Large-Scale Graph Processing

Amir H. Payberah
Swedish Institute of Computer Science

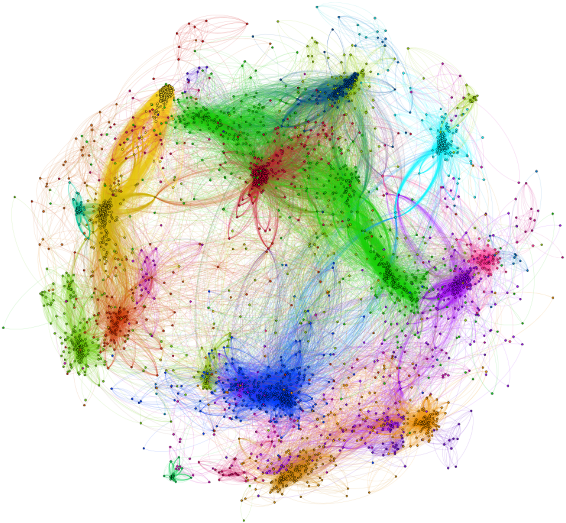
amir@sics.se
May 13-15, 2014

Introduction

- Graphs provide a flexible abstraction for describing relationships between discrete objects.

- Many problems can be modeled by graphs and solved with appropriate graph algorithms.

Large Graph



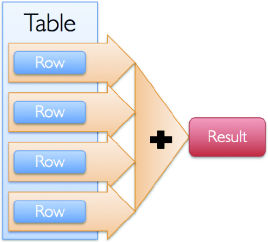
Large-Scale Graph Processing

- Large graphs need large-scale processing.

- A large graph either cannot fit into memory of single computer or it fits with huge cost.

Question

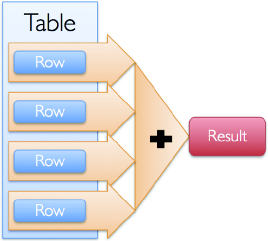
Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?



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.

- Why?



Graph Algorithms Characteristics (1/2)

- Unstructured problems
  - Difficult to extract parallelism based on partitioning of the data: the irregular structure of graphs.
  - Limited scalability: unbalanced computational loads resulting from poorly partitioned data.

Graph Algorithms Characteristics (1/2)

- Unstructured problems
  - Difficult to extract parallelism based on partitioning of the data: the irregular structure of graphs.
  - Limited scalability: unbalanced computational loads resulting from poorly partitioned data.

- Data-driven computations
  - Difficult to express parallelism based on partitioning of computation: the structure of computations in the algorithm is not known a priori.
  - The computations are dictated by nodes and links of the graph.

Graph Algorithms Characteristics (2/2)

- Poor data locality
  - The computations and data access patterns do not have much locality: the irregular structure of graphs.

Graph Algorithms Characteristics (2/2)

- Poor data locality
  - The computations and data access patterns do not have much locality: the irregular structure of graphs.

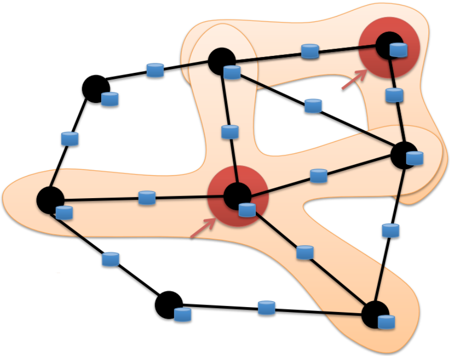
- High data access to computation ratio
  - Graph algorithms are often based on exploring the structure of a graph to perform computations on the graph data.
  - Runtime can be dominated by waiting memory fetches: low locality.

Proposed Solution

Graph-Parallel Processing

Proposed Solution

Graph-Parallel Processing

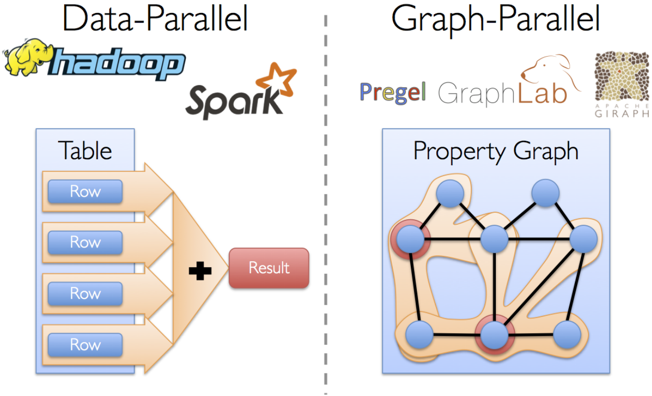


- Computation typically depends on the neighbors.

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.

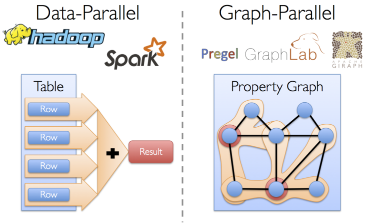
Data-Parallel vs. Graph-Parallel Computation



Data-Parallel vs. Graph-Parallel Computation

- Data-parallel computation
  - Record-centric view of data.
  - Parallelism: processing independent data on separate resources.

