Programming Models and Tools for Distributed Graph Processing

Vasia Kalavri
kalavriv@inf.ethz.ch

31st British International Conference on Databases
10 July 2017, London, UK
ABOUT ME

» Postdoctoral Fellow at ETH Zürich
  » Systems Group: https://www.systems.ethz.ch/

» PMC member of Apache Flink

» Research interests
  » Large-scale graph processing
  » Streaming dataflow engines

» Current project:
  » Predictive datacenter analytics and management
  » Strymon: http://strymon.systems.ethz.ch/
TUTORIAL OUTLINE

- Distributed Graph Processing (DGP)
  - when do we need distribution?
  - misconceptions and truths

- Specialized Models for DGP
  - execution semantics
  - user interfaces
  - performance issues

- General-Purpose Models for DGP

- Recap
THIS TUTORIAL IS NOT ABOUT

- Graph databases
- RDF stores
- Single-node systems
- Shared-memory systems
- Performance comparison of tools
MODELING THE WORLD AS A GRAPH

Social networks

Transportation networks

Actor-movie networks

The web
ANALYZING GRAPHS

What’s the cheapest way to reach Zurich from London through Berlin?

If you like “Inside job” you might also like “The Bourne Identity”

These are the top-10 relevant results for the search term “graph”
BASICS

"node" or "vertex"

"edge"
“node” or “vertex”

“edge”

5 is out-neighbor
1, 2 are in-neighbors
3 is a 2-hop neighbor
4 has in-degree = 2 and out-degree = 1
DISTRIBUTED GRAPH PROCESSING

- Shared-nothing memory model
- Distributed algorithms for analysis
- Graph partitioning
GRAPH PARTITIONING

- Communication usually “flows” along edges
- Minimize communication while balancing the computation load
- Many graphs have skewed degree distributions
WHEN DO YOU NEED DISTRIBUTED GRAPH PROCESSING?
MISCONCEPTION #1

MY GRAPH IS SO BIG, IT DOESN'T FIT IN A SINGLE MACHINE
A SOCIAL NETWORK

WTF: The Who to Follow Service at Twitter

Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, Reza Zadeh

Twitter, Inc.
@pankaj @ashishgoel @lintool @aneeshs @dongwang218 @reza_zadeh

2 billion users, one could storing each vertex id as a 32-bit (signed) integer, in which case each edge would require eight bytes. On a machine with 72 GB memory, we could reasonably expect to handle graphs with approximately eight billion edges: 64 GB to hold the entire graph in memory, and 8 GB for the operating system, the graph processing engine, and memory for actually executing graph algorithms. Of course, storing the source and destination vertices of each edge in the manner described above is quite wasteful; the
Naive Who(m) to Follow:

- compute a friends-of-friends list per user
- exclude existing friends
- rank by common connections
DON’T JUST CONSIDER YOUR INPUT GRAPH SIZE. INTERMEDIATE DATA MATTERS TOO!
MISCONCEPTION #2

DISTRIBUTED PROCESSING IS ALWAYS FASTER THAN SINGLE-NODE

Data Science Rockstar
# Scalability! But at what COST?

Frank McSherry  
Unaffiliated  

Michael Isard  
Unaffiliated*  

Derek G. Murray  
Unaffiliated†

<table>
<thead>
<tr>
<th>scalable system</th>
<th>cores</th>
<th>twitter</th>
<th>uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratosphere [8]</td>
<td>16</td>
<td>950s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [21]</td>
<td>16</td>
<td>1159s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [10]</td>
<td>128</td>
<td>1784s</td>
<td>≥ 8000s</td>
</tr>
<tr>
<td>Giraph [10]</td>
<td>128</td>
<td>200s</td>
<td>≥ 8000s</td>
</tr>
<tr>
<td>GraphLab [10]</td>
<td>128</td>
<td>242s</td>
<td>714s</td>
</tr>
<tr>
<td>GraphX [10]</td>
<td>128</td>
<td>251s</td>
<td>800s</td>
</tr>
</tbody>
</table>
Scalability! But at what COST?

