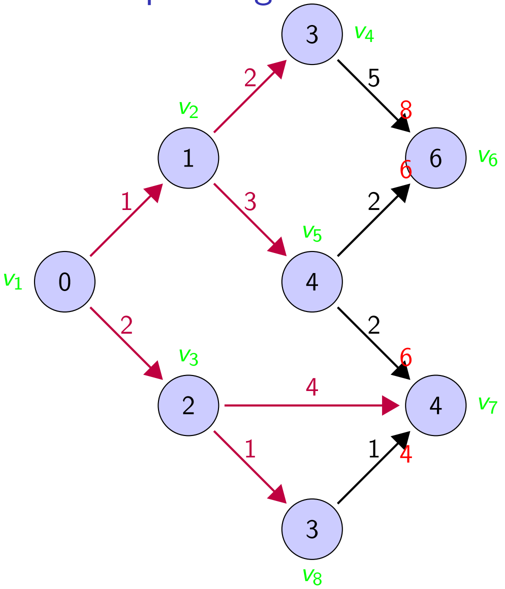
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 3
  - GetMessages



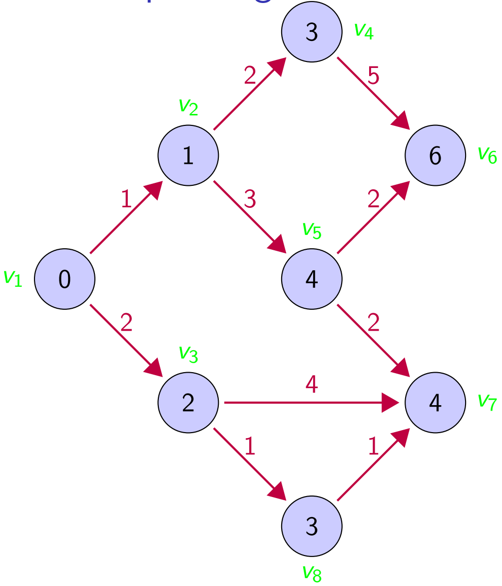
Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 3
  - GetMessages
  - Update



Example: Dijkstra's single source shortest path algorithm

- Superstep 0
  - Initialize
  - Propagate
  - Halt
- Superstep 1
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 2
  - GetMessages
  - Update
  - Propagate
  - Halt
- Superstep 3
  - GetMessages
  - Update
  - ~~Propagate~~
  - Halt



Shortest path (code snippet)

```
public static class ShortestPathVertex extends Vertex<Text, IntWritable, IntWritable> {
    public void compute(Iterator<IntWritable> messages) throws IOException {

        int minDist = isStartVertex() ? 0 : Integer.MAX_VALUE;

        while (messages.hasNext()) {
            IntWritable msg = messages.next();
            if (msg.get() < minDist) {
                minDist = msg.get();
            }
        }
        if (minDist < this.getValue().get()) {
            this.setValue(new IntWritable(minDist));
            for (Edge<Text, IntWritable> e : this.getEdges()) {
                sendMessage(e, new IntWritable(minDist + e.getValue().get()));
            }
        }
        else {
            voteToHalt();
        }
    }
}
```

Pregel implementation

- Pregel uses **master-worker architecture**
- Initialization
  - **Graph is partitioned by vertex ID** $(\text{hash}(ID) \mod N)$
    - vertex and its adjacency list live in the same partition
  - Master assigns partitions to workers which load graph in memory
- **Master coordinates supersteps**
  - During a superstep, **workers invoke UDF on their local partitions during a superstep and asynchronously deliver messages**
  - Execution continues as long as there are active vertices or messages to be delivered
  - During termination, vertices output their state as result of the computation
  - **Synchronization barrier** via distributed locking service
  - Fault tolerance is achieved through **checkpointing**

Apache Giraph

- Pregel is **proprietary**, but:
  - **Apache Giraph** is an open source implementation of Pregel
  - Runs on **standard Hadoop infrastructure**
  - Computation is executed in **memory** (also has out-of-core support)
  - Uses **Apache ZooKeeper** for synchronization

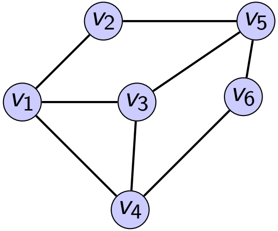
Bulk Synchronous Parallel

- Advantages
  - No race conditions
  - Synchronization barrier guarantees data consistency
  - Simple to make fault tolerant $\rightarrow$ save data on barrier