- Graph-parallel computation
  - Vertex-centric view of graphs.
  - Parallelism: partitioning graph (dependent) data across processing resources, and resolving dependencies (along edges) through iterative computation and communication.



Outline

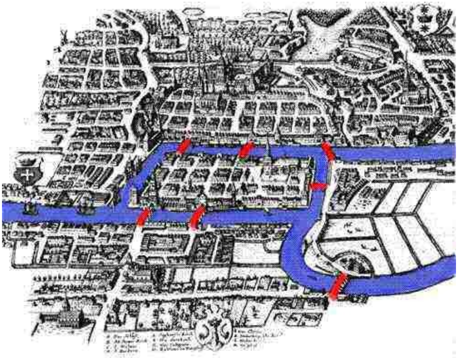
- Pregel
- GraphLab
- PowerGraph
- GraphX

Pregel

oogle

Seven Bridges of Königsberg

- Finding a walk through the city that would cross each bridge once and only once.
- Euler proved that the problem has no solution.



Map of Königsberg in Euler's time, highlighting the river Pregel and the bridges.

Pregel

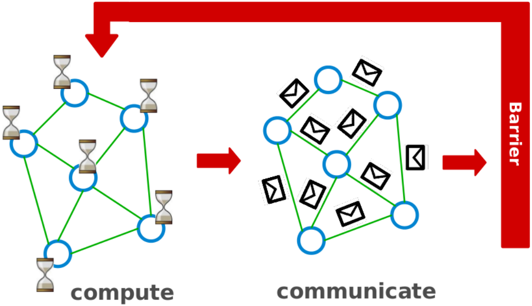
- Large-scale graph-parallel processing platform developed at Google.

- Inspired by bulk synchronous parallel (BSP) model.

Bulk Synchronous Parallel (1/2)

- It is a parallel programming model.
- The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes.
  - There are no special combining, replicating, or broadcasting facilities.

Bulk Synchronous Parallel (2/2)



All vertices update in parallel (at the same time).

Vertex-Centric Programs

- Think like a vertex.
- Each vertex computes individually its value: in parallel
- Each vertex can see its local context, and updates its value accordingly.

Data Model

- A directed graph that stores the program state, e.g., the current value.

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps
- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps

- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel

- 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.

Execution Model (1/3)

- Applications run in sequence of iterations: supersteps
- During a superstep, user-defined functions for each vertex is invoked (method `Compute()`): in parallel
- 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 (2/3)

- Superstep 0: all vertices are in the active state.

Execution Model (2/3)

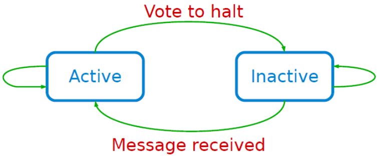
- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.

Execution Model (2/3)

- 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.

Execution Model (2/3)

- 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.



Execution Model (3/3)

- Aggregation: a mechanism for global communication, monitoring, and data.

Execution Model (3/3)

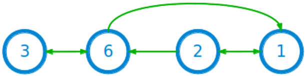
- Aggregation: a mechanism for global communication, monitoring, and data.
- Runs after each superstep.
- Each vertex can provide a value to an aggregator in superstep S.
- The system combines those values and the resulting value is made available to all vertices in superstep $S + 1$.

Execution Model (3/3)

- Aggregation: a mechanism for global communication, monitoring, and data.
- Runs after each superstep.
- Each vertex can provide a value to an aggregator in superstep S.
- The system combines those values and the resulting value is made available to all vertices in superstep $S + 1$.
- A number of predefined aggregators, e.g., min, max, sum.
- Aggregation operators should be commutative and associative.

Example: Max Value (1/4)

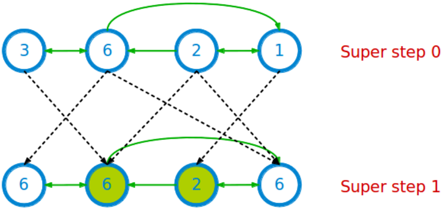
```
i_val := val

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

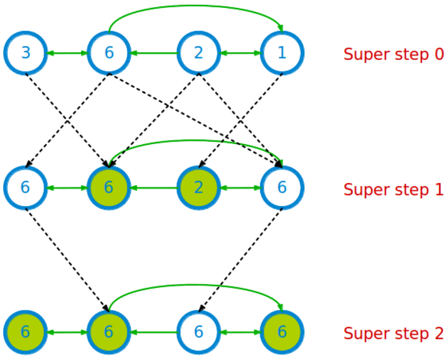
Example: Max Value (2/4)

```
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)
```



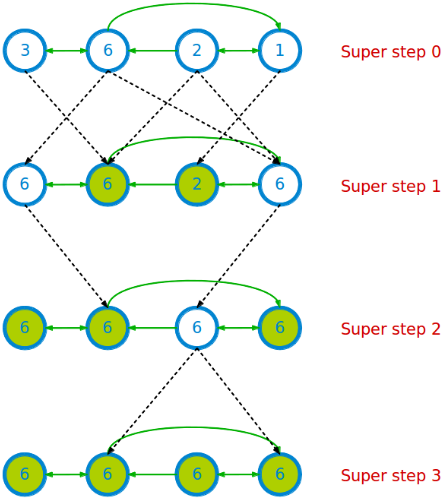
Example: Max Value (3/4)

