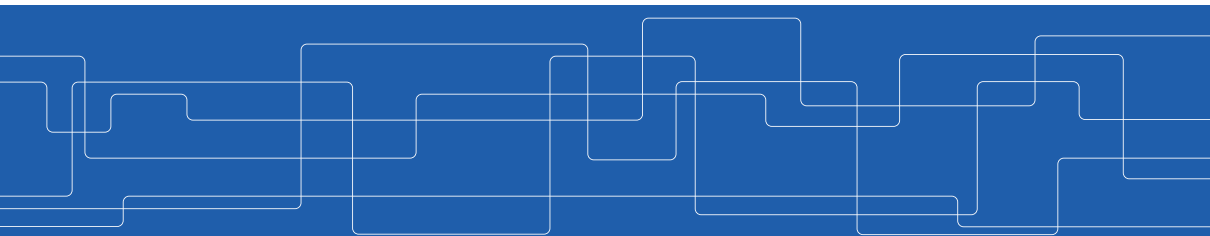




Large Scale Graph Processing - Pregel and GraphLab

Amir H. Payberah
payberah@kth.se
2020-09-23



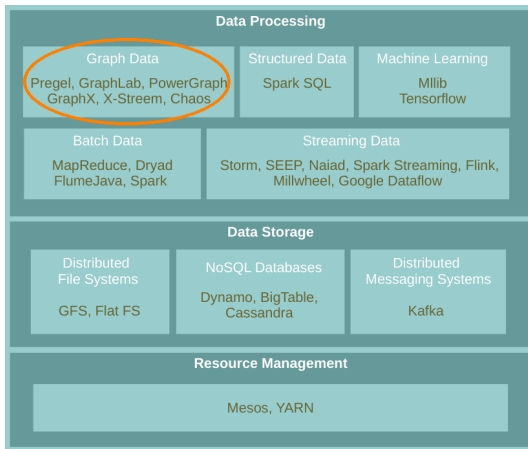


The Course Web Page

`https://id2221kth.github.io`

`https://tinyurl.com/y4qph82u`

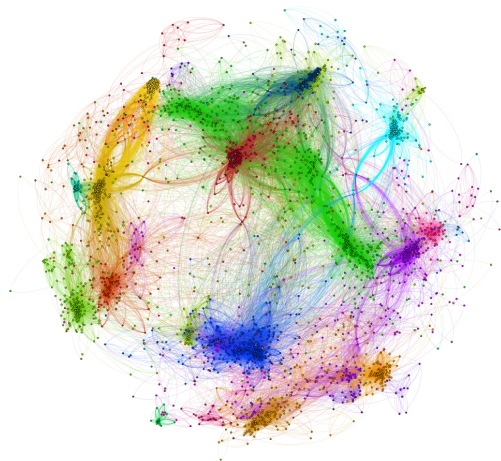
Where Are We?



- ▶ A flexible abstraction for describing relationships between discrete objects.



Large Graph



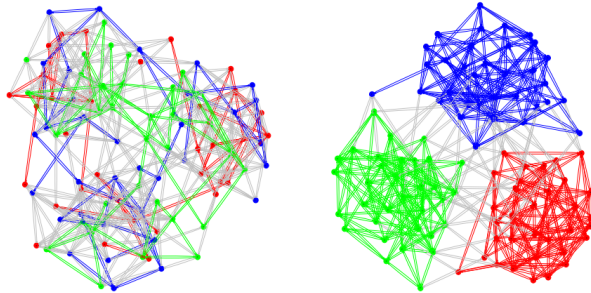


Graph Algorithms Challenges

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

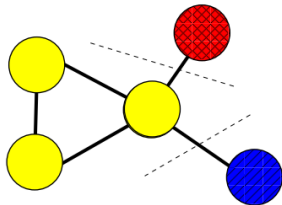
Graph Partitioning

- ▶ Partition large scale graphs and **distribut** to hosts.



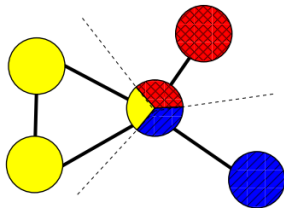
Edge-Cut Graph Partitioning

- ▶ 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**.