Bulk Synchronous Parallel

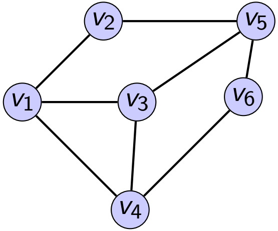
- Advantages
  - No race conditions
  - Synchronization barrier guarantees data consistency
  - Simple to make fault tolerant $\rightarrow$ save data on barrier
- Disadvantages
  - Costly performance since runtime of each superstep depends on slowest machine
  - No support for asynchronous, graph-parallel computations
    - Critical for MLDM algorithms

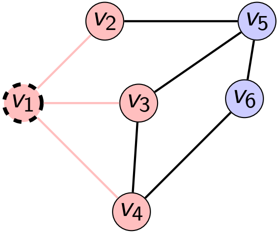
Properties of graph parallel algorithms



Dependency graph

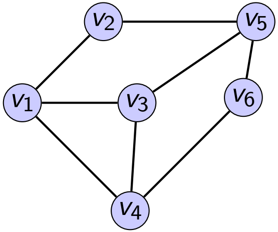
Properties of graph parallel algorithms



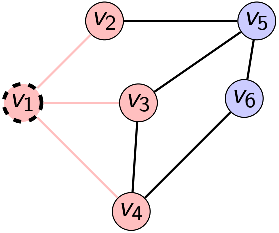


Dependency graph **Local updates**

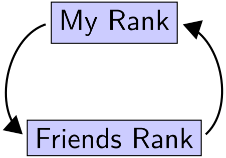
Properties of graph parallel algorithms



Dependency graph

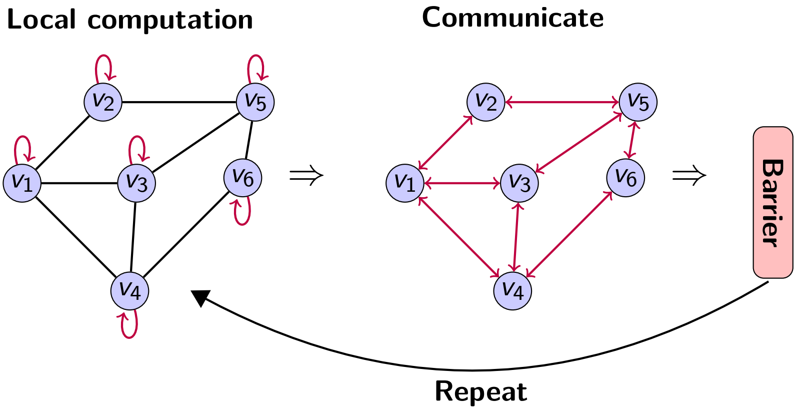


Local updates



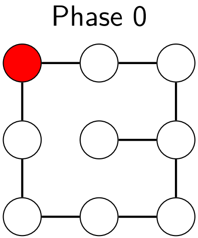
Iterative computation

Bulk Synchronous Parallel: recap



Problem with Bulk Synchronous

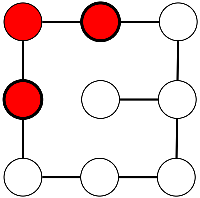
- Example: If red neighbor then turn red



Problem with Bulk Synchronous

- Example: If red neighbor then turn red

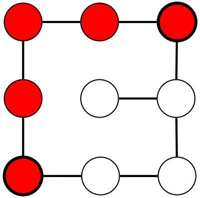
Phase 1



Problem with Bulk Synchronous

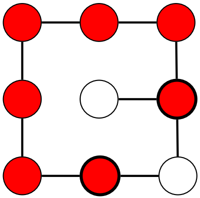
- Example: If red neighbor then turn red

Phase 2



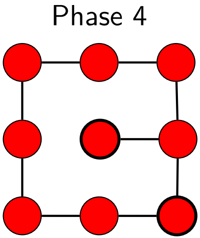
Problem with Bulk Synchronous

- Example: If red neighbor then turn red



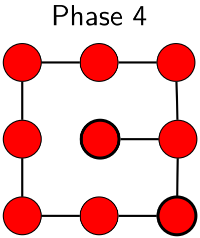
Problem with Bulk Synchronous

- Example: If red neighbor then turn red



Problem with Bulk Synchronous

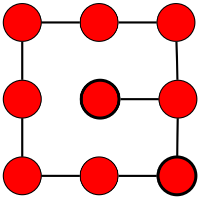
- Example: If red neighbor then turn red



- Bulk Synchronous computation
  - Evaluate condition on all vertices for every phase
    4 phases with 9 computations $\Rightarrow$ **36 computations**

Problem with Bulk Synchronous

- Example: If red neighbor then turn red



- Bulk Synchronous computation
  - Evaluate condition on all vertices for every phase
    4 phases with 9 computations ⇒ **36 computations**

