UMD DATA605 - Big Data Systems

# Graph Data Processing Giraph, GraphX

**Instructor**: Dr. GP Saggese - gsaggese@umd.edu**
**TAs**: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
**v1.1**

SCIENCE ACADEMY

# Options for Processing Graph Data

- Write your own programs
  - Extract the relevant data, and construct an in-memory graph
  - Different storage options help to different degrees with this
- Write queries in a declarative language
  - Works for some classes of graph queries/tasks
  - E.g., cypher for Neo4j
- Use a general-purpose distributed programming framework
  - E.g.: Hadoop or Spark
  - Hard to program many graph analysis tasks this way
- Use a graph-specific programming framework
  - Goal is to simplify writing graph analysis tasks, and scale them to very large volumes at the same time
  - E.g., Giraph or GraphX

# Option 2: Declarative Interfaces

- No consensus on declarative, high-level languages (like SQL) for either querying or for analysis
  - Too much variety in the types of queries/analysis tasks
  - Makes it hard to find and exploit commonalities
- Some limited, but useful solutions:
  - XQuery for XML
    - Limited to tree-structured data
  - SPARQL for RDF
    - Standardized query language, but limited functionality
  - Cypher by Neo4j
  - Datalog-based frameworks for specifying analysis tasks
    - Many prototypes, typically specific to some analysis task

# Option 3: MapReduce

- A popular option for (batch) processing very large datasets, as we know
  - More specifically: Hadoop or Spark
- Two key advantages:
  - Scalability without worrying about scheduling, distributed execution, fault tolerance, and so on. . .
  - Relatively simple programming framework
- Disadvantages:
  - Hard to use directly for graph analysis tasks
  - Each "traversal" effectively requires a new "map-reduce" phase
    - Hadoop framework not ideal for large numbers of phases, Spark is better, though
- However, much work on showing how different graph analysis tasks can be done using MapReduce

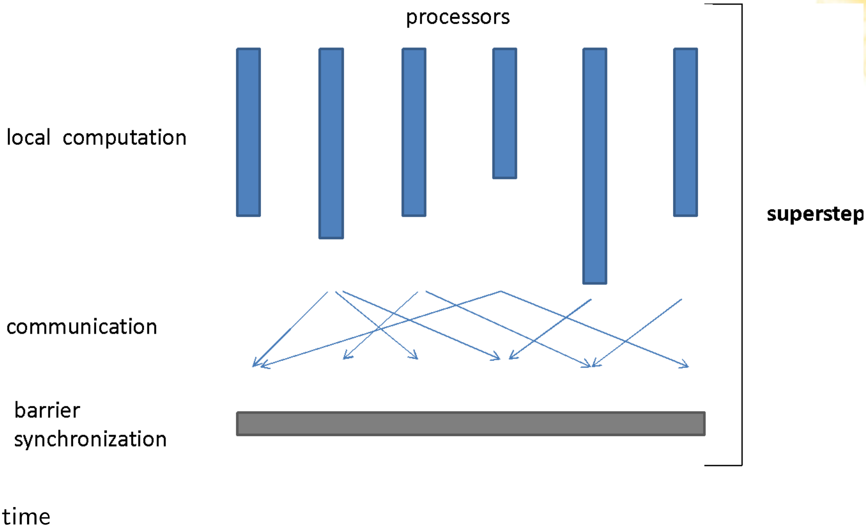
# Option 3: MapReduce

- Disadvantages:
  - Hard to use this for graph analysis task
  - Each "traversal" effectively requires a new "map-reduce" phase
    - Each job is execute $N$ times
  - Hadoop framework not ideal for large numbers of phases (even with YARN)
  - Not efficient – too much redundant work
    - Mappers send PR values and structure of graph
  - In PageRank example: repeated reading and parsing of the inputs for each iteration
    - Extensive I/O at input, shuffle/sort, output

# Option 4: Graph Programming Frameworks

- Frameworks (analogous to MapReduce) proposed for analyzing large volumes of graph data
  - An attempt at addressing limitations of MapReduce
  - Most are *vertex-centric*
    - Programs written from the point of view of a vertex
  - Most based on message passing between nodes
- Pregel: original framework proposed by Google
  - Based on "Bulk Synchronous Parallel" (BSP) model
- Giraph: an open-source implementation of Pregel built on top of Hadoop
- GraphLab: asynchronous execution
- GraphX: built on top of Spark