```
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)

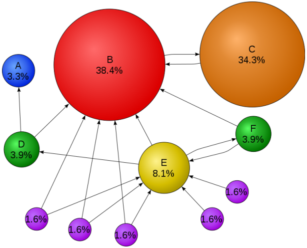
```
i_val := val

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

- Update ranks in parallel.
- Iterate until convergence.

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



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] = 0.15 + total

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

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

Partitioning the Graph

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

Implementation (1/4)

- Master-worker model.
- User programs are copied on machines.
- One copy becomes the master.

Implementation (2/4)

- The master is responsible for
  - Coordinating workers activity.
  - Determining the number of partitions.
- Each worker is responsible for
  - Maintaining the state of its partitions.
  - Executing the user's `Compute()` method on its vertices.
  - Managing messages to and from other workers.

Implementation (3/4)

- The master assigns one or more partitions to each worker.

Implementation (3/4)

- The master assigns one or more partitions to each worker.

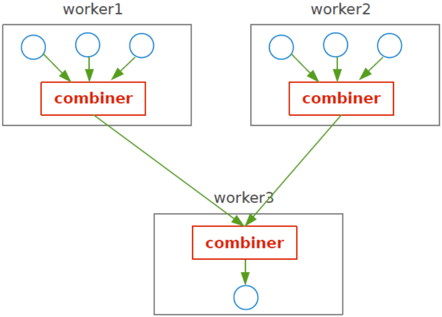
- The master assigns a portion of user input to each worker.
  - Set of records containing an arbitrary number of vertices and edges.
  - If a worker loads a vertex that belongs to that worker's partitions, the appropriate data structures are immediately updated.
  - Otherwise the worker enqueues a message to the remote peer that owns the vertex.

Implementation (4/4)

- After the input has finished loading, all vertices are marked as active.
- The master instructs each worker to perform a superstep.
- After the computation halts, the master may instruct each worker to save its portion of the graph.

Combiner

- Sending a message between workers incurs some overhead: use combiner.
- This can be reduced in some cases: sometimes vertices only care about a summary value for the messages it is sent (e.g., min, max, sum, avg).



Fault Tolerance (1/2)

- Fault tolerance is achieved through checkpointing.
- At start of each superstep, master tells workers to save their state:
  - Vertex values, edge values, incoming messages
  - Saved to persistent storage
- Master saves aggregator values (if any).
- This is not necessarily done at every superstep: costly

Fault Tolerance (2/2)

- When master detects one or more worker failures:
  - All workers revert to last checkpoint.
  - Continue from there.
  - That is a lot of repeated work.
  - At least it is better than redoing the whole job.

Pregel Summary

- Bulk Synchronous Parallel model
- Vertex-centric
- Superstep: sequence of iterations
- Master-worker model
- Communication: message passing

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.

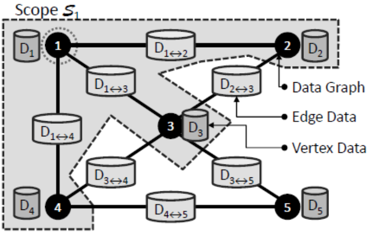
GraphLab

Data Model

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

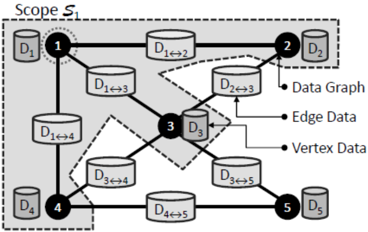
Vertex Scope

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



Execution Model (1/4)

- 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.



Execution Model (2/4)

- 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.

Execution Model (3/4)

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**

```
1    (f, v) ← RemoveNext (T)
2    (T′, S_v) ← f(v, S_v)
3    T ← T ∪ T′
```

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

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

- After executing an update function $(f, g, \cdots)$ the modified scope data in S_v is written back to the data graph.
- 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 (4/4)

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

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
```

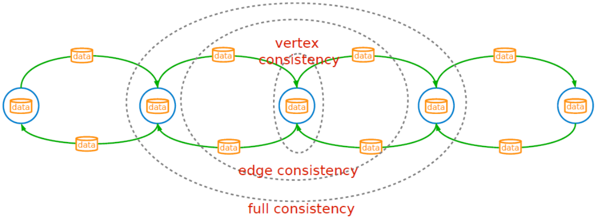
$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Data Consistency (1/3)

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

Data Consistency (1/3)

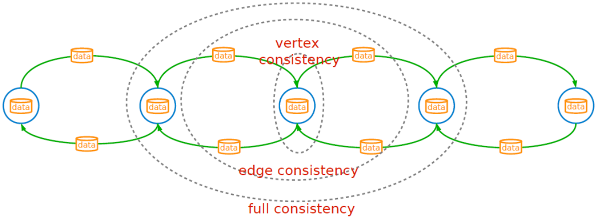
- Overlapped scopes: race-condition in simultaneous execution of two update functions.



