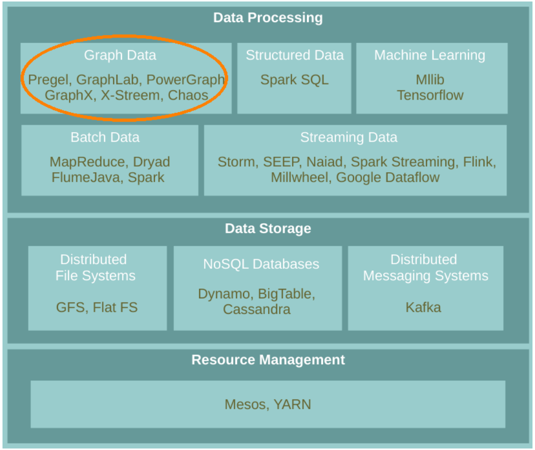
Large Scale Graph Processing - Pregel, GraphLab, and XStream

Amir H. Payberah
payberah@kth.se
08/10/2018

The Course Web Page

https://id2221kth.github.io

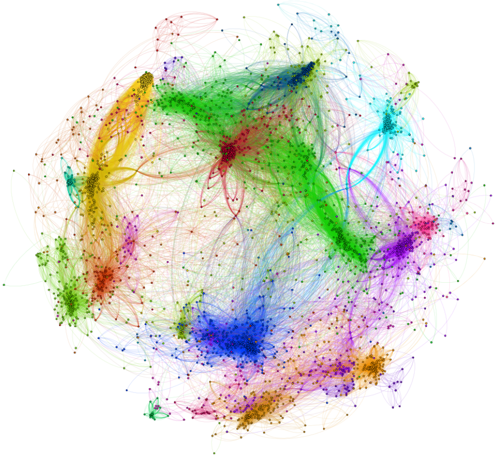
Where Are We?



Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File Systems
GFS, Flat FS
NoSQL Databases
Dynamo, BigTable,
Cassandra
Distributed
Messaging Systems
Kafka
Resource Management
Mesos, YARN

- A flexible abstraction for describing relationships between discrete objects.

Large Graph



Graph Algorithms Challenges

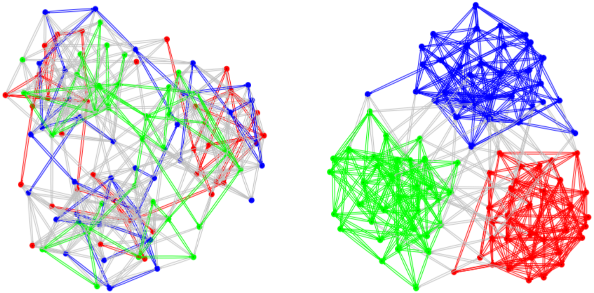
- Difficult to extract parallelism based on partitioning of the data.
- Difficult to express parallelism based on partitioning of computation.

Graph Algorithms Challenges

- Difficult to extract parallelism based on partitioning of the data.
- Difficult to express parallelism based on partitioning of computation.
- Graph partition is a challenging problem.

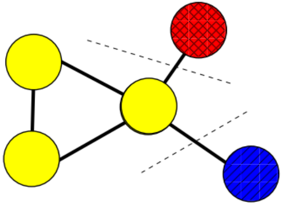
Graph Partitioning

- Partition large scale graphs and distribut to hosts.



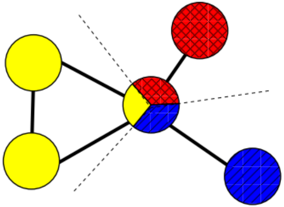
Edge-Cut Graph Partitioning

- Divide vertices of a graph into disjoint clusters.
- Nearly equal size (w.r.t. the number of vertices).
- With the minimum number of edges that span separated clusters.



