CS 744: POWERGRAPH

Shivaram Venkataraman

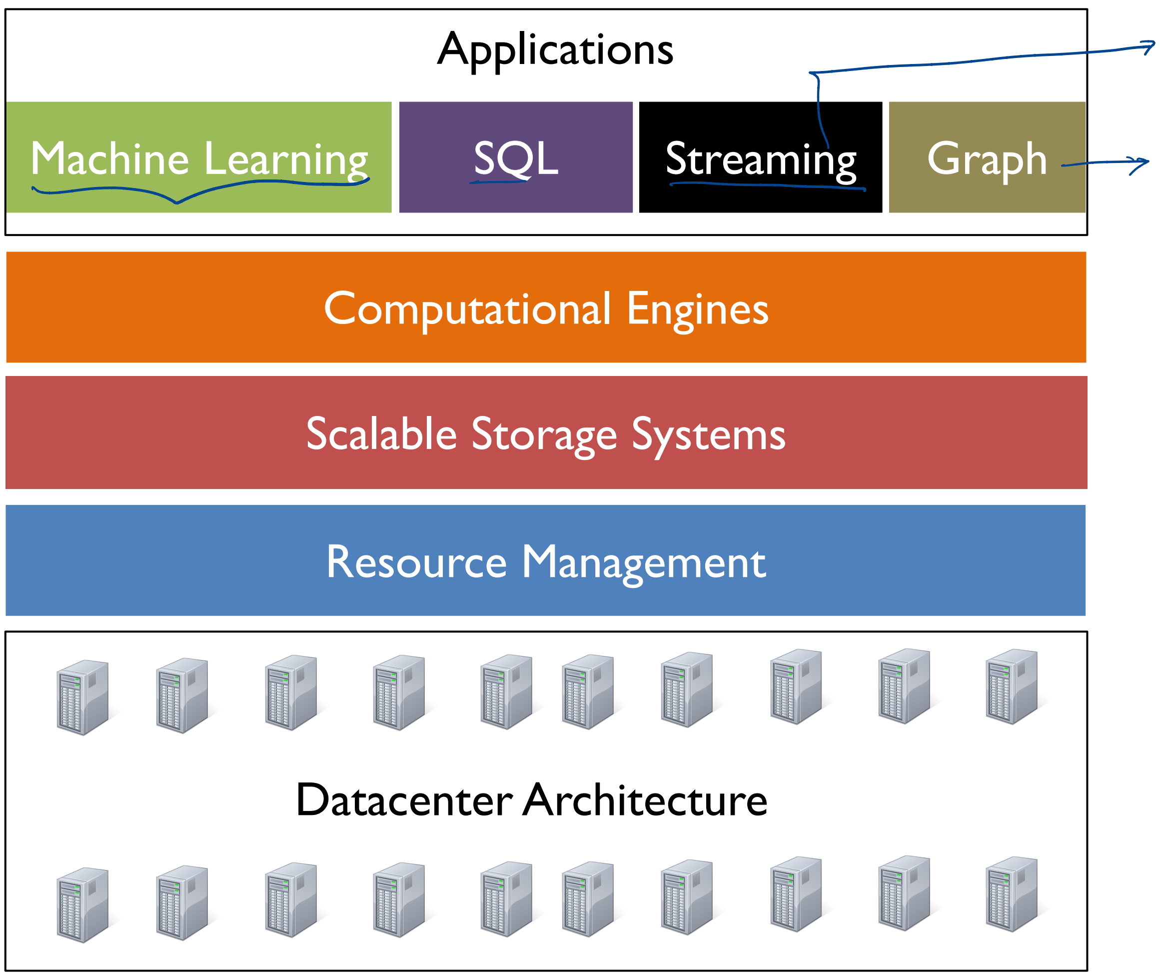
Fall 2020

ADMINISTRIVIA

- 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



Applications

Machine Learning

SQL

Streaming

Graph

Computational Engines

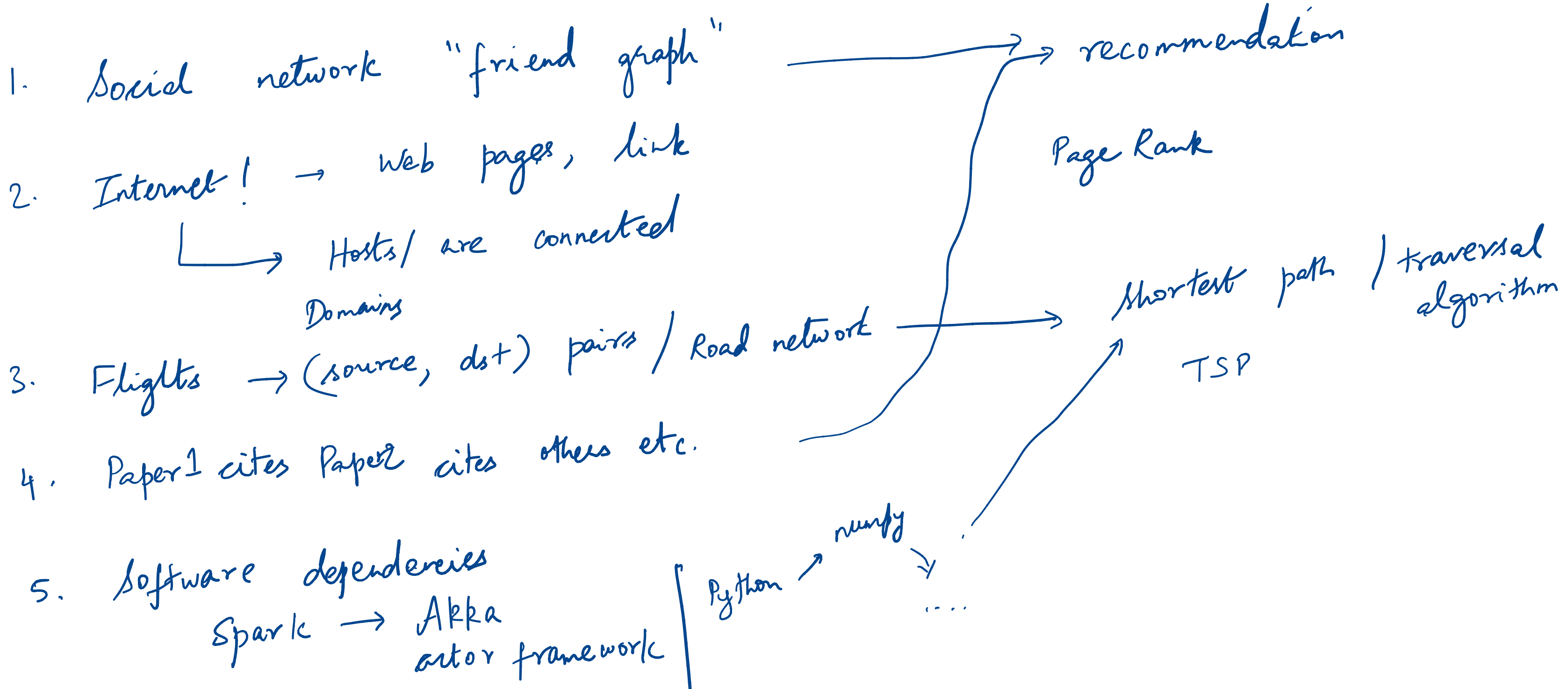
Scalable Storage Systems

Resource Management

Datacenter Architecture

Naiad, Spark Streaming

GRAPH DATA



Datasets

1. Social network "friend graph" → recommendation

2. Internet! → web pages, link
   - → Hosts/ are connected