- Asynchronous computation (Wave-font)
  - Evaluate condition only when neighbor changes
    4 phases each with 2 computation ⇒ **8 computations**

Outline

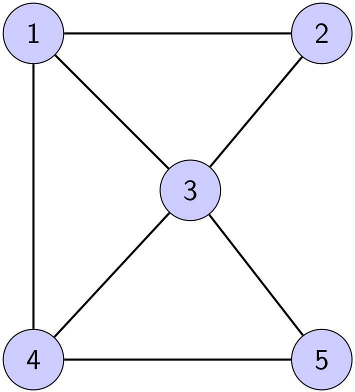
1. Background
2. Vertex-centric approaches
   - Pregel/Giraph
   - GraphLab
   - PowerGraph
3. Discussion
   - Dataflow engines
   - Summary

GraphLab

- Support for asynchronous updates
  - Critical for MLDM algorithms
    - Graph structured computation
    - Asynchronous iterative computation
    - Dynamic computation
    - Serializability
- GraphLab abstraction
  - Data graph
  - Update functions
  - Data consistency
- Shared memory (2010), Distributed memory (2012)

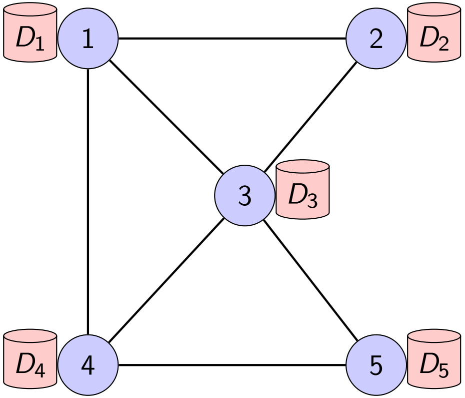
Data Graph

- **Graph** $G = (V, E, D)$



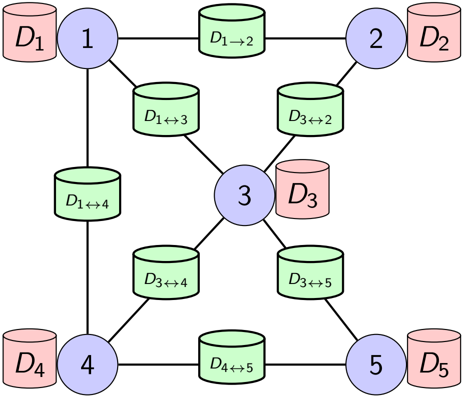
Data Graph

- **Graph** $G = (V, E, D)$
  - D_i is user defined data associated with vertex i



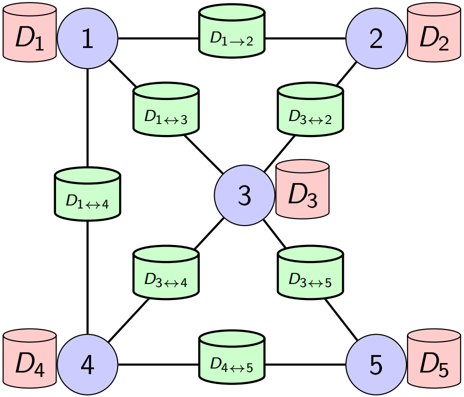
Data Graph

- **Graph** $G = (V, E, D)$
  - D_i is user defined data associated with vertex i
  - $D_{i \to j}$ is user defined data associated with each edge $i \to j$



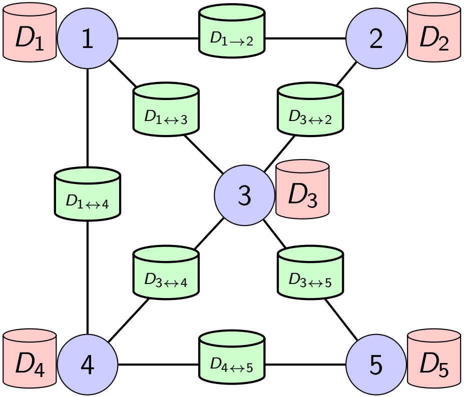
Data Graph

- **Graph** $G = (V, E, D)$
  - D_i is user defined data associated with vertex i
  - $D_{i \to j}$ is user defined data associated with each edge $i \to j$
    - represents model parameters, algorithm state, and statistical data