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

Data Consistency (1/3)

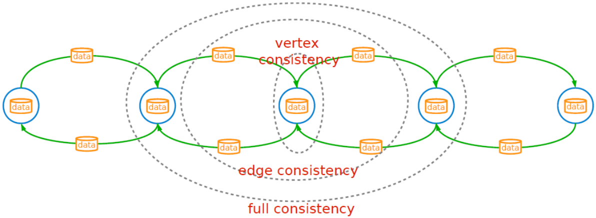
- Overlapped scopes: race-condition in simultaneous execution of two update functions.



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

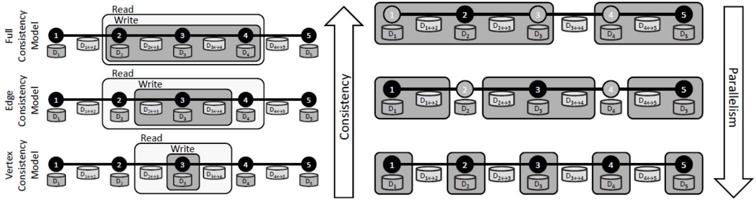
Data Consistency (1/3)

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

Data Consistency (2/3)



Consistency vs. Parallelism

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

Data Consistency (3/3)

- Proving the correctness of a parallel algorithm: sequential consistency

Data Consistency (3/3)

- 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.

Data Consistency (3/3)

- 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.
- 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.

GraphLab Implementation

- Shared memory implementation
- Distributed implementation

GraphLab Implementation

- Shared memory implementation
- Distributed implementation

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?

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**

$\quad \boxed{(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})}$

$\quad (\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$

$\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

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

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

$(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})$

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

$\boxed{\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'}$

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

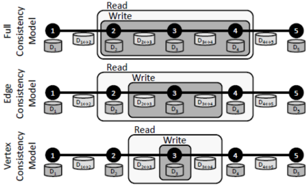
- How to add new task in the queue?
  - FIFO: only permits task creation but do not permit task reordering.
  - Prioritized: permits task reordering at the cost of increased overhead.

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

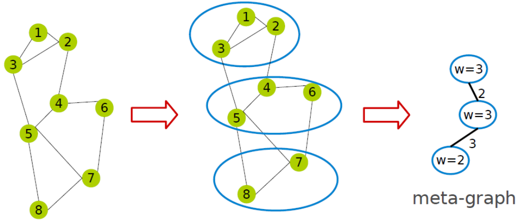
- Shared memory implementation
- Distributed implementation

Distributed Implementation

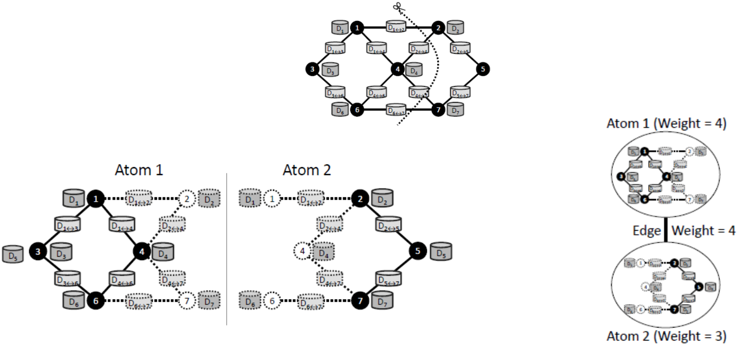
- Graph partitioning
  - How to efficiently load, partition and distribute the data graph across machines?
- Consistency
  - How to achieve consistency in the distributed setting?
- Fault tolerance

Graph Partitioning - 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.
- 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