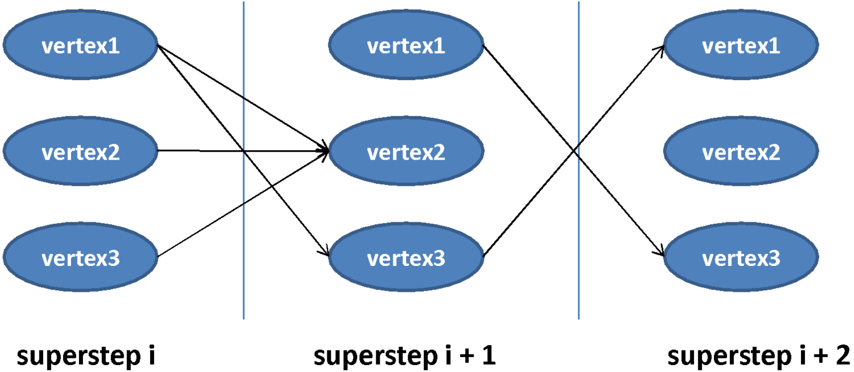
# Bulk Synchronous Parallel (BSP)



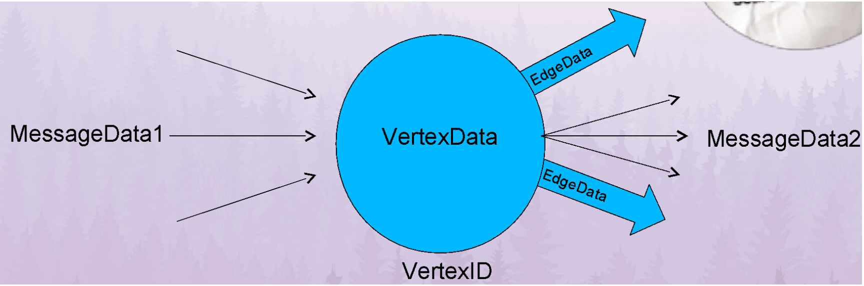

SCIENCE ACADEMY

# Vertex-centric BSP

- Each vertex has an id, a value, a list of its adjacent vertex ids and the corresponding edge values
- Each vertex is invoked in each superstep, can recompute its value and send messages to other vertices, which are delivered over superstep barriers
- Advanced features : termination votes, combiners, aggregators, topology mutations



# A vertex view



# Designed for iterations

- Stateful (in-memory)
  - Keep all data in memory if possible
- Only intermediate values (messages) sent
  - To communicate with other vertices
- Hits the disk at input, output, checkpoint
- Can go out-of-core
  - If the data does not fit into memory

# Graph modeling in Giraph

- BasicComputation< I extends WritableComparable, // VertexID – vertex ref V extends Writable, // VertexData – a vertex datum E extends Writable, // EdgeData – an edge label M extends Writable> // MessageData-– message payload

SCIENCE ACADEMY

# Giraph "Hello World"

```
public class GiraphHelloWorld extends BasicComputation<IntWritable,
IntWritable, NullWritable, NullWritable> { public void
compute(Vertex<IntWritable, IntWritable, NullWritable> vertex, Iterable
messages) { System.out.println(“Hello world from the:” + vertex.getId() + ” :
“); for (Edge<IntWritable, NullWritable> e : vertex.getEdges()) {
System.out.println(” ” + e.getTargetVertexId()); } System.out.println(““);
vertex.voteToHalt(); }
```

# Example: Ping neighbors

```
public void compute(Vertex<Text, DoubleWritable, DoubleWritable> vertex,
Iterable ms ){ if (getSuperstep() == 0) { sendMessageToAllEdges(vertex,
vertex.getId()); } else { for (Text m : ms) { if (vertex.getEdgeValue(m) ==
null) { vertex.addEdge(EdgeFactory.create(m, SYNTHETIC_EDGE)); } } }
vertex.voteToHalt(); }
```

# Giraph PageRank Example

public class PageRankComputation extends BasicComputation<IntWritable, FloatWritable, NullWritable, FloatWritable> {