   - Domains

3. Flights → (source, dst) pairs / Road network → Shortest path / traversal algorithm

4. Paper1 cites Paper2 cites others etc.

5. Software dependencies
   - Spark → Akka actor framework | Python → numpy → ....

Application

recommendation

Page Rank

Shortest path / traversal algorithm

TSP

GRAPH ANALYTICS

→ SQL queries on Tabular data

Perform computations on graph-structured data

Examples

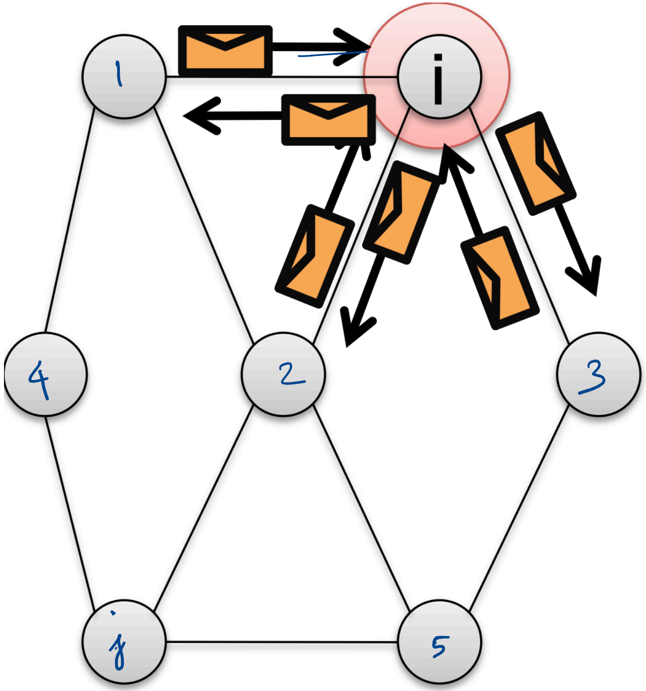
- PageRank
- Shortest path
- Connected components →
- …