- 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.
- 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 - 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.
- 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 ∈ 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
```

GraphLab Summary

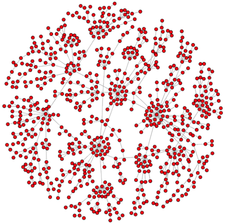
- Asynchronous model
- Vertex-centric
- Communication: distributed shared memory
- Three consistency levels: full, edge-level, and vertex-level

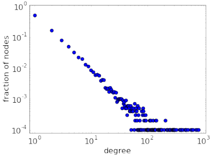
GraphLab Limitations

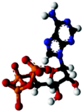
- Poor performance on Natural graphs.

Natural Graphs

- Graphs derived from natural phenomena.
- Skewed Power-Law degree distribution.



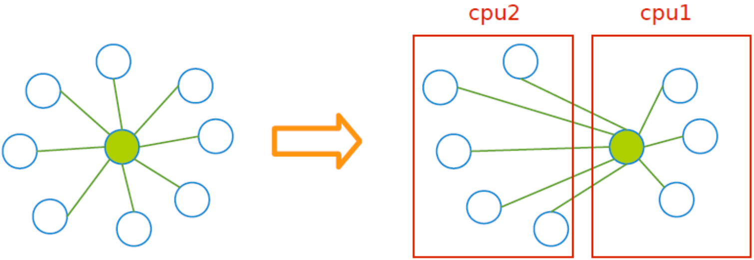




Natural Graphs Challenges

- Traditional graph-partitioning algorithms (edge-cut algorithms) perform poorly on Power-Law Graphs.

- Challenges of high-degree vertices.

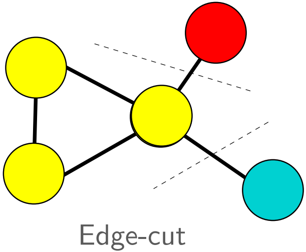


Proposed Solution

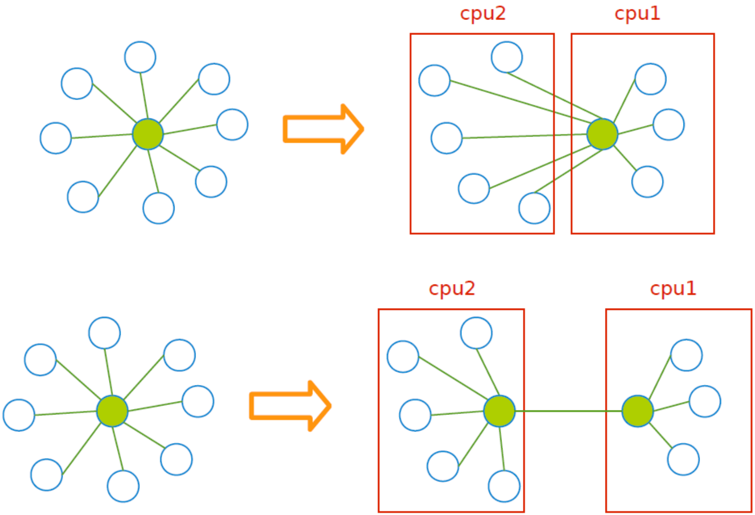
Vertex-Cut Partitioning

Proposed Solution

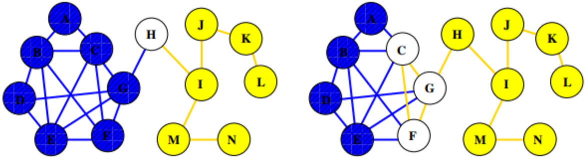
Vertex-Cut Partitioning



Edge-cut vs. Vertex-cut Partitioning



Edge-cut vs. Vertex-cut Partitioning



Edge-cut Vertex-cut

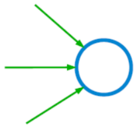
PowerGraph

PowerGraph

- Vertex-cut partitioning of graphs.
- Factorizes the GraphLab update function into the Gather, Apply and Scatter phases (GAS).

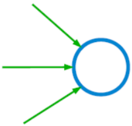
Gather-Apply-Scatter Programming Model

- Gather
  - Accumulate information about neighborhood through a generalized sum.



Gather-Apply-Scatter Programming Model

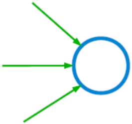
- Gather
  - Accumulate information about neighborhood through a generalized sum.



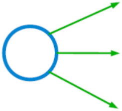
- Apply
  - Apply the accumulated value to center vertex.

Gather-Apply-Scatter Programming Model

- Gather
  - Accumulate information about neighborhood through a generalized sum.



- Apply
  - Apply the accumulated value to center vertex.
- Scatter
  - Update adjacent edges and vertices.



Data Model

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

Execution Model (1/2)

- Vertex-centric programming: implementing the `GASVertexProgram` interface (vertex-program for short).

- Similar to `Comput` in Pregel, and `update` function in GraphLab.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_{u-v}, D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_{u-v}, D_v) → (D_{u-v}^new, Accum)
}
```

Execution Model (2/2)

Input: Center vertex u
if *Cache Disabled* **then**
```
// Basic Gather-Apply-Scatter Model
```
foreach *neighbor v in gather_nbrs(u)* **do**
$a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
$(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$
else if *Cache Enabled* **then**
```
// Faster GAS Model with Delta Caching
```
if *cached accumulator a_u is empty* **then**
foreach *neighbor v in gather_nbrs(u)* **do**
$a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
$(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$
if *a_v and Δa are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
else $a_v \leftarrow \text{Empty}$

Execution Model (2/2)

Input: Center vertex u

if *Cache Disabled* **then**

```
// Basic Gather-Apply-Scatter Model
```

foreach *neighbor v in gather_nbrs(u)* **do**

$a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

$D_u \leftarrow \text{apply}(D_u, a_u)$

foreach *neighbor v scatter_nbrs(u)* **do**

$(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

else if *Cache Enabled* **then**

```
// Faster GAS Model with Delta Caching
```

if *cached accumulator a_u is empty* **then**

foreach *neighbor v in gather_nbrs(u)* **do**

$a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

$D_u \leftarrow \text{apply}(D_u, a_u)$

foreach *neighbor v scatter_nbrs(u)* **do**

$(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

if *a_v and Δa are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$

else $a_v \leftarrow$ Empty

Example: PageRank

```
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)
```

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

Scheduling (1/5)

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) ← RemoveNext (T)
    (T', S_v) ← f(v, S_v)
    T ← T ∪ T'
```

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

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

- PowerGraph inherits the dynamic scheduling of GraphLab.

Scheduling (2/5)

- Initially all vertices are active.

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 $G = (V, E, D')$

Scheduling (2/5)

- Initially all vertices are active.

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) ← RemoveNext (T)
    (T′, S_v) ← f(v, S_v)
    T ← T ∪ T′
```

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

- PowerGraph executes the vertex-program on the active vertices until none remain.

Scheduling (2/5)

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 $G = (V, E, D')$

- Initially all vertices are active.
- PowerGraph executes the vertex-program on the active vertices until none remain.
- The order of executing activated vertices is up to the PowerGraph execution engine.

