GraphLab and PowerGraph

Background context

- machine learning and data mining!! (MLDM)
  - scale of ML and DM problems is growing
  - sequential execution speeds have plateaued
  - therefore, we need parallel versions of ML/DM programs

- hard problem; no good approach yet
  - low-level primitives like MPI and pthreads
    - force the user to solve all the problems that distribution frameworks (like MapReduce) try to solve
    - e.g., serialization/deserialization, load balancing, locking and deadlock, data races, fault tolerance, and so on
    - but, very efficient implementations are possible if you want to put in the huge effort required
  - MapReduce
    - works great for non-iterative, embarrassingly parallel applications
    - does not have a natural "iterative graph" programming abstraction
    - fails when:
      - there are computational dependencies in the data; parallel, independent map/reduce doesn't work
      - there is an iterative structure to the computation; e.g., gradient descent. Forces the MR programmer to invoke MR iteratively themselves, and deal with scheduling, termination, and other issues.
      - computation fits in memory; common implementations of MR force outputs to disk (just an implementation shortcoming, not fundamental to MR)
  - DAGs like Dryad
    - no iteration, no dynamically prioritized computation

Why graphs?

- it turns out that many (most?) machine learning and data mining applications are naturally amenable to graph structuring
  - intuition: MLDM is often about dependencies between data – a dependency is an edge between two pieces of data; data is a vertex

- graphs permit **asynchronous** and **iterative** computation
  - synchronous: all parameters are updated simultaneously, using parameter values from the previous time step. requires a barrier, suffers from the straggler problem.
    - iterative MR is naturally synchronous.

    - asynchronous: update parameters using most recent parameter values as input.
        - naturally adapts to differences in execution speed coming from heterogeneity in hardware, network, or even the data itself (like differences in degree distributions of vertices)
- graphs permit **dynamic computation**
    - in many algorithms, iterative computation converges asymmetrically
        - e.g., pagerank: some nodes converge quickly, while others take a long time
        - if you update all parameters equally often, waste effort recomputing parameters that have already converged
- there is a notion of **serializability** for graph structured computation
    - ensuring that a parallel execution has an equivalent sequential execution
    - do this by locking vertices as needed, depending on the consistency model

GraphLab

We'll talk through three versions of GraphLab:

1. Shared-memory, multicore parallel GraphLab (i.e., one machine, no network)
2. Distributed GraphLab (i.e., many machines with a network)
3. PowerGraph (i.e., distributed graphlab optimized for powerlaw graphs)

Shared memory, multicore parallel graphlab

Data graph G = (V, E, D)

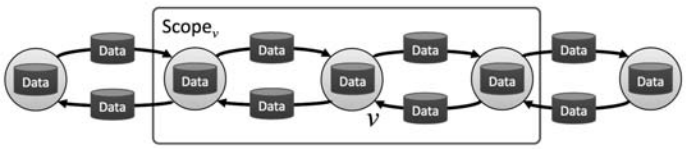
- set of vertices V, set of edges E, and user-defined data D
- data can be associated with:
    - each vertex { Dv : v in V }
    - each edge {Du→v : {u, v} in E }
- data is mutable, but the graph structure is static

Computation is encoded in "update functions."

- stateless procedure that (a) modifies data within the scope of a vertex, and (b) schedules the future execution of update functions on other vertices T

$$\textbf{Update} : f(v, \mathcal{S}_v) \rightarrow (\mathcal{S}_v, \mathcal{T})$$

- Sv: scope of vertex v is the data stored in v, as well as in all adjacent vertices and edges of v



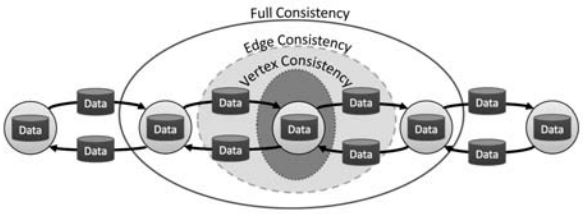
Graphlab maintains a set of vertices to be updated, and iteratively and in parallel runs update functions on them.

Algorithm 2: GraphLab Execution Model

Input: Data Graph $G = (V, E, D)$
Input: Initial vertex set $\mathcal{T} = \{v_1, v_2, ...\}$
while $\mathcal{T}$ *is not Empty* **do**
1 $v \leftarrow$ `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 $G = (V, E, D')$

Data consistency

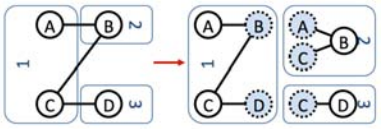
- scopes can overlap, so simultaneously executing two update functions can result in a collision / race condition
- graphlab offers three consistency models, allowing users to trade off performance and consistency as appropriate for their computation.
    - think of a consistency model as offering exclusion sets – concurrently executing update functions cannot share overlapping exclusion sets.



