GraphLab: A New Framework For Parallel Machine Learning

Amir H. Payberah
amir@sics.se

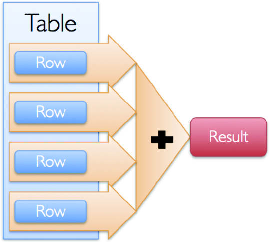
Amirkabir University of Technology
(Tehran Polytechnic)

AMIRKABIR UNIVERSITY
OF TECHNOLOGY

Reminder

Data-Parallel Model for Large-Scale Graph Processing

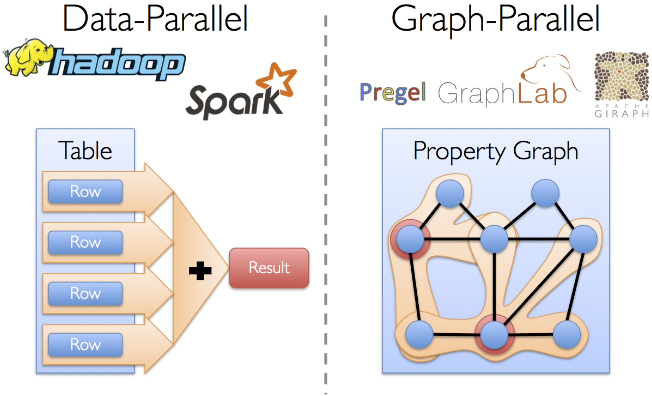
- The platforms that have worked well for developing parallel applications are not necessarily effective for large-scale graph problems.



Graph-Parallel Processing

- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general data-parallel systems.

Data-Parallel vs. Graph-Parallel Computation



Pregel

- Vertex-centric

Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)

Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)
- Runs in sequence of iterations (supersteps)

Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)
- Runs in sequence of iterations (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.

Pregel Limitations

- Inefficient if different regions of the graph converge at different speed.
- Can suffer if one task is more expensive than the others.
- Runtime of each phase is determined by the slowest machine.

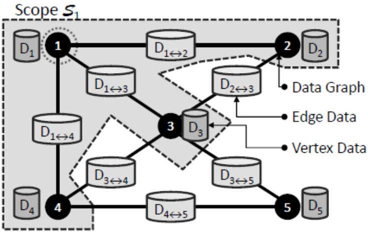
GraphLab

Data Model

- A directed graph that stores the program state, called data graph.

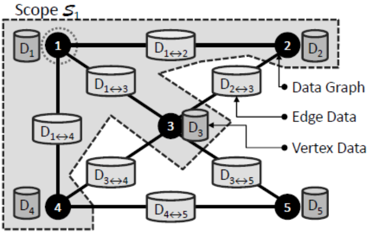
Vertex Scope

- The scope of vertex v is the data stored in vertex v, in all adjacent vertices and adjacent edges.



Programming Model (1/3)

- Rather than adopting a message passing as in Pregel, GraphLab allows the user defined function of a vertex to read and modify any of the data in its scope.



Programming Model (2/3)

- Update function: user-defined function similar to `Compute` in Pregel.
- Can read and modify the data within the scope of a vertex.
- Schedules the future execution of other update functions.

Programming Model (3/3)

- Sync function: similar to aggregate in Pregel.
- Maintains global aggregates.
- Performs periodically in the background.

Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**

$$(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})$$
$$(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$$
$$\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$$

Output: Modified Data Graph $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple (f, v) consisting of an update function f and a vertex v.

Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple (f, v) consisting of an update function f and a vertex v.
- After executing an update function $(f, g, \cdots)$ the modified scope data in S_v is written back to the data graph.

Example: PageRank

```
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
```

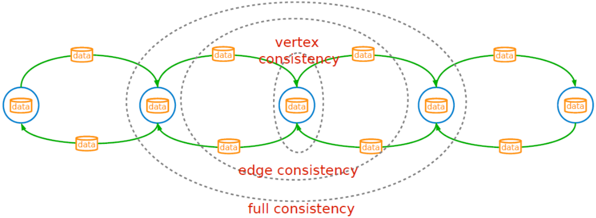
$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Data Consistency (1/3)

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

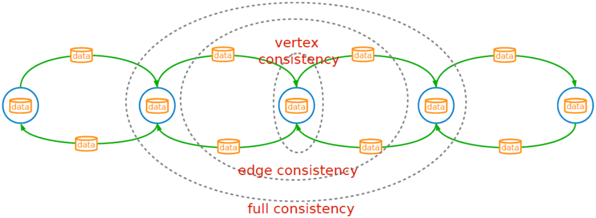
Data Consistency (1/3)

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



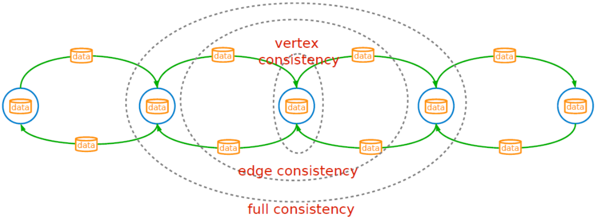
- Full consistency: during the execution $f(v)$, no other function reads or modifies data within the v scope.