Scheduling (2/5)

- Initially all vertices are active.

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 $G = (V, E, D')$

- PowerGraph executes the vertex-program on the active vertices until none remain.

- The order of executing activated vertices is up to the PowerGraph execution engine.

- Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.

Scheduling (2/5)

- Initially all vertices are active.

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**
1. $(f, v) \leftarrow \texttt{RemoveNext}\,(\mathcal{T})$
2. $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
3. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

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

- PowerGraph executes the vertex-program on the active vertices until none remain.
- The order of executing activated vertices is up to the PowerGraph execution engine.
- Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
- Vertices can activate themselves and neighboring vertices.

Scheduling (3/5)

- PowerGraph can execute both synchronously and asynchronously.
  - Bulk synchronous execution
  - Asynchronous execution

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.
- Super-step: a complete series of GAS minor-steps.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.
- Super-step: a complete series of GAS minor-steps.
- Changes made to the vertex/edge data are committed at the end of each minor-step and are visible in the subsequent minor-steps.

Scheduling - Asynchronous Execution (5/5)

- 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.

Scheduling - Asynchronous Execution (5/5)

- 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.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.

Scheduling - Asynchronous Execution (5/5)

- 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.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
  - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.

Scheduling - Asynchronous Execution (5/5)

- 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.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
  - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.
  - GraphLab implements Dijkstras solution, where forks are acquired sequentially according to a total ordering.

Scheduling - Asynchronous Execution (5/5)

- 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.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
  - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.
  - GraphLab implements Dijkstras solution, where forks are acquired sequentially according to a total ordering.
  - PowerGraph implements Chandy-Misra solution, which acquires all forks simultaneously.

Delta Caching (1/2)

- Changes in a few of its neighbors $\rightarrow$ triggering a vertex-program
- The gather operation is invoked on all neighbors: wasting computation cycles

Delta Caching (1/2)

- Changes in a few of its neighbors → triggering a vertex-program
- The gather operation is invoked on all neighbors: wasting computation cycles
- Maintaining a cache of the accumulator a_v from the previous gather phase for each vertex.
- The scatter can return an additional Δa, which is added to the cached accumulator a_v.

Delta Caching (2/2)

Input: Center vertex u

if *Cache Disabled* **then**

- `// Basic Gather-Apply-Scatter Model`
- **foreach** *neighbor v in gather_nbrs(u)* **do**
  - $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$
- $D_u \leftarrow \text{apply}(D_u, a_u)$
- **foreach** *neighbor v scatter_nbrs(u)* **do**
  - $(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

else if *Cache Enabled* **then**

- `// Faster GAS Model with Delta Caching`
- **if** *cached accumulator a_u is empty* **then**
  - **foreach** *neighbor v in gather_nbrs(u)* **do**
    - $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$
- $D_u \leftarrow \text{apply}(D_u, a_u)$
- **foreach** *neighbor v scatter_nbrs(u)* **do**
  - $(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$
  - **if** a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  - **else** $a_v \leftarrow \text{Empty}$

Delta Caching (2/2)

Input: Center vertex u
if *Cache Disabled* **then**
```
// Basic Gather-Apply-Scatter Model
```
foreach *neighbor v in gather_nbrs(u)* **do**
$a_u \leftarrow \mathrm{sum}(a_u, \mathrm{gather}(D_u, D_{u-v}, D_v))$
$D_u \leftarrow \mathrm{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
$(D_{u-v}) \leftarrow \mathrm{scatter}(D_u, D_{u-v}, D_v)$
else if *Cache Enabled* **then**
```
// Faster GAS Model with Delta Caching
```
if *cached accumulator a_u is empty* **then**
foreach *neighbor v in gather_nbrs(u)* **do**
$a_u \leftarrow \mathrm{sum}(a_u, \mathrm{gather}(D_u, D_{u-v}, D_v))$
$D_u \leftarrow \mathrm{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
$(D_{u-v}, \Delta a) \leftarrow \mathrm{scatter}(D_u, D_{u-v}, D_v)$
if *a_v and Δa are not Empty* **then** $a_v \leftarrow \mathrm{sum}(a_v, \Delta a)$
else $a_v \leftarrow$ Empty

Example: PageRank (Delta-Caching)

