GraphLab: A New Framework For Parallel Machine Learning

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)
Reminder
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

Data-Parallel

Graph-Parallel

Table

Row

Row

Row

Row

Result

Property Graph

hadoop

Spark

Pregel

GraphLab

Apache Giraph
Vertex-centric
Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)
- **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.
Data Model

- A directed graph that stores the program state, called data graph.
The **scope** of vertex $v$ is the data stored in vertex $v$, in all adjacent vertices and adjacent edges.
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.
- **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.
- **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 $\mathcal{T} = \{(f, v_1), (g, v_2), \ldots\}$

**while** $\mathcal{T}$ is not Empty **do**

1. $(f, v) \leftarrow \text{RemoveNext}(\mathcal{T})$
2. $(\mathcal{T}', S_v) \leftarrow f(v, S_v)$
3. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

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

**Input:** Data Graph $G = (V, E, D)$

**Input:** Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), \ldots\}$

**While** $\mathcal{T}$ is not Empty do

1. $(f, v) \leftarrow \text{RemoveNext}(\mathcal{T})$
2. $(\mathcal{T}', S_v) \leftarrow f(v, S_v)$
3. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

**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 $v$. 

---

Amir H. Payberah (Tehran Polytechnic)
Execution Model

**Input:** Data Graph $G = (V, E, D)$

**Input:** Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), \ldots\}$

while $\mathcal{T}$ is not Empty do

1. $(f, v) \leftarrow \text{RemoveNext}(\mathcal{T})$
2. $(\mathcal{T}', S_v) \leftarrow f(v, S_v)$
3. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

**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

```python
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.
▶ **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 $v$. 
Sequential Consistency (1/2)

- Proving the correctness of a parallel algorithm: *sequential consistency*
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.
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.
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.
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

[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
In what order should the tasks (vertex-update function pairs) be called?
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 $\mathcal{T} = \{(f, v_1), (g, v_2), \ldots\}$

while $\mathcal{T}$ is not Empty do

1. $(f, v) \leftarrow \text{RemoveNext}(\mathcal{T})$
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 $G = (V, E, D')$

▶ How to add new task in the queue?
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**.
  - $k \gg$ number of machines

- **meta-graph**: the graph of atoms (one vertex for each atom).

- **Atom weight**: the amount of data it stores.
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.
Each atom is stored as a separate file on a distributed storage system, e.g., HDFS.
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.
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
Construct a **vertex coloring**: assigns a color to each vertex such that no adjacent vertices share the same color.
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.
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.
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)
- 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**)
• 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
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 \textit{periodically} signals all computation activity to \texttt{halt}.

- Then \textit{synchronizes all caches} (ghosts) and \texttt{saves to disk} all data which has been modified since the last snapshot.

- \textbf{Simple}, but eliminates the systems advantage of \textit{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.

```plaintext
if v was already snapshotted then
   Quit
Save Dv // 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 Du→v if edge u → v exists
      Save Dv→u if edge v → 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?