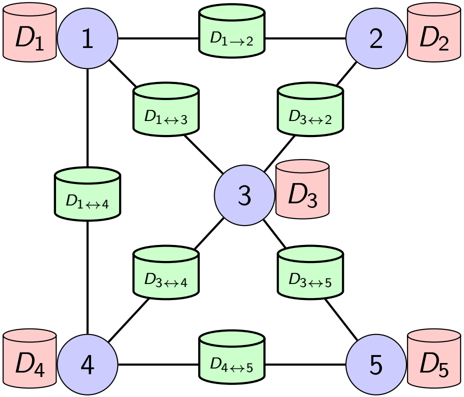
Data Graph

- **Graph** $G = (V, E, D)$
  - D_i is user defined data associated with vertex i
  - $D_{i \to j}$ is user defined data associated with each edge $i \to j$
    - represents model parameters, algorithm state, and statistical data
- Example: Social network
  - Vertex Data D_i
    - User profile
    - Current interests estimates

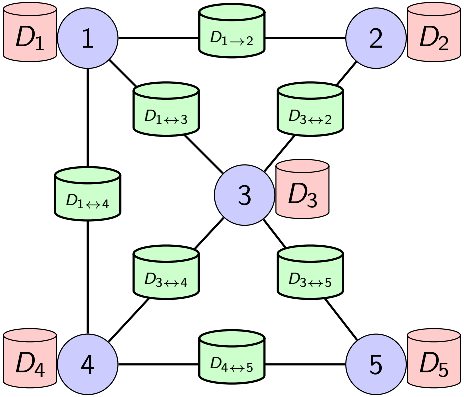


Data Graph

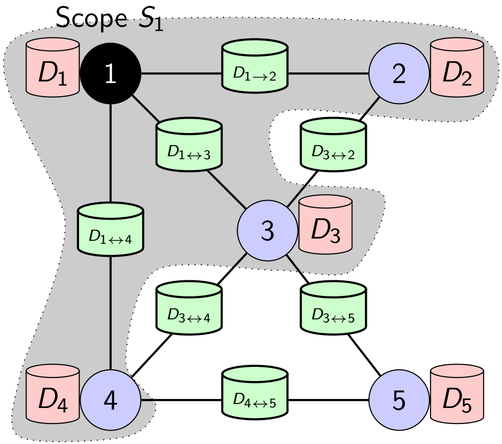
- **Graph** $G = (V, E, D)$
  - D_i is user defined data associated with vertex i
  - $D_{i \to j}$ is user defined data associated with each edge $i \to j$
    - represents model parameters, algorithm state, and statistical data
- Example: Social network
  - Vertex Data D_i
    - User profile
    - Current interests estimates
  - Edge Data $D_{i \to j}$
    - Relationship (friend, relative, classmate,...)



Update functions

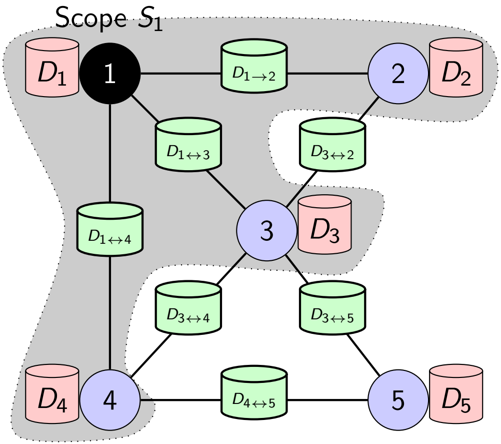


Update functions



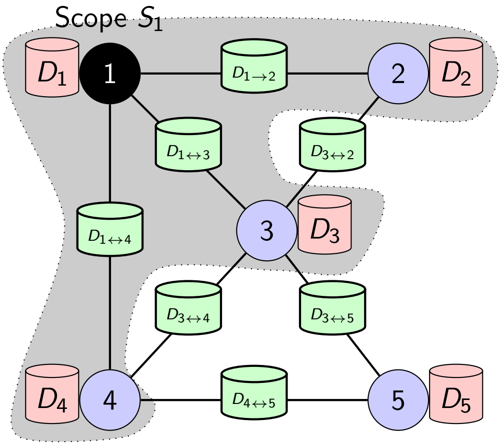
Update functions

- **Update function**
  $f(v, S_v) \rightarrow (S_v, T)$
  - User defined function
  - When applied to a vertex v, transforms the data in the scope S_v
  - Schedules the future execution of the update functions on other vertices T



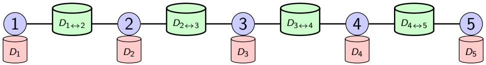
Update functions

- **Update function**
  $f(v, S_v) \to (S_v, T)$
  - User defined function
  - When applied to a vertex v, transforms the data in the scope S_v
  - Schedules the future execution of the update functions on other vertices T
- PageRank example
  - f computes weighted sum of the current ranks of neighbors, update its own rank
  - neighbors are scheduled for update if required