Frank McSherry   Michael Isard   Derek G. Murray  
Unaffiliated     Unaffiliated*  Unaffiliated†

<table>
<thead>
<tr>
<th>scalable system</th>
<th>cores</th>
<th>twitter</th>
<th>uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratosphere [8]</td>
<td>16</td>
<td>950s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [21]</td>
<td>16</td>
<td>1159s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [10]</td>
<td>128</td>
<td>1784s</td>
<td>≥ 8000s</td>
</tr>
<tr>
<td>Giraph [10]</td>
<td>128</td>
<td>200s</td>
<td>≥ 8000s</td>
</tr>
<tr>
<td>GraphLab [10]</td>
<td>128</td>
<td>242s</td>
<td>714s</td>
</tr>
<tr>
<td>GraphX [10]</td>
<td>128</td>
<td>251s</td>
<td>800s</td>
</tr>
<tr>
<td>Single thread (SSD)</td>
<td>1</td>
<td>153s</td>
<td>417s</td>
</tr>
</tbody>
</table>
GRAPHS DON'T APPEAR OUT OF THIN AIR
GRAPHS DON'T APPEAR OUT OF THIN AIR
WHEN DO YOU NEED DISTRIBUTED GRAPH PROCESSING?

▸ When you do have **really big** graphs

▸ When the **intermediate data** of your computation can be very large

▸ When your data is **already distributed**

▸ When you have a **distributed data pipeline**
HOW DO WE EXPRESS A DISTRIBUTED GRAPH ANALYSIS TASK?
GRAPH APPLICATIONS ARE DIVERSE

- Iterative value propagation
  - PageRank, Connected Components, Label Propagation

- Traversals and path exploration
  - Shortest paths, centrality measures

- Ego-network analysis
  - Personalized recommendations

- Pattern mining
  - Finding frequent subgraphs
SPECIALIZED

PROGRAMMING MODELS FOR

DISTRIBUTED GRAPH PROCESSING
RECENT DISTRIBUTED GRAPH PROCESSING HISTORY

2004  MapReduce
2009  Pregel
2010  Giraph++
2012  PowerGraph
2013  Arctic++
2014  Tinkerpop
2015  Arabesque

- Graph Traversals
- Pattern Matching
- Iterative value propagation
- Ego-network analysis

Signal-Collect
Pegasus
NScale
HIGH-LEVEL PROGRAMMING MODELS

- Hide distribution complexity behind an *abstraction*
  - data partitioning
  - data representation
  - communication mechanisms
- Programmers can focus on the *logic* of their application
  - logical view of graph data
  - a set of methods to read, write, and communicate across views
**partitioned view:**
the state exposed to the API methods by the programming abstraction
VERTEX-CENTRIC: THINK LIKE A VERTEX

- Express the computation from the view of a single vertex
- Vertices communicate through messages

Malewicz, Grzegorz, et al.
*Pregel: a system for large-scale graph processing.*
*ACM SIGMOD, 2010.*
The partitioned view consists of a vertex, its out-neighbors, an inbox, and an outbox.
(V_{i+1}, outbox) ← compute(V_i, inbox)
**Input**: directed graph $G=(V,E)$

$\text{activeVertices} \leftarrow V$

$\text{superstep} \leftarrow 0$

while $\text{activeVertices} \neq \emptyset$ do
  for $v \in \text{activeVertices}$ do
    $\text{inbox}_v \leftarrow \text{receiveMessages}(v)$
    $\text{outbox}_v = \text{compute}(\text{inbox}_v)$
  end for
  $\text{superstep} \leftarrow \text{superstep} + 1$
end while
VC INTERFACE

void compute(Iterator[M] messages);
VV getValue();
void setValue(VV newValue);
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();
void voteToHalt();
PAGERANK: THE WORD COUNT OF GRAPH PROCESSING

<table>
<thead>
<tr>
<th>VertexID</th>
<th>Out-degree</th>
<th>Transition Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1/3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
PR(3) = 0.5*PR(1) + 0.33*PR(4) + PR(5)
void compute(messages):
    sum = 0.0
    for (m <- messages) do
        sum = sum + m
    end for
    setValue(0.15/numVertices + 0.85*sum)
    for (edge <- getOutEdges()) do
        sendMessageTo(
            edge.target(), getValue() / numEdges)
    end for
VC ANTI-PATTERNS

- Non-iterative algorithms
  - Superstep execution
- Non-local state access
  - Propagate a message in 2 supersteps to access 2-hop neighborhood
- Communication with in-neighbors
  - Insert opposite-direction edges to regard in-neighbors as out-neighbors
TRIANGLE COUNTING

- A vertex needs to know whether there is an edge between its neighbors
- It has to detect this through messages
- It takes 3 supersteps to propagate a message along the triangle’s edges
TRIANGLE COUNTING

I'm your neighbor

superstep #1
“3” is my neighbor

superstep #2
“3” is my neighbor, too. It’s a triangle!

superstep #3
PERFORMANCE ISSUES

- Skewed degree distribution
  - high communication load
  - high memory requirements
- Synchronization
- Asymmetrical convergence