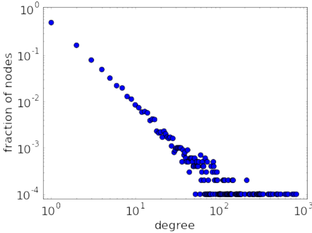
Vertex-Cut Graph Partitioning

- Divide edges of a graph into disjoint clusters.
- Nearly equal size (w.r.t. the number of edges).
- With the minimum number of replicated vertices.

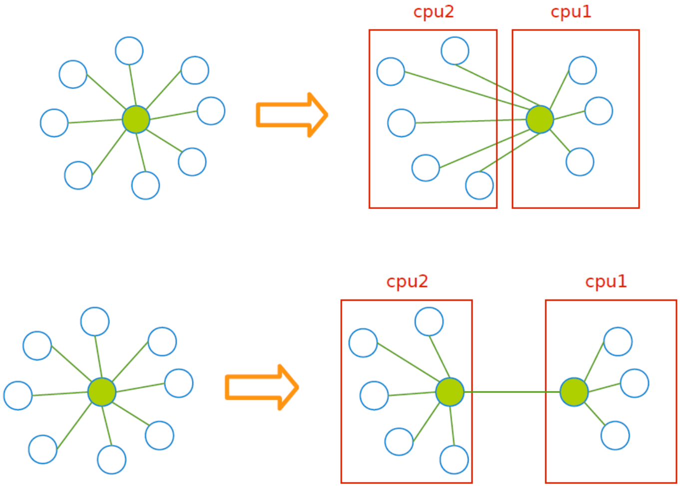


Edge-Cut vs. Vertex-Cut Graph Partitioning (1/2)

- Natural graphs: skewed Power-Law degree distribution.
- Edge-cut algorithms perform poorly on Power-Law Graphs.



Edge-Cut vs. Vertex-Cut Graph Partitioning (2/2)



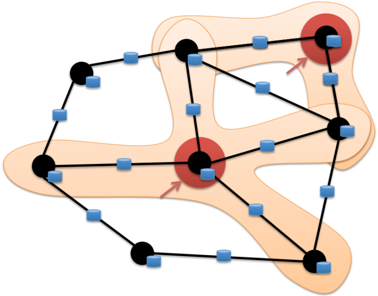
Different Approached to Process Large Scale Graphs

- Think like a vertex
- Think like an edge
- Think like a table
- Think like a graph
- Think like a matrix

Think Like a Vertex

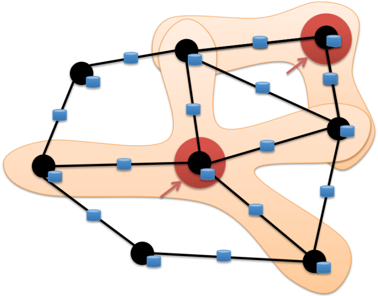
Think Like a Vertex

- Each vertex computes individually its value (in parallel)
- Computation typically depends on the neighbors.

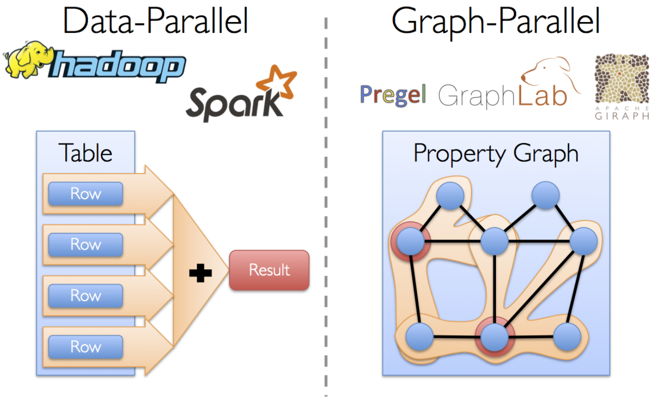


Think Like a Vertex

- Each vertex computes individually its value (in parallel)
- Computation typically depends on the neighbors.
- Also know as graph-parallel processing model.