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

  sum(a, b):
    return a + b

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

  Scatter(i -> j):
    if R[i] changed then activate(j)
    return wij * R[i].delta
```

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

Graph Partitioning

- Vertex-cut partitioning.
- Evenly assign edges to machines.
  - Minimize machines spanned by each vertex.
- Two proposed solutions:
  - Random edge placement.
  - Greedy edge placement.

Random Vertex-Cuts

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

Greedy Vertex-Cuts (1/2)

- $A(v)$: set of machines that contain adjacent edges of v.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge 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.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge 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 neither vertex has been assigned, then assign the edge to the least loaded machine.

Greedy Vertex-Cuts (2/2)

- Coordinated edge placement:
  - Requires coordination to place each edge
  - Slower, but higher quality cuts

- Oblivious edge placement:
  - Approx. greedy objective without coordination
  - Faster, but lower quality cuts

PowerGraph Summary

- Gather-Apply-Scatter programming model
- Synchronous and Asynchronous models
- Vertex-cut graph partitioning

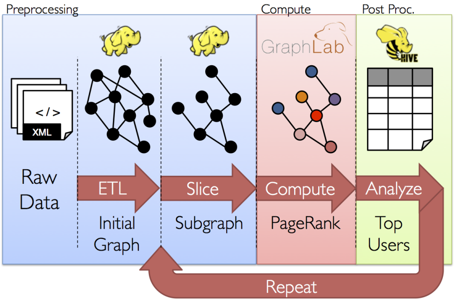
- Any limitations?

Data-Parallel vs. Graph-Parallel Computation

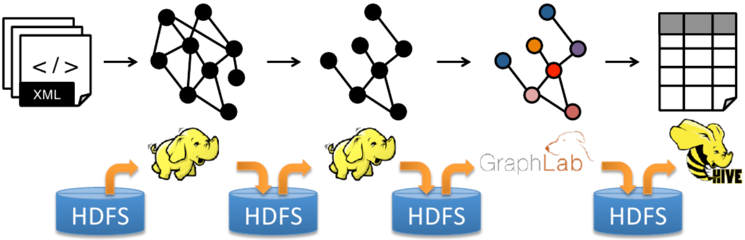
- Graph-parallel computation: restricting the types of computation to achieve performance.

Data-Parallel vs. Graph-Parallel Computation

- Graph-parallel computation: restricting the types of computation to achieve performance.
- But, the same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.

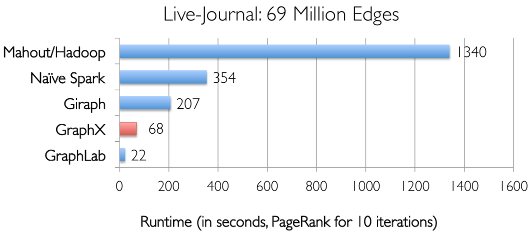


Data-Parallel and Graph-Parallel Pipeline

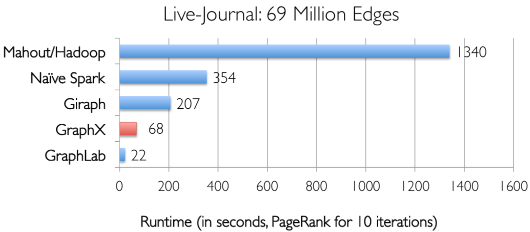


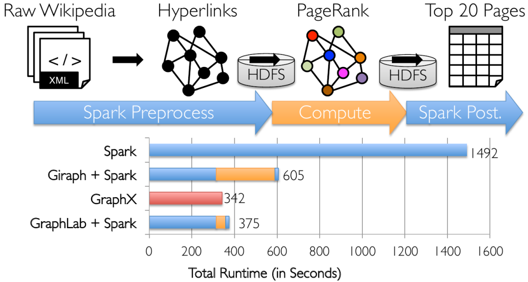
- Moving between table and graph views of the same physical data.
- Inefficient: extensive data movement and duplication across the network and file system.

GraphX vs. Data-Parallel/Graph-Parallel Systems



GraphX vs. Data-Parallel/Graph-Parallel Systems





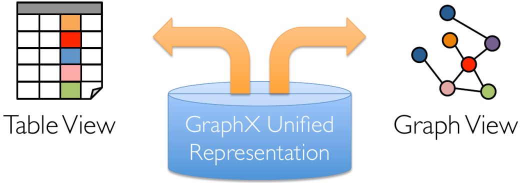
GraphX

GraphX

- New API that blurs the distinction between Tables and Graphs.
- New system that unifies Data-Parallel and Graph-Parallel systems.
- It is implemented on top of Spark.

Unifying Data-Parallel and Graph-Parallel Analytics

- Tables and Graphs are composable views of the same physical data.
- Each view has its own operators that exploit the semantics of the view to achieve efficient execution.

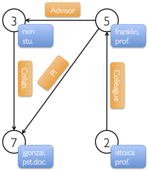


Data Model

- **Property Graph**: represented using two Spark RDDs:
  - Edge collection: VertexRDD
  - Vertex collection: EdgeRDD

```scala
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

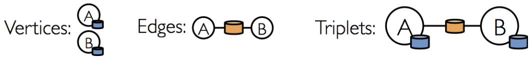
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

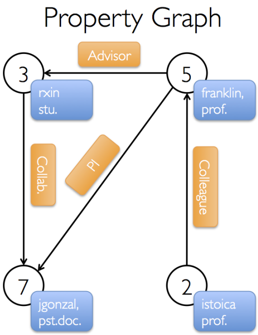
Primitive Data Types

```
// Vertex collection
class VertexRDD[VD] extends RDD[(VertexId, VD)]

// Edge collection
class EdgeRDD[ED] extends RDD[Edge[ED]]
case class Edge[ED](srcId: VertexId = 0, dstId: VertexId = 0,
                    attr: ED = null.asInstanceOf[ED])

// Edge Triple
class EdgeTriplet[VD, ED] extends Edge[ED]
```

- EdgeTriplet represents an edge along with the vertex attributes of its neighboring vertices.