Data Consistency (2/3)



- 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.

Data Consistency (3/3)



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

Sequential Consistency (1/2)

- Proving the correctness of a parallel algorithm: sequential consistency

Sequential Consistency (1/2)

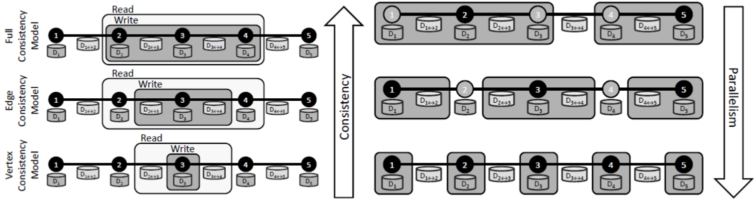
- Proving the correctness of a parallel algorithm: sequential consistency
- Sequential consistency: if for every parallel execution, there exists a sequential execution of update functions that produces an equivalent result.

Sequential Consistency (2/2)

- A simple method to achieve serializability is to ensure that the scopes of concurrently executing update functions do not overlap.

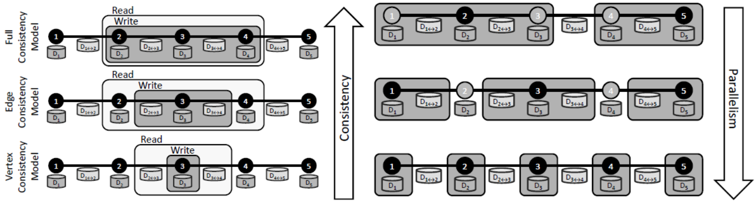
Sequential Consistency (2/2)

- A simple method to achieve serializability is to ensure that the scopes of concurrently executing update functions do not overlap.
  - The full consistency model is used.



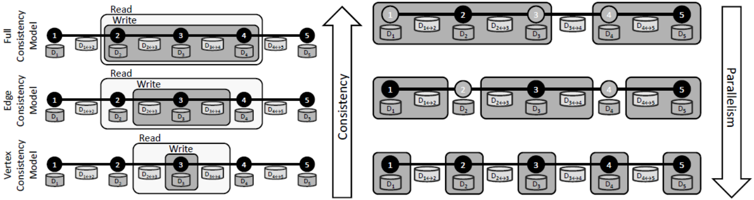
Sequential Consistency (2/2)

- A simple method to achieve serializability is to ensure that the scopes of concurrently executing update functions do not overlap.
  - The full consistency model is used.
  - The edge consistency model is used and update functions do not modify data in adjacent vertices.

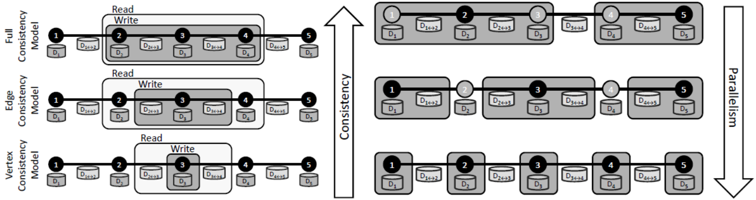


Sequential Consistency (2/2)

- A simple method to achieve serializability is to ensure that the scopes of concurrently executing update functions do not overlap.
  - The full consistency model is used.
  - The edge consistency model is used and update functions do not modify data in adjacent vertices.
  - The vertex consistency model is used and update functions only access local vertex data.



Consistency vs. Parallelism



Consistency vs. Parallelism

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

GraphLab Implementation

- Shared memory implementation
- Distributed implementation

GraphLab Implementation

- Shared memory implementation
- Distributed implementation

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
$\quad \boxed{(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})}$
$\quad (\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
$\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- In what order should the tasks (vertex-update function pairs) be called?

Tasks Schedulers (1/2)

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
  $\boxed{(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})}$
  $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- In what order should the tasks (vertex-update function pairs) be called?
  - A collection of base schedules, e.g., round-robin, and synchronous.
  - Set scheduler: enables users to compose custom update schedules.

Tasks Schedulers (2/2)

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**

$(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
$(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
$\boxed{\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'}$

Output: Modified Data Graph $G = (V, E, D')$

- How to add new task in the queue?

Tasks Schedulers (2/2)

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\boxed{\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'}$
Output: Modified Data Graph $G = (V, E, D')$

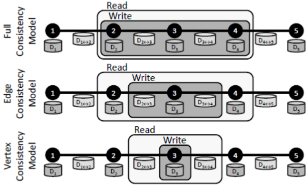
- How to add new task in the queue?
  - FIFO: only permits task creation but do not permit task reordering.
  - Prioritized: permits task reordering at the cost of increased overhead.