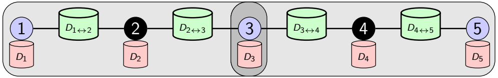
Data consistency

- Scopes often overlap



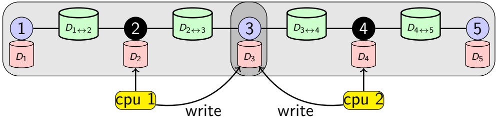
Data consistency

- Scopes often overlap

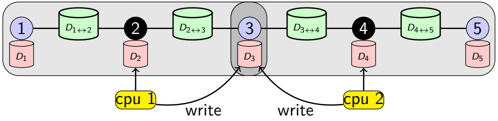


Data consistency

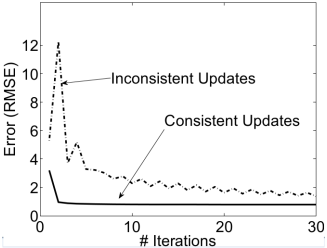
- Scopes often overlap
- Simultaneous execution of update functions can lead to race-conditions
  - may lead to inconsistent data
  - may lead to corrupt data



Data consistency



- Scopes often overlap
- Simultaneous execution of update functions can lead to race-conditions
  - may lead to inconsistent data
  - may lead to corrupt data
- MLDM algorithms perform better with strict consistency
  - Example: alternating least squares

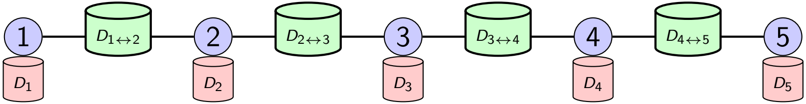


Data consistency

- GraphLab provides three consistency models
  - **Full consistency** $>$ **Edge consistency** $>$ **Vertex consistency**
- Choice determines **how much computations can overlap?**

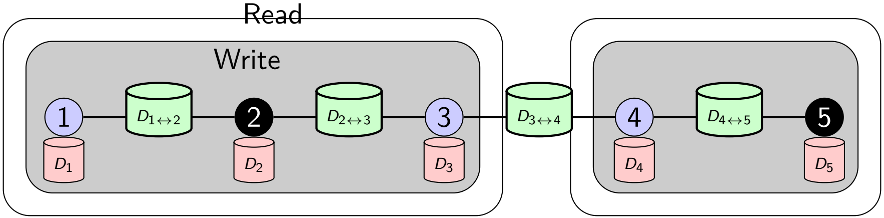
Full consistency

- Scopes of concurrently executing update functions do not overlap
- Update functions have **complete read/write access** to entire scope
- Concurrently executing update functions must be at least two vertices apart



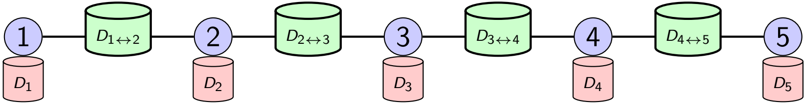
Full consistency

- Scopes of concurrently executing update functions do not overlap
- Update functions have **complete read/write access** to entire scope
- Concurrently executing update functions must be at least two vertices apart



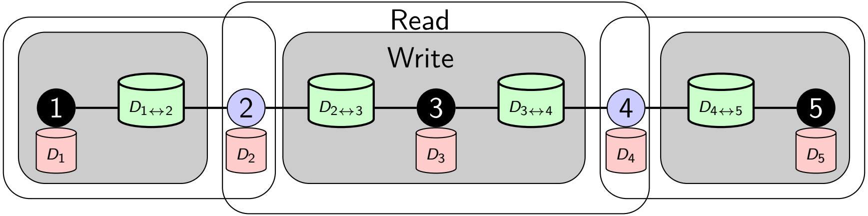
Edge consistency

- Allows update functions with slightly overlapping scopes
- Update function has **exclusive read/write access to its vertex and adjacent edges**
- Update function has **read only access to adjacent vertices**



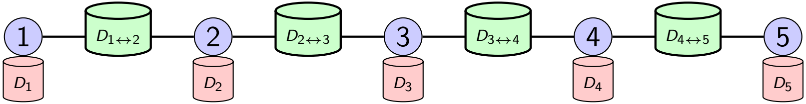
Edge consistency

- Allows update functions with slightly overlapping scopes
- Update function has **exclusive read/write access to its vertex and adjacent edges**
- Update function has **read only access to adjacent vertices**