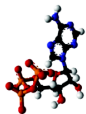
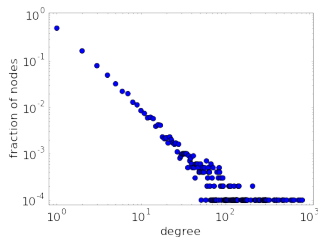
Vertex-Cut Graph Partitioning

- ▶ 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**.

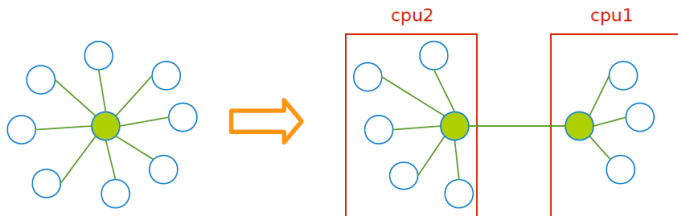
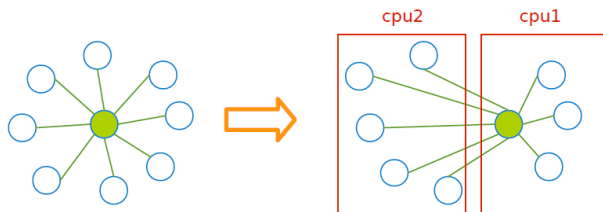


Edge-Cut vs. Vertex-Cut Graph Partitioning (1/2)

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

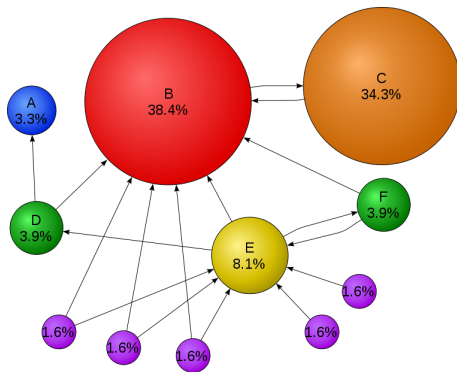


Edge-Cut vs. Vertex-Cut Graph Partitioning (2/2)





PageRank with MapReduce

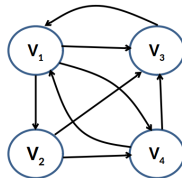


$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

PageRank Example (1/2)

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

▶ Input



V1: [0.25, V2, V3, V4]

V2: [0.25, V3, V4]

V3: [0.25, V1]

V4: [0.25, V1, V3]

▶ Share the rank among all outgoing links

V1: (V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3)

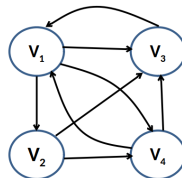
V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

V4: (V1, 0.25/2), (V3, 0.25/2)

PageRank Example (2/2)

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



V1: (V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3)

V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

V4: (V1, 0.25/2), (V3, 0.25/2)

▶ Output after one iteration

V1: [0.37, V2, V3, V4]

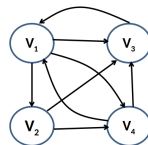
V2: [0.08, V3, V4]

V3: [0.33, V1]

V4: [0.20, V1, V3]

PageRank in MapReduce - Map (1/2)

► Map function



```
map(key: [url, pagerank], value: outlink_list)
  for each outlink in outlink_list:
    emit(key: outlink, value: pagerank / size(outlink_list))

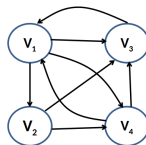
emit(key: url, value: outlink_list)
```

► Input (key, value)

```
((V1, 0.25), [V2, V3, V4])
((V2, 0.25), [V3, V4])
((V3, 0.25), [V1])
((V4, 0.25), [V1, V3])
```


PageRank in MapReduce - Map (2/2)

► Map function



```
map(key: [url, pagerank], value: outlink_list)
  for each outlink in outlink_list:
    emit(key: outlink, value: pagerank / size(outlink_list))

emit(key: url, value: outlink_list)
```