- smart partitioning
  - copy high-degree vertices
  - split supersteps into several sub-supersteps
- support asynchronous and semi-synchronous execution
- monitor the “active” portion of the graph
SIGNAL–COLLECT (SCATTER–GATHER)

- Express the computation from the view of a single vertex
- Vertices send their values as signals to their in-neighbors and collect signals from their out-neighbors to compute their new values

**SIGNAL-COLLECT (SCATTER-GATHER)**

Signal

Superstep $i$

Collect

Superstep $i+1$

\begin{align*}
\text{outbox} &\leftarrow \text{signal}(V_i) \\
V_{i+1} &\leftarrow \text{collect}(\text{inbox})
\end{align*}

No concurrent access to inbox and outbox
**SIGNAL-COLLECT SEMANTICS**

**Input:** directed graph $G=(V,E)$

$activeVertices \leftarrow V$

$superstep \leftarrow 0$

while $activeVertices \neq \emptyset$ do

  for $v \in activeVertices$ do

    $outbox_v \leftarrow signal(v)$

    $newState \leftarrow collect(inbox_v, v.state)$

    if $newState \neq v.state$ do

      $v.state = newState$

      $activeVertices$

    end if

  end for

  $superstep \leftarrow superstep + 1$

end while
void signal();
VV getValue();
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();

void collect(Iterator[M] messages);
void setValue(VV newValue);
VV getValue();
int superstep();
void signal():
    for (edge <- getOutEdges()) do
        sendMessageTo(
            edge.target(), getValue() / numEdges)
    end for

void collect(messages):
    sum = 0.0
    for (m <- messages) do
        sum = sum + m
    end for

setValue(0.15 / numVertices + 0.85 * sum)

update vertex rank
sum up received messages
distribute rank to neighbors

SIGNAL-COLLECT PAGERANK
SIGNAL–COLLECT ANTI-PATTERNS

- Algorithms that require concurrent access to the inbox and outbox
  - signal has **read-access** to the vertex state and **write-access** to the outbox
  - collect has **read-access** to the inbox and **write-access** to the state
- Vertices cannot generate messages and update their states in the same phase
  - e.g. decide whether to propagate a message based on its content
  - **workaround**: store the message in the vertex-value
GATHER-SUM-APPLY-SCATTER (GSA)

- Express the computation from the view of a single vertex
- Vertices produce a message per edge, gather and aggregate partial results, update their state with the final aggregate

GATHER-SUM-APPLY SUPERSTEPS

Superstep i
- Gather
- Sum
- Apply

Superstep i+1
- Gather

Message generation is parallelized over the edges!
Input: directed graph $G = (V, E)$

$a_v \leftarrow \text{empty}$

for $v \in V$ do
  for $n \in v.\text{inNeighbors}$ do
    $a_v \leftarrow \text{sum}(a_v, \text{gather}(S_v, S_{(v,n)}, S_n))$
  end for
  $S_v \leftarrow \text{apply}(S_v, a_v)$
  $S(v,n) \leftarrow \text{scatter}(S_v, S_{(v,n)}, S_n)$
end for
GSA INTERFACE

T **gather**(VV sourceV, EV edgeV, VV targetV);

T **sum**(T left, T right);

VV **apply**(VV value, T sum);

EV **scatter**(VV newV, EV edgeV, VV oldV);
double gather(source, edge, target):
    return target.value() / target.numEdges()

double sum(rank1, rank2):
    return rank1 + rank2

double apply(sum, currentRank):
    return 0.15 + 0.85*sum
## VC VS. SIGNAL-COLLECT VS. GSA

<table>
<thead>
<tr>
<th></th>
<th>Update Function Properties</th>
<th>Update Function Logic</th>
<th>Communication Scope</th>
<th>Communication Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex-Centric</strong></td>
<td>arbitrary</td>
<td>arbitrary</td>
<td>any vertex</td>
<td>arbitrary</td>
</tr>
<tr>
<td><strong>Signal-Collect</strong></td>
<td>arbitrary</td>
<td>based on received messaged</td>
<td>any vertex</td>
<td>based on vertex state</td>
</tr>
<tr>
<td><strong>GSA</strong></td>
<td>associative &amp; commutative</td>
<td>based on neighbors’ values</td>
<td>neighborhood</td>
<td>based on vertex state</td>
</tr>
</tbody>
</table>
PROBLEMS WITH VERTEX-PARALLEL MODELS

- Excessive communication
- Worker load imbalance
- Global Synchronization
- High memory requirements
  - inbox/outbox can grow too large
  - overhead for low-degree vertices in GSA
PARTITION-CENTRIC