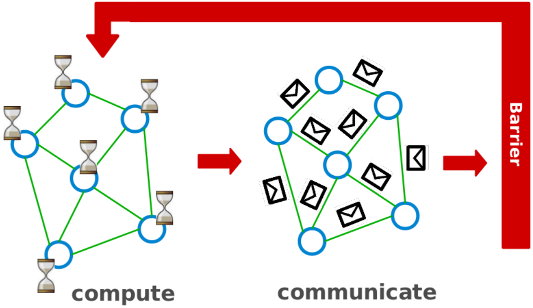
Data-Parallel vs. Graph-Parallel Computation



Pregel

Pregel

- Large-scale graph-parallel processing platform developed at Google.
- Inspired by bulk synchronous parallel (BSP) model.

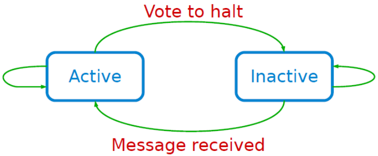


Execution Model (1/2)

- Applications run in sequence of iterations, called supersteps
- A vertex in superstep S can:
  - reads messages sent to it in superstep S−1.
  - sends messages to other vertices: receiving at superstep S+1.
  - modifies its state.
- Vertices communicate directly with one another by sending messages.

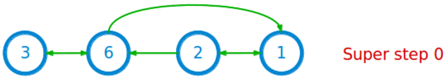
Execution Model (2/2)

- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.
- A halted vertex can be active if it receives a message.
- The whole algorithm terminates when:
  - All vertices are simultaneously inactive.
  - There are no messages in transit.



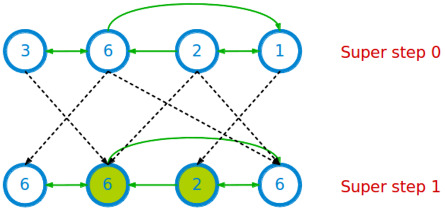
Example: Max Value (1/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



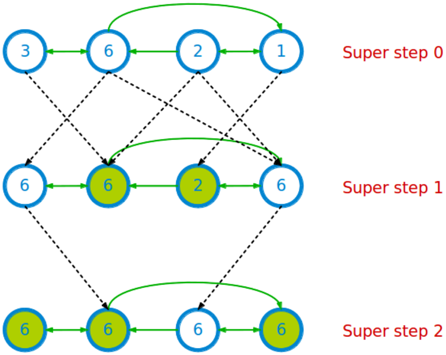
Example: Max Value (2/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



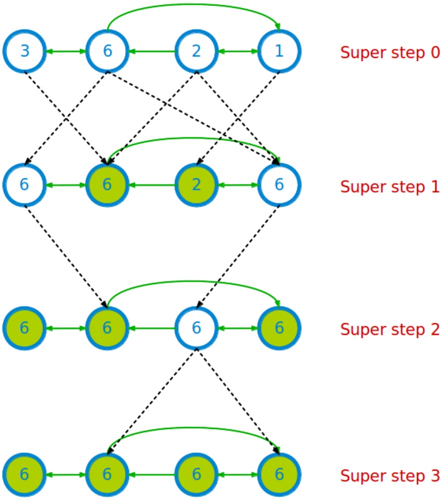
Example: Max Value (3/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```

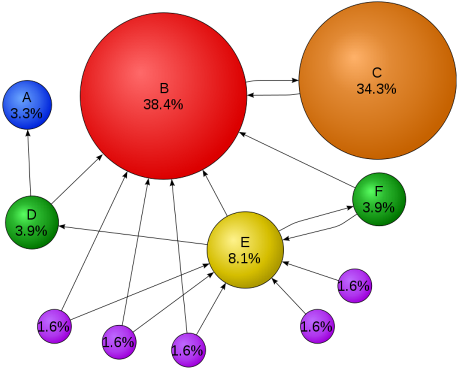


Example: Max Value (4/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



Example: PageRank



$$\mathtt{R[i]} = 0.15 + \sum_{\mathtt{j} \in \mathtt{Nbrs(i)}} \mathtt{w_{ji}R[j]}$$