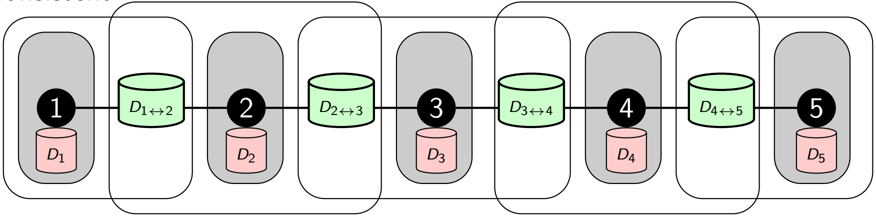
Vertex consistency

- Allows all update functions to run in parallel
- Update function has **write access to central vertex data**
- Update function has **read only access to its adjacent edges**
- Least consistent

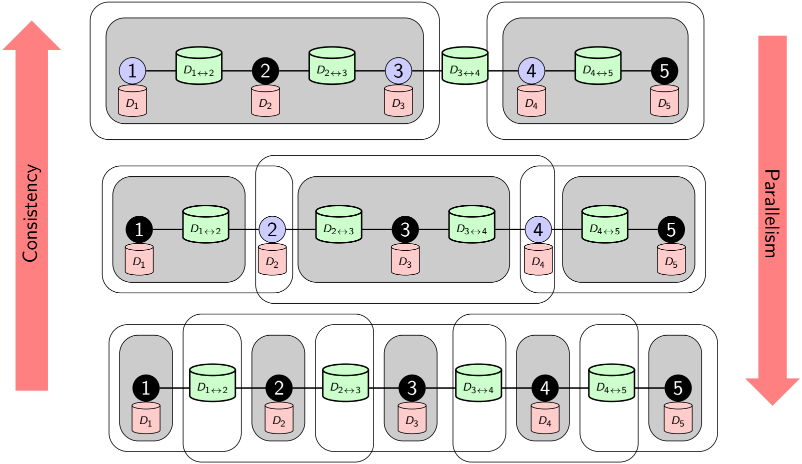


Vertex consistency

- Allows all update functions to run in parallel
- Update function has **write access to central vertex data**
- Update function has **read only access to its adjacent edges**
- Least consistent



Consistency and parallelism



Consistency and correctness

- Choice of consistency model impacts correctness
- GraphLab provides **sequential consistency**
  - For every parallel execution, there exists a sequential execution with same results
- Sequential consistency is guaranteed, if
  - Full consistency model is used
  - Edge consistency model is used and update functions do not modify data in adjacent vertices
  - Vertex consistency model is used and update functions only access local vertex data

Consistency and correctness

- Choice of consistency model impacts correctness
- GraphLab provides **sequential consistency**
  - For every parallel execution, there exists a sequential execution with same results
- Sequential consistency is guaranteed, if
  - Full consistency model is used
  - Edge consistency model is used and update functions do not modify data in adjacent vertices
  - Vertex consistency model is used and update functions only access local vertex data
- Q: Which consistency model would you use for PageRank?

Outline

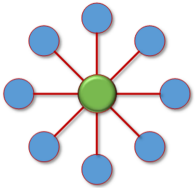
1. Background
2. Vertex-centric approaches
   - Pregel/Giraph
   - GraphLab
   - PowerGraph
3. Discussion
   - Dataflow engines
   - Summary

Handling high-degree vertices

- Existing distributed graph computations systems perform poorly on **natural graphs** (Gonzalez et al. OSDI'12)

Handling high-degree vertices

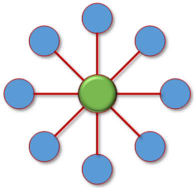
- Existing distributed graph computations systems perform poorly on **natural graphs** (Gonzalez et al. OSDI'12)
- Natural graphs have **highly skewed degree distributions**
  - Problems with **high degree vertices**
    - limited single-machine resources
    - work imbalance
    - sequential computation
    - communication cost
    - graph partitioning



Handling high-degree vertices

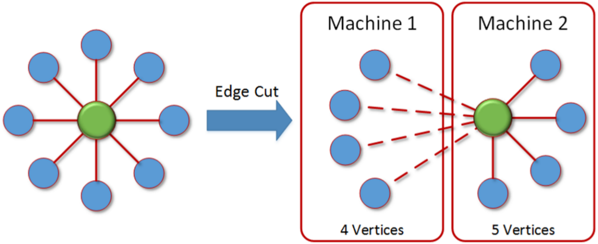
- Existing distributed graph computations systems perform poorly on **natural graphs** (Gonzalez et al. OSDI'12)

- Natural graphs have **highly skewed degree distributions**
  - Problems with **high degree vertices**
    - limited single-machine resources
    - work imbalance
    - sequential computation
    - communication cost
    - graph partitioning