- Express the computation from the view of a partition
- Differentiate between internal and boundary vertices

Tian, Yuanyuan, et al.  
"From think like a vertex to think like a graph."  
THINK LIKE A (SUB)GRAPH

- compute() on the entire partition
- Information flows freely inside each partition
- Network communication between partitions, not vertices
- 2 in an **internal** vertex in P1
- 1, 4 are **boundary** vertices
VERTEX-CENTRIC CONNECTED COMPONENTS

- Propagate the minimum value through the graph
- In each superstep, the value propagates one hop
- Requires diameter + 1 supersets to converge
PARTITION-CENTRIC CONNECTED COMPONENTS

- In each superstep, the value propagates throughout each subgraph
- Communication between partitions only
- Fewer supersteps until convergence
PARTITION-CENTRIC INTERFACE

void compute();
void sendMessageTo(I target, M message);
int superstep();
void voteToHalt();

boolean containsVertex(I id);
boolean isInternalVertex(I id);
boolean isBoundaryVertex(I id);
Collection getInternalVertices();
Collection getBoundaryVertices();
Collection getAllVertices();
void compute():
    if superstep() == 0 then
        for v ∈ getAllVertices() do
            v.getValue().pr = 0
            v.getValue().delta = 0
        end for
    end if
    for iv ∈ internalVertices() do
        outEdges = iv.getNumOutEdges()
        if superstep() == 0 then
            iv.getValue().delta+ = 0.15
        end if
        iv.getValue().delta+ = iv.getMessages()
        if iv.getValue().delta > 0 then
            iv.getValue().pr+ = iv.getValue().delta
            u = 0.85 * iv.getValue().delta/outEdges
            while iv.iterator.hasNext() do
                neighbor = getVertex(iv.iterator().next())
                neighbor.getValue().delta+ = u
            end while
        end if
        iv.getValue().delta = 0
    end for
    for bv ∈ boundaryVertices() do
        bvID = bv.getVertexId()
        if bv.getValue().delta > 0 then
            sendMessageTo(bvID, bv.getValue().delta)
        end if
    end for
end for
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
  end if
  for iv ∈ internalVertices() do
    outEdges = iv.getNumOutEdges()
    if superstep() == 0 then
      iv.getValue().delta+ = 0.15
    end if
    iv.getValue().delta+ = iv.getMessages()
    if iv.getValue().delta > 0 then
      iv.getValue().pr+ = iv.getValue().delta
      u = 0.85 * iv.getValue().delta/outEdges
      while iv.iterator.hasNext() do
        neighbor = getVertex(iv.iterator().next())
        neighbor.getValue().delta+ = u
      end while
    end if
    iv.getValue().delta = 0
  end for
  for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
      sendMessageTo(bvID, bv.getValue().delta)
    end if
  end for
end if
void compute():
    if superstep() == 0 then
        for v ∈ getAllVertices() do
            v.getValue().pr = 0
            v.getValue().delta = 0
        end for
    end if
    for iv ∈ internalVertices() do
        outEdges = iv.getNumOutEdges()
        if superstep() == 0 then
            iv.getValue().delta+ = 0.15
        end if
        iv.getValue().delta+ = iv.getMessages()
        if iv.getValue().delta > 0 then
            iv.getValue().pr+ = iv.getValue().delta
            u = 0.85 * iv.getValue().delta/outEdges
            while iv.iterator.hasNext() do
                neighbor = getVertex(iv.iterator().next())
                neighbor.getValue().delta+ = u
            end while
            iv.getValue().delta = 0
        end if
    end for
    for bv ∈ boundaryVertices() do
        bvID = bv.getVertexId()
        if bv.getValue().delta > 0 then
            sendMessageTo(bvID, bv.getValue().delta)
        end if
        bv.getValue().delta = 0
    end for
PARTITION-CENTRIC PAGERANK

void compute():
    if superstep() == 0 then
        for v ∈ getAllVertices() do
            v.getValue().pr = 0
            v.getValue().delta = 0
        end for
    end if
    for iv ∈ internalVertices() do
        outEdges = iv.getNumOutEdges()
        if superstep() == 0 then
            iv.getValue().delta+ = 0.15
        end if
        iv.getValue().delta+ = iv.getMessages()
        if iv.getValue().delta > 0 then
            iv.getValue().pr+ = iv.getValue().delta
            u = 0.85 * iv.getValue().delta/outEdges
            while iv.iterator.hasNext() do
                neighbor = getVertex(iv.iterator().next())
                neighbor.getValue().delta+ = u
            end while
        end if
        iv.getValue().delta = 0
    end for
    for bv ∈ boundaryVertices() do
        bvID = bv.getVertexId()
        if bv.getValue().delta > 0 then
            sendMessageTo(bvID, bv.getValue().delta)
            bv.getValue().delta = 0
        end if
    end for