Example: PageRank

```
Pregel_PageRank(i, messages):
  // receive all the messages
  total = 0
  foreach(msg in messages):
    total = total + msg

  // update the rank of this vertex
  R[i] = 0.15 + total

  // send new messages to neighbors
  foreach(j in out_neighbors[i]):
    sendmsg(R[i] * wij) to vertex j
```

$$\mathrm{R}[\mathrm{i}] = 0.15 + \sum_{\mathrm{j} \in \mathrm{Nbrs}(\mathrm{i})} \mathrm{w}_{\mathrm{ji}} \mathrm{R}[\mathrm{j}]$$

Graph Partitioning

- Edge-cut partitioning
- The pregel library divides a graph into a number of partitions.
- Each partition consists of vertices and all of those vertices' outgoing edges.
- Vertices are assigned to partitions based on their vertex-ID (e.g., hash(ID)).

System Model

- Master-worker model.
- The master
  - Coordinates workers.
  - Assigns one or more partitions to each worker.
  - Instructs each worker to perform a superstep.
- Each worker
  - Executes the local computation method on its vertices.
  - Maintains the state of its partitions.
  - Manages messages to and from other workers.

Fault Tolerance

- Fault tolerance is achieved through checkpointing.
  - Saved to persistent storage
- At start of each superstep, master tells workers to save their state:
  - Vertex values, edge values, incoming messages
- Master saves aggregator values (if any).
- When master detects one or more worker failures:
  - All workers revert to last checkpoint.

Pregel Limitations

- Inefficient if different regions of the graph converge at different speed.
- Runtime of each phase is determined by the slowest machine.

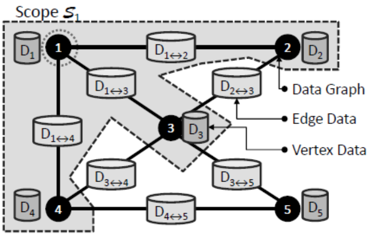
GraphLab/Turi

GraphLab

- GraphLab allows asynchronous iterative computation.
- Vertex scope of vertex `v`: the data stored in `v`, and in all adjacent vertices and edges.
- A vertex can read and modify any of the data in its scope (shared memory).

GraphLab

- GraphLab allows asynchronous iterative computation.
- Vertex scope of vertex v: the data stored in v, and in all adjacent vertices and edges.
- A vertex can read and modify any of the data in its scope (shared memory).



Example: PageRank (GraphLab)

```
GraphLab_PageRank(i)
  // compute sum over neighbors
  total = 0
  foreach(j in in_neighbors(i)):
    total = total + R[j] * wji

  // update the PageRank
  R[i] = 0.15 + total

  // trigger neighbors to run again
  foreach(j in out_neighbors(i)):
    signal vertex-program on j
```

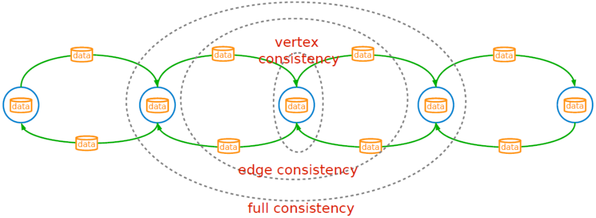
$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Consistency (1/4)

- Overlapped scopes: race-condition in simultaneous execution of two update functions.

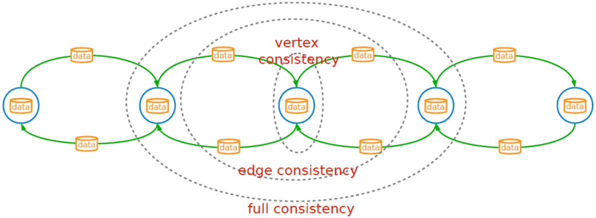
Consistency (1/4)

- Overlapped scopes: race-condition in simultaneous execution of two update functions.