► Intermediate (key, value)

```
(V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3), (V3, 0.25/2), (V4, 0.25/2), (V1, 0.25/1),
(V1, 0.25/2), (V3, 0.25/2)
(V1, [V2, V3, V4])
(V2, [V3, V4])
(V3, [V1])
(V4, [V1, V3])
```



PageRank in MapReduce - Shuffle

► Intermediate (key, value)

```
(V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3), (V3, 0.25/2), (V4, 0.25/2), (V1, 0.25/1),  
(V1, 0.25/2), (V3, 0.25/2)  
(V1, [V2, V3, V4])  
(V2, [V3, V4])  
(V3, [V1])  
(V4, [V1, V3])
```

► After shuffling

```
(V1, 0.25/1), (V1, 0.25/2), (V1, [V2, V3, V4])  
(V2, 0.25/3), (V2, [V3, V4])  
(V3, 0.25/3), (V3, 0.25/2), (V3, 0.25/2), (V3, [V1])  
(V4, 0.25/3), (V4, 0.25/2), (V4, [V1, V3])
```



PageRank in MapReduce - Reduce (1/2)

▶ Reduce function

```
reducer(key: url, value: list_pr_or_urls)
  outlink_list = []
  pagerank = 0

  for each pr_or_urls in list_pr_or_urls:
    if is_list(pr_or_urls):
      outlink_list = pr_or_urls
    else
      pagerank += pr_or_urls

  emit(key: [url, pagerank], value: outlink_list)
```

▶ Input of the Reduce function

```
(V1, 0.25/1), (V1, 0.25/2), (V1, [V2, V3, V4])
(V2, 0.25/3), (V2, [V3, V4])
(V3, 0.25/3), (V3, 0.25/2), (V3, 0.25/2), (V3, [V1])
(V4, 0.25/3), (V4, 0.25/2), (V4, [V1, V3])
```



PageRank in MapReduce - Reduce (2/2)

▶ Reduce function

```
reducer(key: url, value: list_pr_or_urls)
  outlink_list = []
  pagerank = 0

  for each pr_or_urls in list_pr_or_urls:
    if is_list(pr_or_urls):
      outlink_list = pr_or_urls
    else
      pagerank += pr_or_urls

  emit(key: [url, pagerank], value: outlink_list)
```

▶ Output

```
((V1, 0.37), [V2, V3, V4])
((V2, 0.08), [V3, V4])
((V3, 0.33), [V1])
((V4, 0.20), [V1, V3])
```



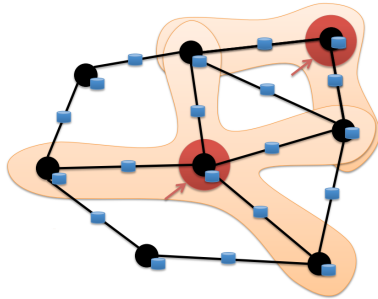
Problems with MapReduce for Graph Analytics

- ▶ MapReduce does **not directly support iterative** algorithms.
 - Invariant graph-topology-data **re-loaded** and **re-processed** at each iteration is **wasting** I/O, network bandwidth, and CPU
- ▶ **Materializations** of intermediate results at every MapReduce iteration **harm performance**.

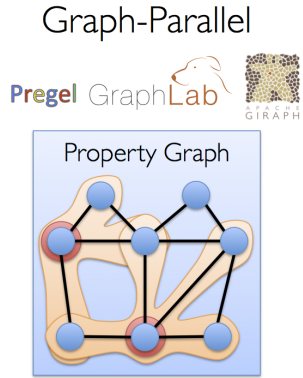
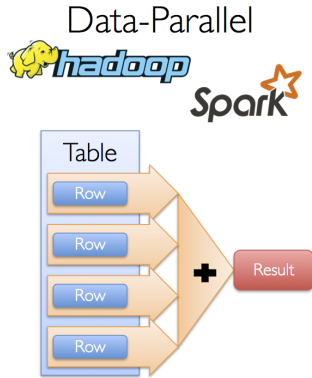
Think Like a Vertex

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.



