Graph Processing

Connor Gramazio
Spiros Boosalis
why not MapReduce?

semantics: awkward to write graph algorithms

efficiency: mapreduces serializes state (e.g. all nodes and edges) while pregel keeps state local (e.g. nodes stay on same machine, network only passes messages)
why not a single machine?

large graphs won’t fit in memory
why not other graph systems?

they don't support things like fault-tolerance
why not just use own infrastructure?

when writing a new (parallel, graph) algorithm, most of the work will be re-implementing infrastructure just to represent the graph and execute the algorithm (rather than writing the algorithm itself).
single-source shortest paths for log-normal random graphs (mean out-degree of 128) with 800 worker tasks on 300 multicore machines

1,000,000,000 nodes in about 10min
nodes and edges

node has state
node has out-edges
node's out-edges have state
computation

node gets messages from superstep S-1
node may send messages to superstep S+1
node may mutate its state
node may mutate its out-edges' state
node may change (local?) topology
changing graph topology

e.g. clustering replaces a cluster nodes with a single node

e.g. MST removes all but the tree edges
template <typename VertexValue,
         typename EdgeValue,
         typename MessageValue>

class Vertex {
public:
    virtual void Compute(MessageIterator* msgs) = 0;
    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();
    void SendMessageTo(const string& dest_vertex,
                        const MessageValue& message);
    void VoteToHalt();
};
class PageRankVertex : public Vertex<double, void, double> {
    public:
        virtual void Compute(MessageIterator* msgs) {
            if (superstep() >= 1) {
                double sum = 0;
                for (; !msgs->Done(); msgs->Next())
                    sum += msgs->Value();
                *MutableValue() =
                    0.15 / NumVertices() + 0.85 * sum;
            }
            if (superstep() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue() / n);
            } else {
                VoteToHalt();
            }
        }
    }
computation: halting

node receives message -> activate node
node votes to halt -> deactivate node

Pregel program halts when ...
every node is inactive
and no messages were sent
computation: parallelism

message passing model

machines store same nodes throughout

network only passes messages
computation: parallelism

synchronous across “supersteps”
(i.e. iteration S+1 waits on iteration S)

asynchronous across “steps”
(same code on different nodes can be run concurrently)
fault-tolerance: checkpointing

on some supersteps, each worker persists state

master doesn't ping worker -> worker process terminates
(e.g. preemption by higher priority job)

worker doesn't pong master -> master marks worker as failed
(e.g. hardware failure)
-> restore checkpoint
partitioning

node id => index

default: hash(id) mod |machines|

user-defined partition functions must deterministically partition a node to a machine given only its unique identifier
development

progress monitoring
unittesting framework
single-machine mode for prototyping/debugging
Open Source

Apache Giraph
GPS (Stanford)
Distributed GraphLab
Distributed GraphLab

A Framework for Machine Learning and Data Mining in the Cloud

Purpose:
Better support for machine learning and data mining (MLDM) parallelization and execution at scale
PageRank Example (GraphLab2)

\[ R[i] = \frac{1 - \alpha}{N} + \alpha \sum_{\text{j links to } i} \frac{R[j]}{\text{OutLinks}[j]} \]

*i* is the webpage, *α* is the *random reset* probability and *N* is the number of webpages.

```cpp
PageRank_vertex_program(vertex i) {
  // (Gather) Compute the sum of my neighbors rank
  double sum = 0;
  foreach(vertex j : in_neighbors(i)) {
    sum = sum + j.rank / num_out_neighbors(j);
  }
  // (Apply) Update my Rank (i)
  i.old_rank = i.rank;
  i.rank = (1-\alpha)/num_vertices + \alpha*sum;
  // (Scatter) If necessary signal my neighbors to recompute their rank
  if(abs(i.old_rank - i.rank) > EPSILON) {
    foreach(vertex j : out_neighbors(i)) signal(j);
  }
}
```
What MLDM properties need support?

- Graph structured computation
- Asynchronous iterative computation
- Dynamic computation
- Serializability
Parts of Graph Lab

1. Data graph
2. Update function/Execution
3. Sync operation
Data graph

Data graph = program state
G = (V, E, D), where D = user defined data
  - data: model parameters, algorithm state, statistical data

Data can be assigned to vertices and edges in anyway they see fit
Graph structure cannot be changed in execution
Data graph PageRank example
Update Function

Represents user computation

Update: $f(v, S_v) \rightarrow (T, S_v')$

$S_v = $ data stored in $v$, and in adjacent vertices and edges

$T = $ set of vertices that are eventually executed by update function later
Update function, continued

Update: \( f(v, S_v) \rightarrow (T, S_v') \)

Can schedule only vertices that undergo substantial change from the function
Serializable execution

**Full consistency model:** scopes of concurrent execution functions never overlap

**Edge consistency model:** each update has exclusive write access to its vertex and adjacent edges; only read access to adjacent vertices

**Vertex consistency model:** allows all update functions to be run in parallel
Algorithm 2: GraphLab Execution Model

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

**Input:** Initial vertex set $\mathcal{T} = \{v_1, v_2, \ldots\}$

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

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

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

Run-time determines best ordering of vertices
→ Minimizes things like latency
All vertices in T must be eventually executed
Sync Operation

Concurrently maintains global values

Sync operation: associative commutative sum

\[ Z = \text{Finalize} \left( \bigoplus_{v \in V} \text{Map}(S_v) \right) \]

Differences from Pregel:
1. Introduces Finalize stage
2. Runs sync continuously in the background to maintain updated estimates
Distributed GraphLab Design