Boundary Vertices

for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
        sendMessageTo(bvID, bv.getValue().delta)
        bv.getValue().delta = 0
    end if
end for
GENERAL-PURPOSE PROGRAMMING MODELS FOR DISTRIBUTED GRAPH PROCESSING
- Partition by rows, columns, blocks
- Efficient compressed-row/column representations
- Algorithms expressed as vector-matrix multiplications
BREADTH-FIRST SEARCH

1  2  3  4  5
1  0  0  1  1  0
2  1  0  0  1  0
3  0  0  0  0  0
4  0  1  1  0  1
5  0  0  1  0  0
BREADTH-FIRST SEARCH

\[
\begin{array}{c|c|c|c|c|c}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[\times\]

\[
\begin{array}{c|c|c|c|c|c}
 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

=  
\[
\begin{array}{c|c|c|c|c|c}
 & 1 & 1 & 1 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]
BREADTH-FIRST SEARCH

\[
\begin{array}{ccccc}
& 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

= \[0 \ 1 \ 1 \ 0 \ 0\]
BREADTH-FIRST SEARCH

\[
\begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 \\
5 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\]
DISTRIBUTED DATAFLOWS

- Dataflow programs are directed graphs, where nodes are data-parallel operators (computations) and edges represent data dependencies
- e.g.: Apache Spark, Apache Flink, Naiad
- Graphs are represented with 2 datasets: vertices and edges
for (i <- 1 to iters) {
    val contribs =
        links.join(ranks).values
            .flatMap {
                case (urls, rank) =>
                    val size = urls.size
                    urls.map(url => (url, rank / size))
            }
    ranks = contribs.reduceByKey(_+_)
        .mapValues(0.15 + 0.85 * _)
}
val output = ranks.collect()
HIGH-LEVEL GRAPH APIS ON DATA FLOWS


GELLY: THE APACHE FLINK GRAPH API

- Java & Scala Graph APIs on top of Flink’s DataSet API
WHY GRAPH PROCESSING WITH APACHE FLINK?

▸ Native Iteration Operators
▸ DataSet Optimizations
▸ Ecosystem Integration
▸ Memory Management and Custom Serialization
FAMILIAR ABSTRACTIONS IN GELLY

- Gelly maps high-level abstractions to dataflows
  - vertex-centric
  - scatter-gather
  - gather-sum-apply
  - partition-centric
GELLY VERTEX-CENTRIC SHORTEST PATHS

```scala
final class SSSPComputeFunction extends ComputeFunction {

  override def compute(vertex: Vertex, messages: MessageIterator) = {

    var minDistance = if (vertex.getId == srcId) 0 else Double.MaxValue

    while (messages.hasNext) {
      val msg = messages.next
      if (msg < minDistance)
        minDistance = msg
    }

    if (vertex.getValue > minDistance) {
      setNewVertexValue(minDistance)
      for (edge: Edge <- getEdges)
        sendMessageTo(edge.getTarget, vertex.getValue + edge.getValue)
    }
  }
}
```
Scatter-Gather Dataflow

iteration

coGroup

gather()

Produce new vertex states

Solution set <Vid, Vv>

Messages <Vid, M>

Produce messages

coGroup

scatter()

Workset <Vid, Vv>

Edge set <Sid, Tid, Ev> cached

initial vertices
Gather-Sum-Apply
Dataflow
If you like “Inside job,” you might also like “The Bourne Identity.”

What’s the cheapest way to reach Zurich from London through Berlin?

Diverse graph models and applications

These are the top-10 relevant results for the search term “graph.”

Do you need distributed graph processing?
Specialized graph processing abstractions

- Vertex-Centric: arbitrary, arbitrary, any vertex, arbitrary
- Signal-Collect: arbitrary, based on received messages, any vertex, based on vertex state
- GSA: associative & commutative, based on neighbors' values, neighborhood, based on vertex state

Vertex-parallel models are very widespread. Beware of performance issues and anti-patterns!
General-purpose models for graph processing

Linear algebra primitives
Distributed dataflows

Vasia Kalavri
kalavriv@inf.ethz.ch
Programming Models and Tools for Distributed Graph Processing

Vasia Kalavri
kalavriv@inf.ethz.ch

31st British International Conference on Databases
10 July 2017, London, UK