row of this table

⇒ operations

PREGEL: PROGRAMMING MODEL

→ "Think like a vertex"



```
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);
```

State of this vertex

(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

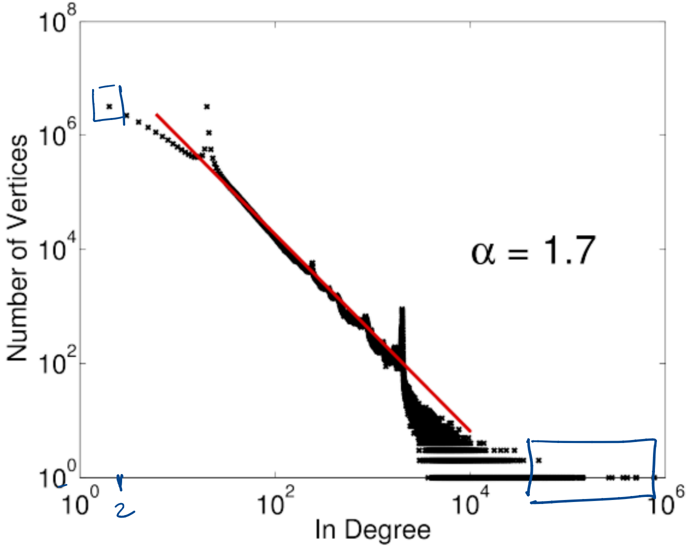
(2) High degree vertices lead to skew in

↳ Communication

↳ Memory pressure (state)

↳ Computation

Hard to partition such graphs



(a) Twitter In-Degree

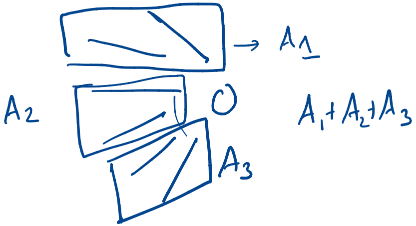
POWERGRAPH

Programming Model:
Gather-Apply-Scatter

→ Execution

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 returns an accumulator
- You can combine accumulators!
- similar to reduction in Spark

Activate on a neighboring vertex from scatter → Allows us to only process necessary vertices in next iteration