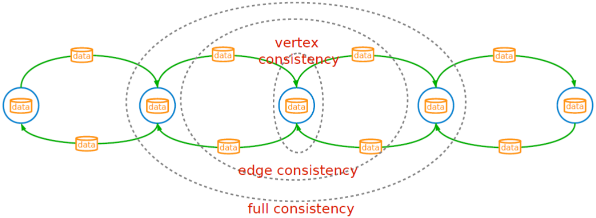
- Full consistency: during the execution `f(v)`, no other function reads or modifies data within the `v` scope.

Consistency (2/4)



- Edge consistency: during the execution `f(v)`, no other function reads or modifies any of the data on `v` or any of the edges adjacent to `v`.

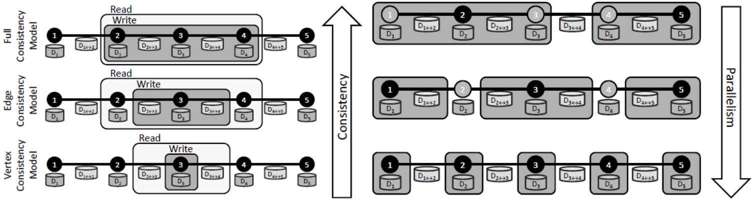
Consistency (3/4)



- Vertex consistency: during the execution `f(v)`, no other function will be applied to `v`.

Consistency (4/4)

Consistency vs. Parallelism



[Low, Y., GraphLab: A Distributed Abstraction for Large Scale Machine Learning (Doctoral dissertation, University of California), 2013.]

Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.

Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)

Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)

Consistency Implementation

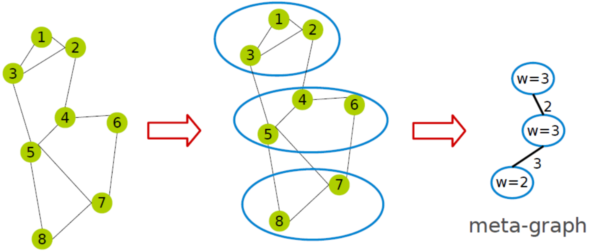
- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)
- Full consistency
  - Central vertex (write-locks), Adjacent vertices (write-locks)

Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)
- Full consistency
  - Central vertex (write-locks), Adjacent vertices (write-locks)
- Deadlocks are avoided by acquiring locks sequentially following a canonical order.

Graph Partitioning

- Edge-cut partitioning.
- Two-phase partitioning:
  1. Convert a large graph into a small meta-graph
  2. Partition the meta-graph



Fault Tolerance - Synchronous

- The systems periodically signals all computation activity to halt.
- Then synchronizes all caches, and saves to disk all data which has been modified since the last snapshot.
- Simple, but eliminates the systems advantage of asynchronous computation.

Fault Tolerance - Asynchronous

- Based on the Chandy-Lamport algorithm.
- The snapshot function is implemented as a function in vertices.
  - It takes priority over all other update functions.

```
if v was already snapshotted then
    Quit
Save D_v // Save current vertex
// Save all edges connected to un-snapshotted vertices
foreach u ∈ N[v] do                                  // Loop over neighbors
    if u was not snapshotted then
        Save D_{u→v} if edge u → v exists
        Save D_{v→u} if edge v → u exists
        Reschedule u for a Snapshot Update
Mark v as snapshotted
```

GraphLab2/Turi (PowerGraph)

PowerGraph

- Factorizes the local vertices functions into the Gather, Apply and Scatter phases.
- Vertx-cut partitioning.

Programming Model

- Gather-Apply-Scatter (GAS)
- Gather: accumulate information from neighborhood.
- Apply: apply the accumulated value to center vertex.
- Scatter: update adjacent edges and vertices.