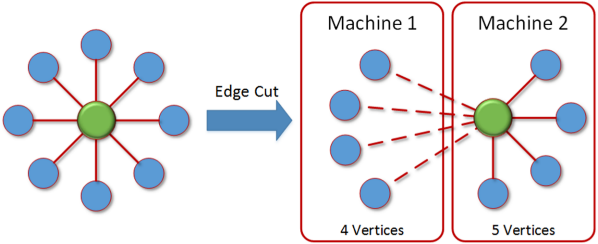
- GraphLab 2.0 (PowerGraph)
  - **G**ather- **A**pply - **S**catter paradigm to allow for pre-aggregation in vertex updates
  - **two-dimensional partitioning** of the data graph ("vertex cut")

Graph partitioning: edge cut vs vertex cut

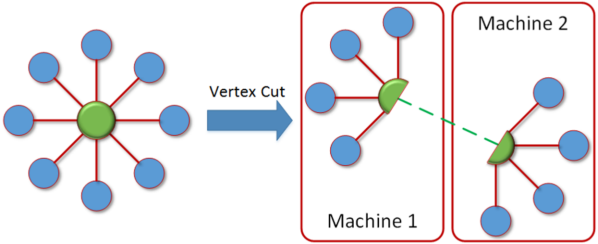


- Edge cut
- Used by Pregel/GraphLab
- Evenly assign vertices to machines

Graph partitioning: edge cut vs vertex cut

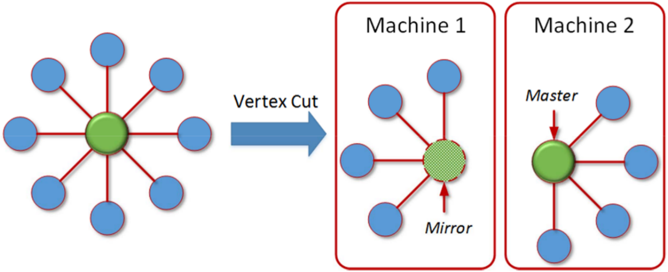


- Edge cut
- Used by Pregel/GraphLab
- Evenly assign vertices to machines



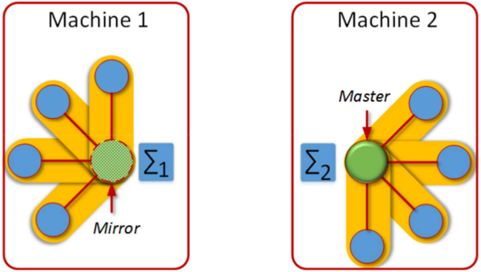
- Vertex cut
- Used by PowerGraph
- Evenly assign edges to machines

G-A-S: example



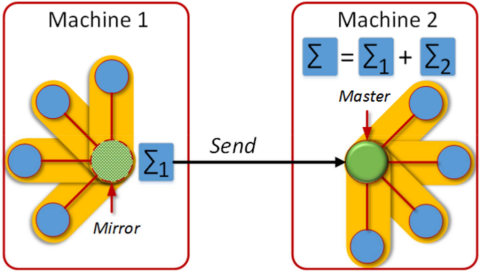
G-A-S: example (Gather phase)

- Compute partial sums on each machine



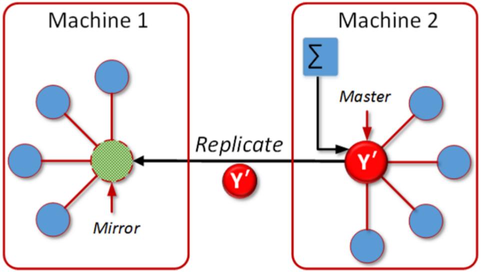
G-A-S: example (Gather phase)

- Compute partial sums on each machine
- Master machine computes total sum



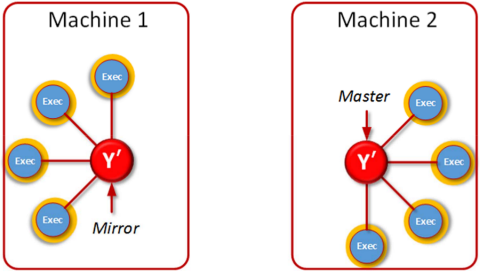
G-A-S: example (Apply phase)

- Apply accumulated value to center vertex
- Replicate value to mirrors



G-A-S: example (Scatter phase)

- Update adjacent edges and vertices
- Initiate neighboring vertex-programs if necessary



Synchronous vs asynchronous graph processing

Synchronous (BSP)

- Computation in phases
  - all vertices participate
  - all messages are sent
- Simple to build
  - no race conditions
  - barrier guarantees consistency
  - simple fault tolerance
- Slow convergence
  - unsuitable for many MLDM problems