Consistency

- Implemented in C++ using PThreads for parallelism.
- Consistency: read-write lock

Consistency

- Implemented in C++ using PThreads for parallelism.
- Consistency: read-write lock
- 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.

GraphLab Implementation

- Shared memory implementation
- Distributed implementation

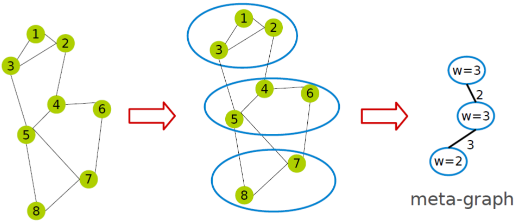
Distributed Implementation

- Graph partitioning
  - How to efficiently load, partition and distribute the data graph across machines?
- Consistency
  - How to achieve consistency in the distributed setting?
- Fault tolerance

Graph Partitioning

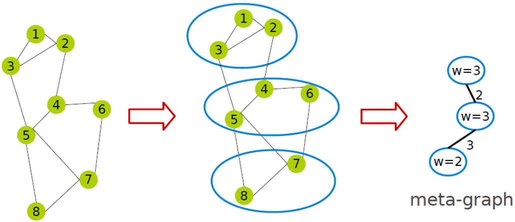
Graph Partitioning - Phase 1 (1/2)

- Two-phase partitioning.
- Partitioning the data graph into k parts, called atom.
    - k $\gg$ number of machines



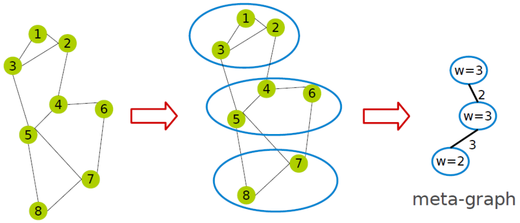
Graph Partitioning - Phase 1 (1/2)

- Two-phase partitioning.
- Partitioning the data graph into k parts, called atom.
  - k $\gg$ number of machines
- meta-graph: the graph of atoms (one vertex for each atom).



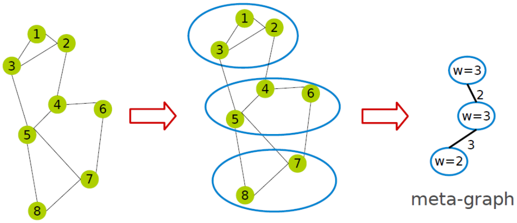
Graph Partitioning - Phase 1 (1/2)

- Two-phase partitioning.
- Partitioning the data graph into k parts, called atom.
  - k ≫ number of machines
- meta-graph: the graph of atoms (one vertex for each atom).
- Atom weight: the amount of data it stores.

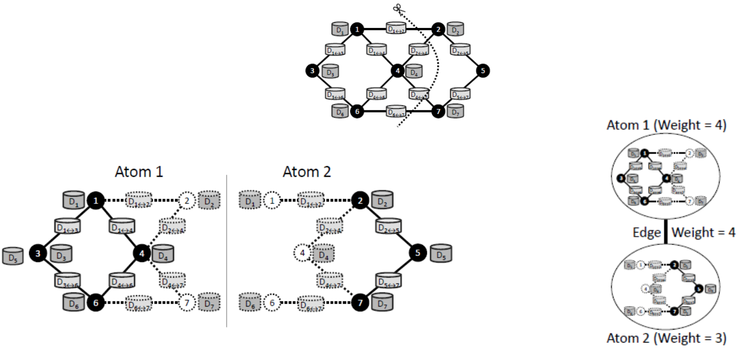


Graph Partitioning - Phase 1 (1/2)

- Two-phase partitioning.
- Partitioning the data graph into k parts, called atom.
  - k $\gg$ number of machines
- meta-graph: the graph of atoms (one vertex for each atom).
- Atom weight: the amount of data it stores.
- Edge weight: the number of edges crossing the atoms.