Execution Model (1/2)

- Initially all vertices are active.
- It executes the vertex-program on the active vertices until none remain.
  - Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
  - Vertices can activate themselves and neighboring vertices.

Execution Model (1/2)

- Initially all vertices are active.
- It executes the vertex-program on the active vertices until none remain.
  - Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
  - Vertices can activate themselves and neighboring vertices.
- PowerGraph can execute both synchronously and asynchronously.

Execution Model (2/2)

- Synchronous scheduling like Pregel.
  - Executing the gather, apply, and scatter in order.
  - Changes made to the vertex/edge data are committed at the end of each step.

Execution Model (2/2)

- Synchronous scheduling like Pregel.
  - Executing the gather, apply, and scatter in order.
  - Changes made to the vertex/edge data are committed at the end of each step.

- Asynchronous scheduling like GraphLab.
  - Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
  - Visible to subsequent computation on neighboring vertices.

Example: PageRank (PowerGraph)

```
PowerGraph_PageRank(i):
  Gather(j -> i):
    return wji * R[j]

  sum(a, b):
    return a + b

  // total: Gather and sum
  Apply(i, total):
    R[i] = 0.15 + total

  Scatter(i -> j):
    if R[i] changed then activate(j)
```

$$\mathtt{R[i]} = 0.15 + \sum_{\mathtt{j} \in \mathtt{Nbrs(i)}} \mathtt{w_{ji}R[j]}$$

Graph Partitioning (1/2)

- Random vertex-cuts
- Randomly assign edges to machines.
- Completely parallel and easy to distribute.
- High replication factor.

Graph Partitioning (2/2)

- Greedy vertex-cuts
- $\mathtt{A(v)}$: set of machines that vertex $\mathtt{v}$ spans.

Graph Partitioning (2/2)

- Greedy vertex-cuts

- $\mathtt{A(v)}$: set of machines that vertex $\mathtt{v}$ spans.

- Case 1: If $\mathtt{A(u)} \cap \mathtt{A(v)} \neq \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to a machine in the intersection.

Graph Partitioning (2/2)

- Greedy vertex-cuts
- $\mathtt{A(v)}$: set of machines that vertex $\mathtt{v}$ spans.
- Case 1: If $\mathtt{A(u)} \cap \mathtt{A(v)} \neq \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to a machine in the intersection.
- Case 2: If $\mathtt{A(u)} \cap \mathtt{A(v)} = \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to one of the machines from the vertex with the most unassigned edges.

Graph Partitioning (2/2)

- Greedy vertex-cuts
- $\mathtt{A(v)}$: set of machines that vertex $\mathtt{v}$ spans.
- Case 1: If $\mathtt{A(u)} \cap \mathtt{A(v)} \neq \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to a machine in the intersection.
- Case 2: If $\mathtt{A(u)} \cap \mathtt{A(v)} = \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to one of the machines from the vertex with the most unassigned edges.
- Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Graph Partitioning (2/2)

- Greedy vertex-cuts
- $\mathtt{A}(\mathtt{v})$: set of machines that vertex $\mathtt{v}$ spans.
- Case 1: If $\mathtt{A}(\mathtt{u}) \cap \mathtt{A}(\mathtt{v}) \neq \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to a machine in the intersection.
- Case 2: If $\mathtt{A}(\mathtt{u}) \cap \mathtt{A}(\mathtt{v}) = \emptyset$, then the edge $(\mathtt{u}, \mathtt{v})$ should be assigned to one of the machines from the vertex with the most unassigned edges.
- Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.
- Case 4: If $\mathtt{A}(\mathtt{u}) = \mathtt{A}(\mathtt{v}) = \emptyset$, then assign the edge $(\mathtt{u}, \mathtt{v})$ to the least loaded machine.