Asynchronous (GraphLab)

- Vertices see latest information from neighbors
- Hard to build
  - Race conditions all the time
  - fault tolerance more complex
  - termination detection
- Fast convergence
  - suitable for MLDM problems

Outline

Outline

1. Background
2. Vertex-centric approaches
   - Pregel/Giraph
   - GraphLab
   - PowerGraph
3. Discussion
   - Dataflow engines
   - Summary

Vertex-centric processin with dataflow engines

Spark

- **GraphX** component for graphs and graph-parallel computations
- Extends Spark RDD with **Property Graph** abstraction (cf. data graph of GraphLab)
- Provides several fundamental graph specific operators
  - e.g., subgraph, joinVertices, aggregateMessages
- Pregel API

SSSP using the GraphX Pregel API

```
// A graph with edge attributes containing distances
val graph: Graph[Long, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)

val sourceId: VertexId = 23 // Source vertex

// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) =>
    if (id == sourceId) 0.0 else Double.PositiveInfinity)

val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => { // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  }, (a, b) => math.min(a, b) // Merge Message
)
println(sssp.vertices.collect.mkString("\n"))
```

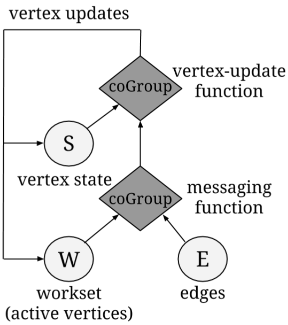
Vertex-centric processing with dataflow engines

Flink

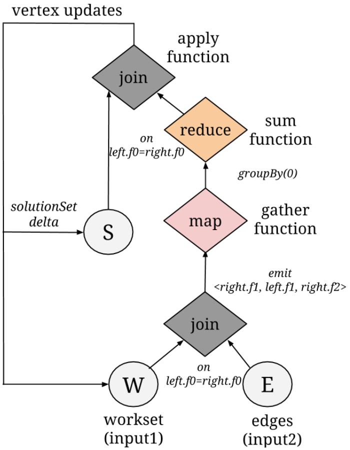
- **Gelly API**: Flink's graph-processing API and library
- Graph representation: DataSet of vertices & DataSet of edges (cf. Data graph of GraphLab)
- Provides several transformations and utilities suitable for graphs
  - e.g., mapVertices, joinWithVertices, mapEdges, filterOnEdges, etc
- Supports Pregel like vertex-centric as well G-A-S computation model

Iterative graph processing with Gelly

- Supports Pregel like vertex-centric as well G-A-S computation
- Internally Flink's delta iteration



Vertex-centric



G-A-S

SSSP using Gelly's vertex centric API

```
DataSet<Edge<Long, Double>> edges = getEdgesDataSet(env);
Graph<Long, Double, Double> graph = Graph.fromDataSet(edges, new InitVertices(), env);
// Execute the vertex-centric iteration
Graph<Long, Double, Double> result = graph.runVertexCentricIteration(
    new SSSPComputeFunction(srcVertexId), new SSSPCombiner(), maxIterations);
// Extract the vertices as the result
DataSet<Vertex<Long, Double>> singleSourceShortestPaths = result.getVertices();
singleSourceShortestPaths.print();
```

```
/* SSSPComputeFunction */
double minDistance = (vertex.getId().equals(srcId)) ? 0d :
        Double.POSITIVE_INFINITY;
for (Double msg : messages)
  minDistance = Math.min(minDistance, msg);
if (minDistance < vertex.getValue()) {
  setNewVertexValue(minDistance);
  for (Edge<Long, Double> e: getEdges()) {
    sendMessageTo(e.getTarget(), minDistance + e.
          getValue());
  }
}
```

```
/* SSSPComiber */
double minMessage = Double.POSITIVE_INFINITY;

for (Double msg: messages) {
  minMessage = Math.min(minMessage, msg);
}
sendCombinedMessage(minMessage);
```

Outline

1. Background
2. Vertex-centric approaches
   - Pregel/Giraph
   - GraphLab
   - PowerGraph
3. Discussion
   - Dataflow engines
   - Summary

Summary

- Google Pregel introduces **vertex-centric BSP** as alternative
- **think like a vertex**: distributed graph processing based on vertex update functions and messaging
- Giraph
  - Open source implementation of Google Pregel
- GraphLab
  - Extends Vertex-centric approach to support **asynchronous updates**
  - Data graph abstraction
  - Update function
  - Sequential consistency
- Powergraph
  - GraphLab 2.0
  - Counter measures for **natural graphs** (high degree vertices)
  - **Vertex cut** instead of edge cut
  - **G-A-S** paradigm