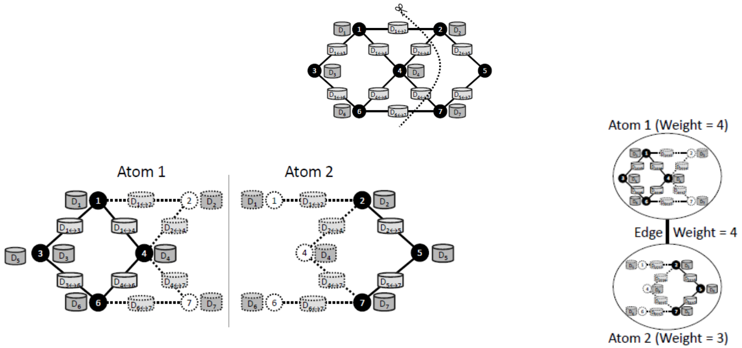


Graph Partitioning - Phase 1 (2/2)



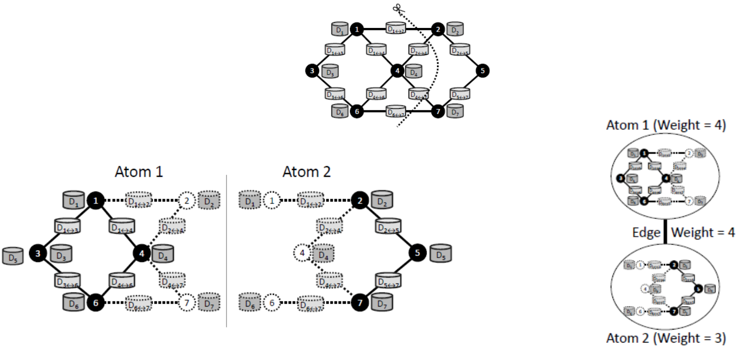
- Each atom is stored as a separate file on a distributed storage system, e.g., HDFS.

Graph Partitioning - Phase 1 (2/2)



- Each atom is stored as a separate file on a distributed storage system, e.g., HDFS.
- Each atom file is a simple binary that stores interior and the ghosts of the partition information.

Graph Partitioning - Phase 1 (2/2)



- Each atom is stored as a separate file on a distributed storage system, e.g., HDFS.
- Each atom file is a simple binary that stores interior and the ghosts of the partition information.
- Ghost: set of vertices and edges adjacent to the partition boundary.

Graph Partitioning - Phase 2

- Meta-graph is very small.
- A fast balanced partition of the meta-graph over the physical machines.
- Assigning graph atoms to machines.

Consistency

Consistency

- To achieve a serializable parallel execution of a set of dependent tasks.
  - Chromatic engine
  - Distributed locking engine

Consistency - Chromatic Engine

- Construct a vertex coloring: assigns a color to each vertex such that no adjacent vertices share the same color.

Consistency - Chromatic Engine

- Construct a vertex coloring: assigns a color to each vertex such that no adjacent vertices share the same color.

- Edge consistency: executing, synchronously, all update tasks associated with vertices of the same color before proceeding to the next color.

Consistency - Chromatic Engine

- Construct a vertex coloring: assigns a color to each vertex such that no adjacent vertices share the same color.

- Edge consistency: executing, synchronously, all update tasks associated with vertices of the same color before proceeding to the next color.

- Full consistency: no vertex shares the same color as any of its distance two neighbors.

Consistency - Chromatic Engine

- Construct a vertex coloring: assigns a color to each vertex such that no adjacent vertices share the same color.
- Edge consistency: executing, synchronously, all update tasks associated with vertices of the same color before proceeding to the next color.
- Full consistency: no vertex shares the same color as any of its distance two neighbors.
- Vertex consistency: assigning all vertices the same color.

Consistency - Distributed Locking Engine

- Associating a readers-writer lock with each vertex.

Consistency - Distributed Locking Engine

- Associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)

Consistency - Distributed Locking Engine

- 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 - Distributed Locking Engine

- 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 - Distributed Locking Engine

- 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.

Fault Tolerance

Fault Tolerance - Synchronous

- The systems periodically signals all computation activity to halt.

Fault Tolerance - Synchronous

- The systems periodically signals all computation activity to halt.
- Then synchronizes all caches (ghosts) and saves to disk all data which has been modified since the last snapshot.

Fault Tolerance - Synchronous

- The systems periodically signals all computation activity to halt.
- Then synchronizes all caches (ghosts) 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.

Fault Tolerance - Asynchronous

- Based on the Chandy-Lamport algorithm.

- The snapshot function is implemented as an update function in vertices.

Fault Tolerance - Asynchronous

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

Fault Tolerance - Asynchronous

- Based on the Chandy-Lamport algorithm.
- The snapshot function is implemented as an update function in vertices.
- The snapshot update takes priority over all other update functions.
- Edge Consistency is used on all update functions.

Fault Tolerance - Asynchronous

- Based on the Chandy-Lamport algorithm.
- The snapshot function is implemented as an update function in vertices.
- The snapshot update takes priority over all other update functions.
- Edge Consistency is used on all update functions.

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

Summary

GraphLab Summary

- Asynchronous model
- Vertex-centric
- Communication: distributed shared memory
- Three consistency levels: full, edge-level, and vertex-level
- Partitioning: two-phase partitioning
- Consistency: chromatic engine (graph coloring), distributed lock engine (reader-writer lock)

GraphLab Limitations

- Poor performance on Natural graphs.

Questions?