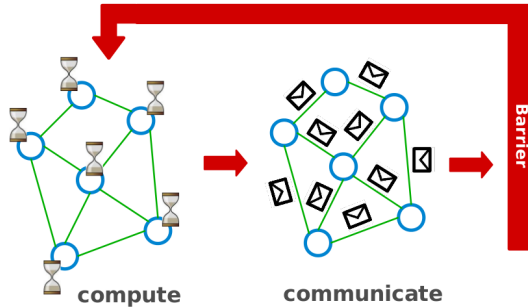
Data-Parallel vs. Graph-Parallel Computation



Pregel

Pregel

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



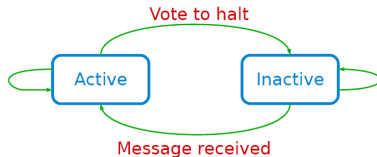


Execution Model (1/2)

- ▶ 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**.

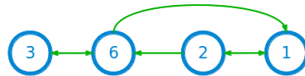
Execution Model (2/2)

- ▶ 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)
```



Super step 0

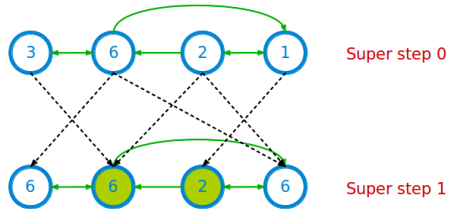
Example: Max Value (2/4)

```

i_val := val

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

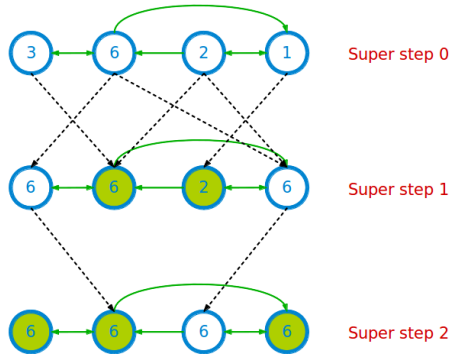
if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```



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)
  
```



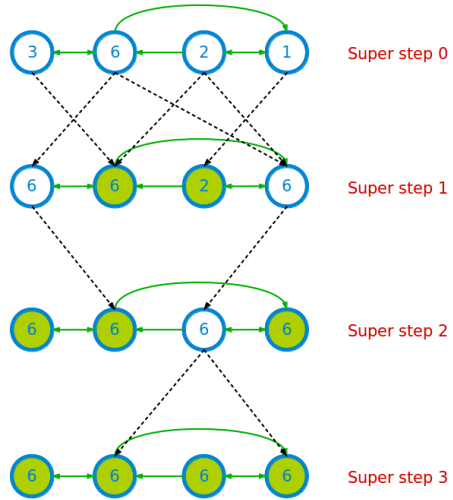
Example: Max Value (4/4)

```

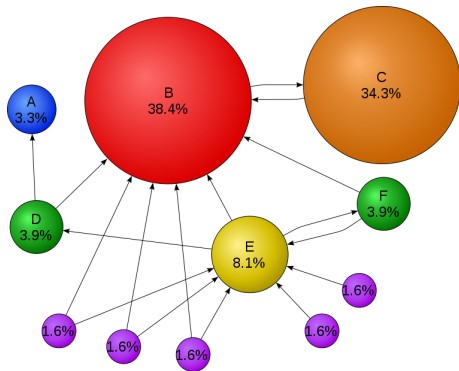
i_val := val

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



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



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

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



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., $\text{hash}(\text{ID})$).



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**.