Shared memory design
Distributed in-memory setting
  • graph and all program state in RAM
Distributed Data Graph

1. Over-partition graph into $k$ parts, $k >> \# \text{ of machines}$

2. Each part is called an **atom**

3. Connectivity stored as a **meta-graph** file
   
   $MG = (V, E)$
   
   $V = \text{all atoms}, E = \text{atom connectivity}$

4. Balanced partition of meta-graph over the physical machines
What is an atom?

Binary compressed journal of commands

- AddVertex(5000, vdata)
- AddEdge(42→ 314, edata)

Ghosts: set of vertices and edges adjacent to partition boundary

- Ghosts are used as caches for their true atom counterparts
Distributed Data Graph, review

- Atom: each partition
- Meta-graph: graph of atom connectivity
- Balanced partitioning using meta-graph
- Ghosts used as caches
- Cache coherence management using simple versioning system
Distributed GraphLab Engines

**Engine:** emulation of execution model
- executes update functions and syncs
- maintains set of scheduled vertices $T$
  - scheduling = implementation specific
- ensures serializability for given a consistency model
Your two engine options

Chromatic Engine
- Executes T partially asynchronously
- Skipping detailed explanation

Locking Engine
- Executes T fully asynchronously and supports vertex priorities

Note: more detail available if wanted
System Design

Initialization Phase

(MapReduce) Graph Builder
- Parsing + Partitioning
- Atom Collection
- Index Construction

Distributed File system
- Raw Graph Data
- Raw Graph Data

GraphLab Execution Phase

Cluster
- TCP RPC Comms
- Monitoring + Atom Placement
  - GL Engine
  - GL Engine

Distributed File system
- Atom Index
- Atom File
- Atom File
- Atom File
- Atom File
- Atom File
- Atom File
- Atom File
Applications

Netflix movie recommendations
Video Co-segmentation (CoSeg)
Named Entity Recognition (NER)
references

*Pregel: A System for Large-Scale Graph Processing*

*Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud*
Start extra slides
Why is GraphLab good?

Sequential shared memory abstraction where each vertex can read and write to data on adjacent vertices and edges
Runtime is responsible for ensuring a consistent parallel execution
Allows focus on sequential computation not the parallel movement of data
Graph structured computation

- Data dependency is important for MLDM
- Graph parallel abstraction = good
  - GraphLab and Pregel = good
  - MapReduce = bad
Asynchronous iterative computation

- Update parameters using most recent parameter values as input
- Asynchronism sets GraphLab apart from iterative MapReduce alternatives like Spark. And also from abstractions like Pregel
Dynamic computation

Parameters can converge at uneven rates
→ Some parameters need less updating
GraphLab allows prioritization of parameters that take longer to converge
→ Can also pull info from neighboring vertices
Relaxes scheduling requirements to make distributed FIFO and priority scheduling efficient
Serializability
Distributed Data Graph

Two-phased partitioning for load balancing on arbitrary cluster sizes
Partitioning strategy allows the same graph partition computation to be reused for different numbers of machines without a full repartitioning step
Chromatic Engine

Based on vertex coloring

• Each color assigned a color s.t. no adjacent vertices share the same color
• **Color-step**: update all vertices in a single color and communicating changes
• Sync operation run between color-steps
• Satisfies consistency constraints
Coloring is perfect... right?

Nope. NP-hard.

Reasonable coloring via heuristics

In practice most MLDM problems have trivial colorings (e.g., bipartite graphs)
Chromatic Engine: Consistency

- Coloring satisfies edge consistency model
  - each update has exclusive write access to its vertex and adjacent edges; only read access to adjacent vertices
- Second-order vertex coloring satisfies full consistency model
  - scopes of concurrent execution functions never overlap
- Assigning all vertices the same color satisfies vertexy consistency model
  - allows all update functions to be run in parallel
Chromatic Engine = partial async

Executes color-steps synchronously
Changes to ghost vertices and edges are asynchronously communicated
All in all, chromatic engine is good but has restrictive scheduling
Aren’t deadlocks a problem?

Deadlocks are avoided by locking sequentially based on ordering

- ordering induced by machine ID followed by vertex ID

Each machine can only update local vertices
Locking and efficiency

Ghost caching helps
All lock requests and sync calls are pipelined

- machines can request locks and data simultaneously
- evaluate update function only when scope is ready
Fault Tolerance (very quick)

Uses asynchronous snapshotting to avoid stopping execution
Distributed Locking Engine
Distributed Locking

Extends mutual exclusion technique in shared memory engine

- each vertex gets a readers-writer lock

Each consistency model uses a different locking protocol
Locking and consistency

Vertex consistency: acquire write-lock on central vertex of each requested scope

Edge consistency: acquire a write lock on central vertex, read locks on adjacent vertices

Full consistency: acquire write locks on central vertex and all adjacent vertices
Pipelined Locking and Prefetching

Algorithm 4: Pipelined Locking Engine Thread Loop

while not done do
  if Pipeline Has Ready Vertex $v$ then
    Execute $(T', S_v) = f(v, S_V)$
    // update scheduler on each machine
    For each machine $p$, Send $\{s \in T' : \text{owner}(s) = p\}$
    Release locks and push changes to $S_V$ in background
  else
    Wait on the Pipeline

$T =$ set of scheduled vertices; $S_v =$ data associated with vertex
Performance of locking

3-D 300x300x300 mesh, connectivity between vertex and all adjacent neighbors, 512 atoms

Graph = binary Markov Random Field

Evaluate 10 iterations of loopy Belief Propagation