state vertex

edge state

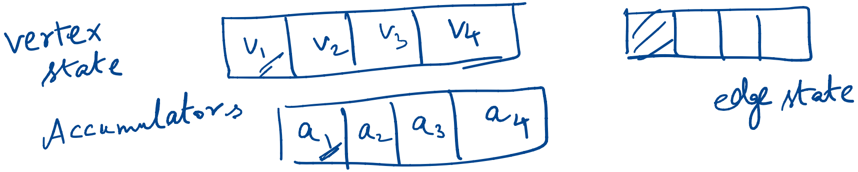
state vertex

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

sum(a, b): return a+b

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

// scatter_nbrs: OUT_NBRS
scatter(Du,D(u,v),Dv):
    if(|Du.delta|> ε) Activate(v)
    return delta
```

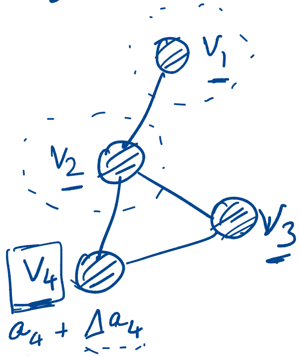
→ change in value

EXECUTION MODEL, CACHING

Could run into race conditions



Single machine. Graph G



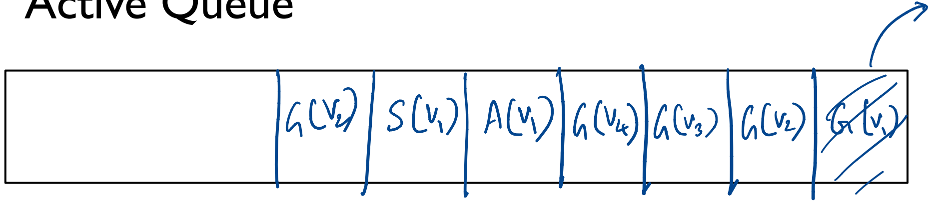
gather(v1.state,
v2.state,
v1v2 edge.state)

apply(v1, a1)

scatter(v1)

Activate(v2)

Active Queue



Delta caching → optimization to speed up gather

→ Cache accumulator value for vertex

→ Optionally scatter returns a delta

Accumulate deltas

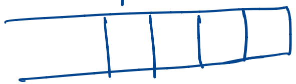
vertex state

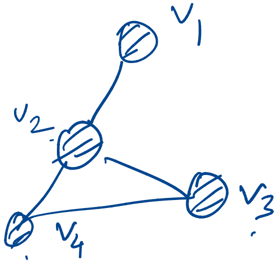
→ Is the latest value available to future operations

Sync vs. Async

SYNC VS ASYNC

Queue of operations





Sync Execution

→ Gather for all active vertices,

Barrier

followed by Apply, Scatter

Barrier after each minor-step

read neighbor state

$G(v_1)$
$G(v_2)$
⋮

Barrier

update local state

$A(v_1)$
$A(v_2)$
⋮

Barrier

$S(v_1)$
⋮

ensures vertex/edge state update is visible in next minor step

Async Execution

Execute active vertices,

as cores become available

No Barriers! Optionally serializable

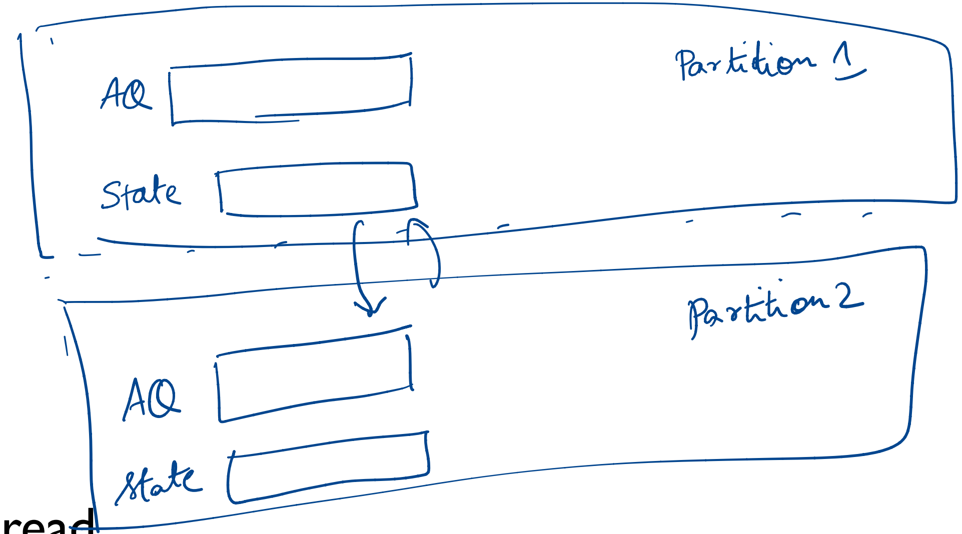
$G(v_1)$
$A(v_1)$ → updates vertex state
$G(v_2)$
$A(v_2)$
⋮
$G(v_3)$

DISTRIBUTED EXECUTION