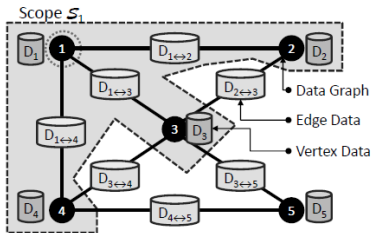


Pregel Limitations

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

GraphLab/Turi

- ▶ GraphLab 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

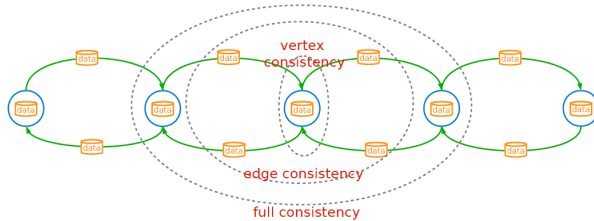
  // update the PageRank
  R[i] = total

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

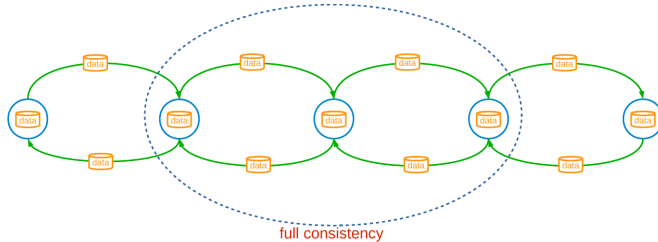
$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

Consistency (1/5)

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

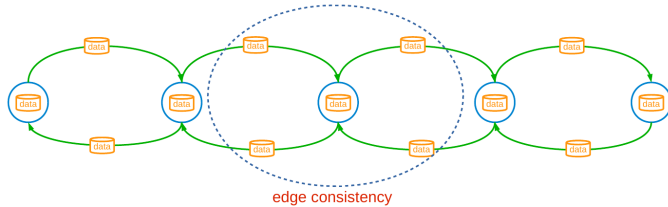


Consistency (2/5)



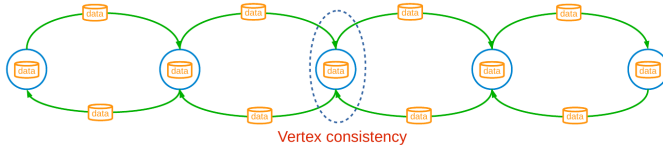
- ▶ **Full consistency:** during the execution $f(v)$, no other function reads or modifies data within the v scope.

Consistency (3/5)



- ▶ **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 .

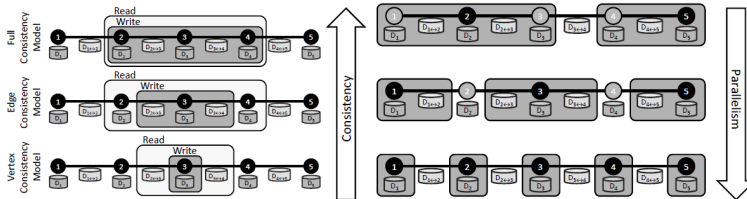
Consistency (4/5)



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

Consistency (5/5)

Consistency vs. Parallelism



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

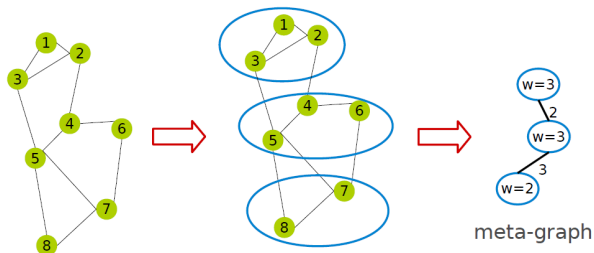


Consistency Implementation

- ▶ **Distributed locking**: associating a **readers-writer** lock with each vertex.
- ▶ **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  $u \in N[v]$  do // Loop over neighbors  
  | if u was not snapshotted then  
    | Save  $D_{u \rightarrow v}$  if edge  $u \rightarrow v$  exists  
    | Save  $D_{v \rightarrow u}$  if edge  $v \rightarrow u$  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.



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.
- ▶ 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.

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

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



Graph Partitioning (1/2)

- ▶ Vertex-cut partitioning.
- ▶ **Random** vertex-cuts: **randomly** assign edges to machines.
- ▶ Completely parallel and easy to **distribute**.
- ▶ **High replication** factor.



Graph Partitioning (2/2)

- ▶ **Greedy** vertex-cuts
- ▶ $A(v)$: set of machines that vertex v spans.
- ▶ **Case 1**: If $A(u) \cap A(v) \neq \emptyset$, then the edge (u, v) should be assigned to a machine in the intersection.
- ▶ **Case 2**: If $A(u) \cap A(v) = \emptyset$, then the edge (u, 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 $A(u) = A(v) = \emptyset$, then assign the edge (u, v) to the least loaded machine.

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



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

Questions?