- Midterm update → Tonight !!

- Course Project reminders

  - Discussion groups
    - Piazza Group Number: email id
    - You can join the corresponding group

  - OH slot: Start this from next week!

extra
Scalable Storage Systems
Datacenter Architecture
Resource Management
Computational Engines
Applications
- Machine Learning
- SQL
- Streaming
- Graph

Naiad, Spark Streaming
GRAPH DATA

Datasets

1. Social network "friend graph"
2. Internet → web pages, link
   → Hosts/are connected
   → Domains
3. Flights → (source, dst) pairs / road network
4. Paper1 cites Paper2 cites others etc.
5. Software dependencies
   → Spark → Akka actor framework
   → Python

Application

→ recommendation
→ Page Rank
→ Shortest path / traversal algorithm
→ TSP
GRAPH ANALYTICS

Perform computations on graph-structured data

Examples
PageRank
Shortest path
Connected components

…

SQL queries on tabular data
Message combiner(Message m1, Message m2):
    return Message(m1.value() + m2.value());

void PregelPageRank(Message msg):
    float total = msg.value();
    vertex.val = 0.15 + 0.85*total;
    foreach(nbr in out_neighbors):
        SendMsg(nbr, vertex.val/num_out_nbrs);

(1) Get messages from Neighbors
(2) Combiner coalesces messages
(3) Computation using the combined message
(4) Send out msgs to Neighbors
repeat till convergence
NATURAL GRAPHS

(1) Distribution of degree is skewed!
   - most vertices have small degree
   - some vertices have very high degree

(2) High degree vertices lead to skew in
   \[ \rightarrow \text{communication} \]
   \[ \rightarrow \text{memory pressure (state)} \]
   \[ \rightarrow \text{computation} \]

Hard to partition such graphs

(a) Twitter In-Degree

\[ \alpha = 1.7 \]
POWERGRAPH

Programming Model: 
Gather-Apply-Scatter

Better Graph Partitioning 
with vertex cuts

Distributed execution (Sync, Async)
GATHER-APPLY-SCATTER

Gather: Accumulate info from nbrs

Apply: Accumulated value to vertex

Scatter: Update adjacent edges, vertices

// gather_nbrs: IN_NBRSS
gather(Du, D(u,v), Dv):
return Dv.rank / #outNbrs(v)

→ sum(a, b): return a + b

apply(Du, acc):
new = 0.15 + 0.85 * acc
Du.delta = (rnew - Du.rank) / #outNbrs(u)
Du.rank = rnew

// scatter_nbrs: OUT_NBRSS
scatter(Du, D(u,v), Dv):
if(|Du.delta| > ε) Activate(v)
return delta
**EXECUTION MODEL, CACHING**

Active Queue

```
| g(v1) | s(v1) | a(v1) | g(v2) | a(v2) | g(v3) | a(v3) |
```

Delta caching → optimization to speed up

- Cache accumulator value for vertex
- Optionally scatter returns a delta

Accumulate deltas

Could run into race conditions

Single machine | Graph \( G \)

Could run into race conditions

\( \bot \) vertex

Ftl

Single machine

Hath

na

li#eaon-atel:oi

Huyser

\( \text{fml/aaufaa.fau4aIy} \)

Eat

Eat

\( \text{scatter UD} \)

\( \text{scatter UD} \)

\( \text{scatter UD} \)

Sync. vs. Async.
SYNC VS ASYNC

Sync Execution

- Gather for all active vertices,
  followed by Apply, Scatter
- Barrier after each minor-step

Async Execution

- Execute active vertices,
  as cores become available
- No Barriers! Optionally serializable

Queue of operations

V1 operations / Vertex Vz

GUD neighbor ensures Vertenl updates vertex state

Barrier state update

update Acu

local Alva is visible in
state Barrier next minor step

ACVD so

GUD edge

A (V1) → updates vertex state
A (v2)
G (V3)
DISTRIBUTED EXECUTION

Symmetric system, no coordinator

Load graph into each machine

Communicate across machines to spread updates, read state
GRAPH PARTITIONING

(a) Edge-Cut

- Every vertex is placed on a machine
- Edges might span across them
- Natural graphs → lots of edges across machines!

(b) Vertex-Cut

- Every edge is placed on a machine
- Vertices might be across machines
- Better balance for natural graphs
RANDOM, GREEDY OBLIVIOUS

Three distributed approaches:

Random Placement

1. Stream through edges
2. Send edge to a random machine

Coordinated Greedy Placement

1. Send edge to a machine that already has one of its vertices

Oblivious Greedy Placement

1. Greedy in parallel so you don't have perfect knowledge of vertex → machine
OTHER FEATURES

Async Serializable engine
- Preventing adjacent vertex from running simultaneously
  → Acquire locks for all adjacent vertices

Fault Tolerance
- Checkpoint at the end of super-step for sync

\[
\begin{align*}
\text{Gather} & \quad \text{Apply} \\
\text{S catter} & \quad \text{S catter}
\end{align*}
\]
SUMMARY

Gather-Apply-Scatter programming model
Vertex cuts to handle power-law graphs
Balance computation, minimize communication
DISCUSSION

https://forms.gle/rKB5hcJgT4NQsFqq8
Consider the PageRank implementation in Spark vs synchronous PageRank in PowerGraph. What are some reasons why PowerGraph might be faster?

- Activate ensures no wasteful computation
- Fine-grained communication in PowerGraph
- Better partitioning!
- Delta caching → avoids computation
(a) Twitter PageRank Runtime
(b) Twitter PageRank Comms
(c) Twitter PageRank Delta Cache
Next class: GraphX

Partitioning in Spark → Co-partitioning

Powergraph has methods to pick which vertices go in a partition