Think Like an Edge

Motivation

- Could we process massive graphs on a single machine?
- Disk-based processing
  - Graph traversal = random access
  - Random access is inefficient for storage

Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM	567	2605	1057	2248
SSD	22.64	355	49.16	298
Disk	0.61	174	1.27	170

Note: 64 byte cachelines, 4K blocks (disk random), 16M chunks (disk sequential)

Eiko Y., and Roy A., "Scale-up Graph Processing: A Storage-centric View", 2013.

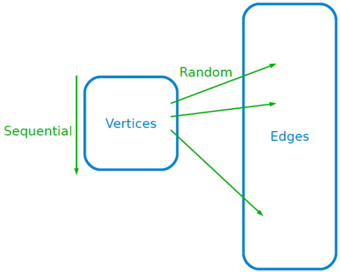
X-Stream

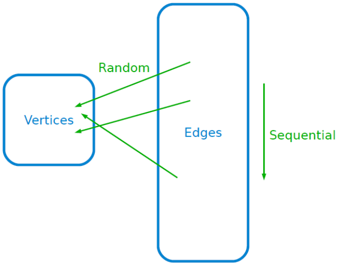
X-Stream

- X-Stream makes graph accesses sequential.
- Contribution:
  - Edge-centric scatter-gather model
  - Streaming partitions

Vertex-Centric vs. Edge-Centric Programming Model (1/2)

- Vertex-centric gather-scatter: iterates over vertices
- Edge-centric gather-scatter: iterates over edges





Vertex-Centric vs. Edge-Centric Programming Model (2/2)

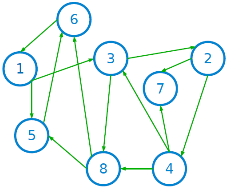
```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Vertex-Centric vs. Edge-Centric Programming Model (2/2)

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Vertex-Centric Breadth First Search (1/5)



vertices

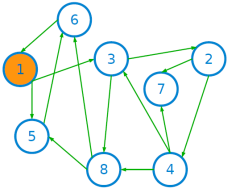
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Vertex-Centric Breadth First Search (2/5)



vertices

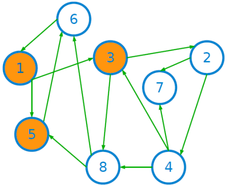
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Vertex-Centric Breadth First Search (3/5)



vertices

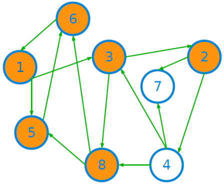
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Vertex-Centric Breadth First Search (4/5)



vertices

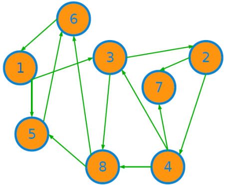
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Vertex-Centric Breadth First Search (5/5)



vertices

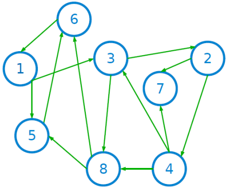
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
```

Edge-Centric Breadth First Search (1/5)



vertices

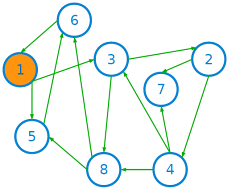
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Edge-Centric Breadth First Search (2/5)



vertices

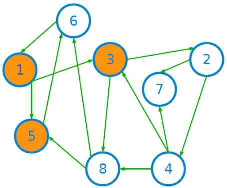
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Edge-Centric Breadth First Search (3/5)



vertices

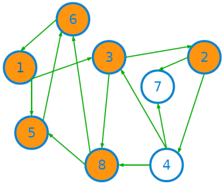
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Edge-Centric Breadth First Search (4/5)



vertices

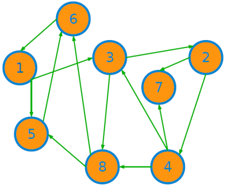
v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Edge-Centric Breadth First Search (5/5)



vertices

v
1
2
3
4
5
6
7
8

edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

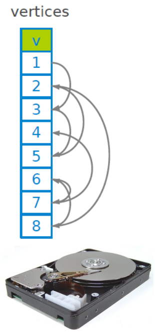
```
Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
```