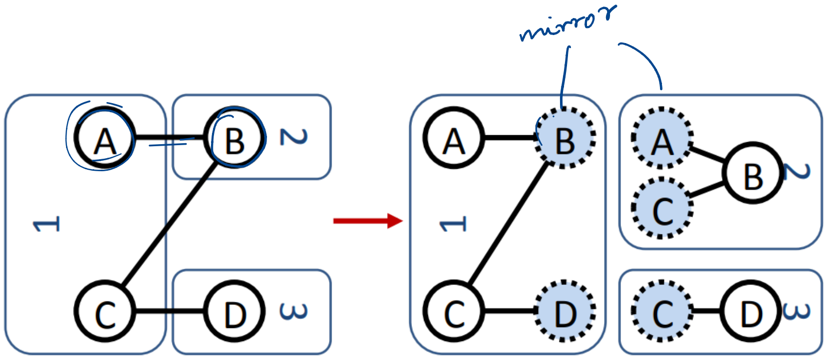
Symmetric system, no coordinator

Load graph into each machine

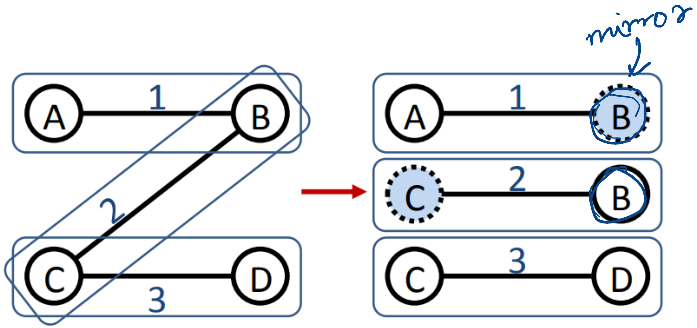
Communicate across machines to spread updates, read state



GRAPH PARTITIONING



(a) Edge-Cut

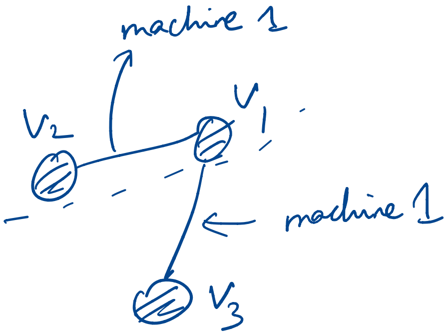


(b) Vertex-Cut

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

- → 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

↳ stream through edges
send edge to a random machine

Coordinated Greedy Placement

↳ send edge to a machine that already has
one of its vertices

Oblivious Greedy Placement

↳ 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

Gather -
Apply -
Scatter

Super step

SUMMARY

Gather-Apply-Scatter programming model

Vertex cuts to handle power-law graphs

Balance computation, minimize communication

DISCUSSION

https://forms.gle/rKB5hcJgT4NQsFgq8

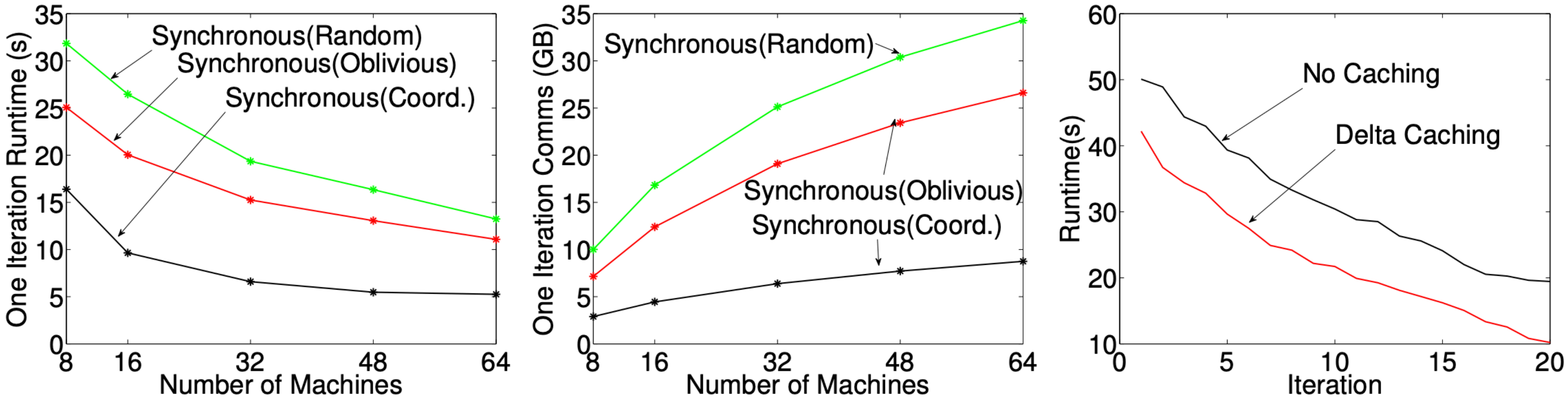
Consider the PageRank implementation in Spark vs synchronous PageRank in PowerGraph. What are some reasons why PowerGraph might be faster?

→ Activate ensures no wasteful computation

→ Fine-grained communication in Power-graph

↳ Better partitioning !

→ Delta caching → avoids computation



(a) Twitter PageRank Runtime (b) Twitter PageRank Comms (c) Twitter PageRank Delta Cache

NEXT STEPS

Next class: GraphX

Partitioning in Spark → Co-partitioning

Vertex state and Neighbors state
Preserve the partitions across iterations!

Spark

Powergraph has methods to pick what vertices go in a partition