- using these, you can achieve sequential consistency, if any of the following are true:
    - the full consistency model is used
    - edge consistency is used, and update functions don't modify data in adjacent vertices
    - vertex consistency is used, and update functions can only modify local vertex data and read adjacent edges
- consistency models enforced with locking

<u>Distributed graphlab</u>

The graph has to be partitioned into several sets, with each set executing on a separate machine on a workstation cluster. Partitioning is done by edge-cutting – i.e., the partition boundary is a set of edges.



- need to maintain "ghost" vertices at the boundary – caches of vertex data
- introduces a cache consistency problem that they solve with versioning
- partitioning controls load balancing – want to be careful to have roughly the same number of vertices per partition, as well as the same number of ghosts
    - vertices proportional to computation
    - ghosts proportional to network load for cache updates

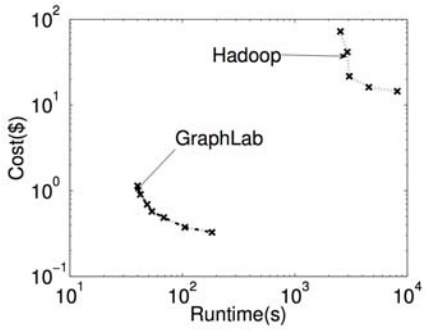
Locking becomes distributed locking

- to enforce a consistency model, need to graph a set of locks
- if need to graph a lock on an edge or vertex that is on the boundary, need to do it on both partitions involved – hence, distributed locking
- deadlock is avoided by having a canonical ordering of lock acquisition

Fault tolerance becomes an issue, since you can experience partial failure

- graphlab solves with checkpointing
- uses Lamport-Chandy algorithm to do asynchronous snapshotting, rather than synchronous "stop the world" snapshot

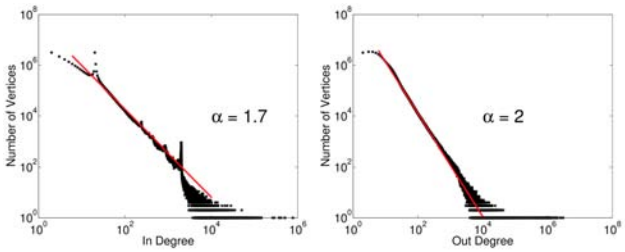
Efficiency: great!



(b) EC2 Price/Performance

Powergraph

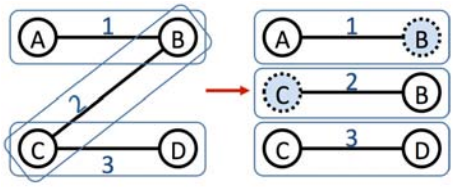
The issue: many graphs have a power-law distribution of connectivity, in which there are a few popular vertices with many, many edges, as well as a large number of unpopular vertices with few edges. Typically see a Zipf distribution of in/out-degree.



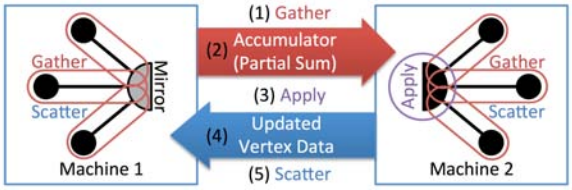
(a) Twitter In-Degree (b) Twitter Out-Degree

This causes problems for graphlab: edge cuts become hugely load imbalanced, since some partitions will have the massively popular high degree nodes. Leads to imbalance in computation (since update functions touch edges) and in network bandwidth (if a popular node is a ghost).

First innovation: instead of partitioning by cutting edges, powergraph partitions by cutting vertices. This essentially picks one partition for each edge, and allows a high degree vertex to be split across multiple partitions.



Second innovation: instead of running an update function for a vertex on a single machine, parallelize the update function across multiple machines. This lets a high degree vertex update be parallelized. Requires the "accumulator" semantics to compose partial updates.



Third innovation: instead of randomly constructing vertex cuts, uses "derandomization" of the edge placement process. Basically, for each edge, consider whether vertices on the edge have already been assigned to machines, and attempt to place the edge with some

locality so that you minimize cross-machine placement. (Details in the paper.) Attempts to minimize cross-machine coordination and data flow.

Does it work? Yes!!

PageRank	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [22]	198s	–	1.1B	50x8
Spark [37]	97.4s	40M	1.5B	50x2
Twister [15]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [36]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

LDA	Tok/sec	Topics	System
Smola et al. [34]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16