Vertex-Centric vs. Edge-Centric Tradeoff

- Vertex-centric scatter-gather: $\frac{\texttt{EdgeData}}{\texttt{RandomAccessBandwidth}}$
- Edge-centric scatter-gather: $\frac{\texttt{Scatters} \times \texttt{EdgeData}}{\texttt{SequentialAccessBandwidth}}$
- Sequential Access Bandwidth $\gg$ Random Access Bandwidth.
- Few scatter gather iterations for real world graphs.

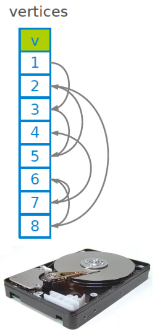
Streaming Partitions (1/4)

- Problem: still have random access to vertex set.



Streaming Partitions (1/4)

- Problem: still have random access to vertex set.



Solution

Partition the graph into streaming partitions.

Streaming Partitions (2/4)

edges

src	dest
1	3
4	7
2	7
4	8
3	2
3	8
4	3
1	5
2	4

vertices

v1
1
2
3
4

v2
5
6
7
8

src	dest
5	6
8	6
8	5
6	1

Subset of vertices

v1
1
2
3
4

edges

src	dest
1	3
4	7
2	7
4	8
3	2
3	8
4	3
1	5
2	4

Fast storage

Slow storage

Streaming Partitions (3/4)

- A streaming partition consists of: a vertex set, an edge list, and an update list.

Streaming Partitions (3/4)

- A streaming partition consists of: a vertex set, an edge list, and an update list.
- The vertex set: a subset of the vertex set of the graph that fits into the memory.
  - Vertex sets are mutually disjoint.
  - Their union equals the vertex set of the entire graph.

Streaming Partitions (3/4)

- A streaming partition consists of: a vertex set, an edge list, and an update list.
- The vertex set: a subset of the vertex set of the graph that fits into the memory.
    - Vertex sets are mutually disjoint.
    - Their union equals the vertex set of the entire graph.
- The edge list: all edges whose source vertex is in the partition's vertex set.

Streaming Partitions (3/4)

- A streaming partition consists of: a vertex set, an edge list, and an update list.
- The vertex set: a subset of the vertex set of the graph that fits into the memory.
  - Vertex sets are mutually disjoint.
  - Their union equals the vertex set of the entire graph.
- The edge list: all edges whose source vertex is in the partition's vertex set.
- The update list: all updates whose destination vertex is in the partition's vertex set.

Streaming Partitions (4/4)

```
// Scatter phase:
for each streaming_partition p
    read in vertex set of p
    for each edge e in edge list of p
        append update to Uout

// shuffle phase:
for each update u in Uout
    let p = partition containing target of u
    append u to Uin(p)
destroy Uout

//gather phase:
for each streaming_partition p
    read in vertex set of p
    for each update u in Uin(p)
        edge_gather(u)
    destroy Uin(p)
```

Summary

Summary

- Think like a vertex
  - Pregel: BSP, synchronous parallel model, message passing, edge-cut
  - GraphLab: asynchronous model, shared memory, edge-cut
  - PowerGraph: synchronous/asynchronous model, GAS, vertex-cut
- Think like an edge
  - XStream: edge-centric GAS, streaming partition

References

- G. Malewicz et al., “Pregel: a system for large-scale graph processing”, ACM SIGMOD 2010
- Y. Low et al., “Distributed GraphLab: a framework for machine learning and data mining in the cloud”, VLDB 2012
- J. Gonzalez et al., “Powergraph: distributed graph-parallel computation on natural graphs”, OSDI 2012
- A. Roy et al., “X-stream: Edge-centric graph processing using streaming partitions”, ACM SOSP 2013.

Questions?