Example (1/3)



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Example (2/3)

```
val sc: SparkContext

// Create an RDD for the vertices
val users: VertexRDD[(String, String)] = sc.parallelize(
    Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
          (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: EdgeRDD[String] = sc.parallelize(
    Array(Edge(3L, 7L, "collab"),     Edge(5L, 3L, "advisor"),
          Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val userGraph: Graph[(String, String), String] =
    Graph(users, relationships, defaultUser)
```

Example (3/3)

```
// Constructed from above
val userGraph: Graph[(String, String), String]

// Count all users which are postdocs
userGraph.vertices.filter((id, (name, pos)) => pos == "postdoc").count

// Count all the edges where src > dst
userGraph.edges.filter(e => e.srcId > e.dstId).count

// Use the triplets view to create an RDD of facts
val facts: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " +
    triplet.attr + " of " + triplet.dstAttr._1)

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

facts.collect.foreach(println(_))
```

Property Operators (1/2)

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

- They yield new graphs with the vertex or edge properties modified by the map function.
- The graph structure is unaffected.

Property Operators (2/2)

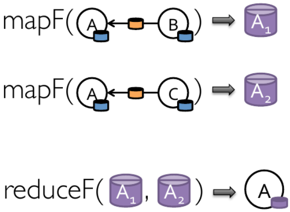
```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

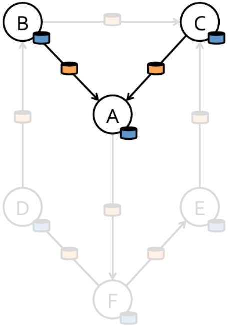
```
val newVertices = graph.vertices.map((id, attr) => (id, mapUdf(id, attr)))
val newGraph = Graph(newVertices, graph.edges)
```

- Both are logically equivalent, but the second one does not preserve the structural indices and would not benefit from the GraphX system optimizations.

Map Reduce Triplets

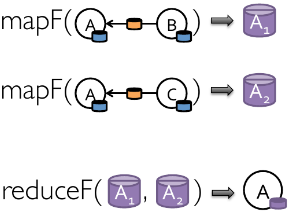
- Map-Reduce for each vertex

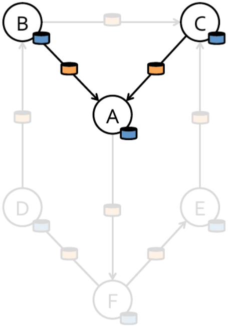




Map Reduce Triplets

- Map-Reduce for each vertex






```
// what is the age of the oldest follower for each user?
val oldestFollowerAge = graph.mapReduceTriplets(
  e => (e.dstAttr, e.srcAttr), // Map
  (a, b) => max(a, b) // Reduce
).vertices
```

Structural Operators

```
class Graph[VD, ED] {
  // returns a new graph with all the edge directions reversed
  def reverse: Graph[VD, ED]

  // returns the graph containing only the vertices and edges that satisfy
  // the vertex predicate
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]

  // a subgraph by returning a graph that contains the vertices and edges
  // that are also found in the input graph
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
}
```

Structural Operators Example

```
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)

// Run Connected Components
val ccGraph = graph.connectedComponents()

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

Join Operators

- To join data from external collections (RDDs) with graphs.

```
class Graph[VD, ED] {
  // joins the vertices with the input RDD and returns a new graph
  // by applying the map function to the result of the joined vertices
  def joinVertices[U](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, U) => VD): Graph[VD, ED]

  // similarly to joinVertices, but the map function is applied to
  // all vertices and can change the vertex property type
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
}
```

Graph Builders

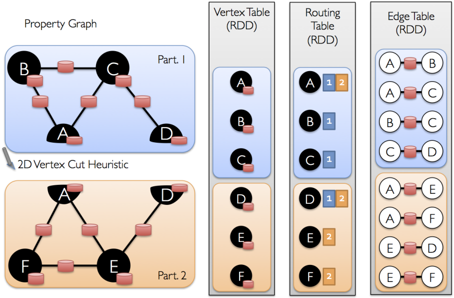
```
// load a graph from a list of edges on disk
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}

// graph file
# This is a comment
2 1
4 1
1 2
```

GraphX and Spark

- GraphX is implemented on top of Spark
- In-memory caching
- Lineage-based fault tolerance
- Programmable partitioning

Distributed Graph Representation (1/2)

- Representing graphs using two RDDs: edge-collection and vertex-collection
- Vertex-cut partitioning (like PowerGraph)



Distributed Graph Representation (2/2)

- Each vertex partition contains a bitmask and routing table.

- Routing table: a logical map from a vertex id to the set of edge partitions that contains adjacent edges.

- Bitmask: enables the set intersection and filtering.
  - Vertices bitmasks are updated after each operation (e.g., mapReduceTriplets).
  - Vertices hidden by the bitmask do not participate in the graph operations.

Summary

- Pregel
  - Synchronous model: super-step
  - Message passing
- GraphLab
  - Asynchronous model: distributed shared-memory
  - Edge-cut partitioning
- PowerGraph
  - GAS programming model
  - Vertex-cut partitioning
- GraphX
  - Unifying data-parallel and graph-parallel analytics
  - Vertex-cut partitioning

Questions?

Acknowledgements

Some pictures were derived from the Spark web site (`http://spark.apache.org/`).