Welcome back!

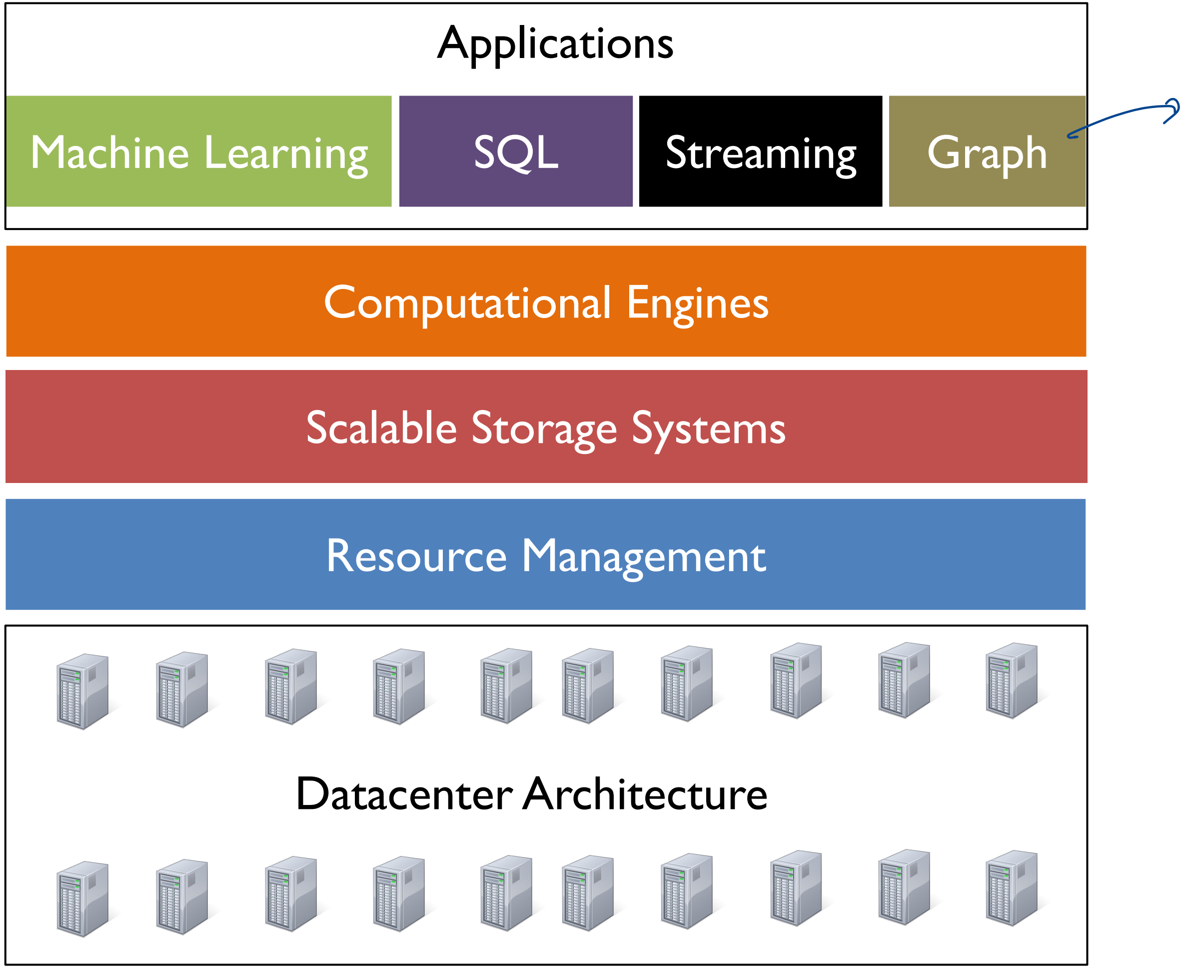
CS 744: POWERGRAPH

Shivaram Venkataraman

Fall 2022

ADMINISTRIVIA

- Midterm grading in progress → end of this week
- Course Project: GCP credits


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

GRAPH DATA

- Web graph hyperlinks

Datasets

- Social network graph
- Maps. Locations → vertices
  Roads → edges
- Knowledge graph

  Obama $\xrightarrow{\text{was president}}$ 2008-2014
- Chemicals / Molecules

Application

- Search (important or relevant pages)
- Recommendation
- Classify chemical structures

GRAPH ANALYTICS

Perform computations on graph-structured data

Examples

- PageRank
- Shortest path
- Connected components
- …

Serving graph data

"OLTP"

→ traverse graph return results

→ latency sensitive

Analytics

Classic

PageRank

Connected Components

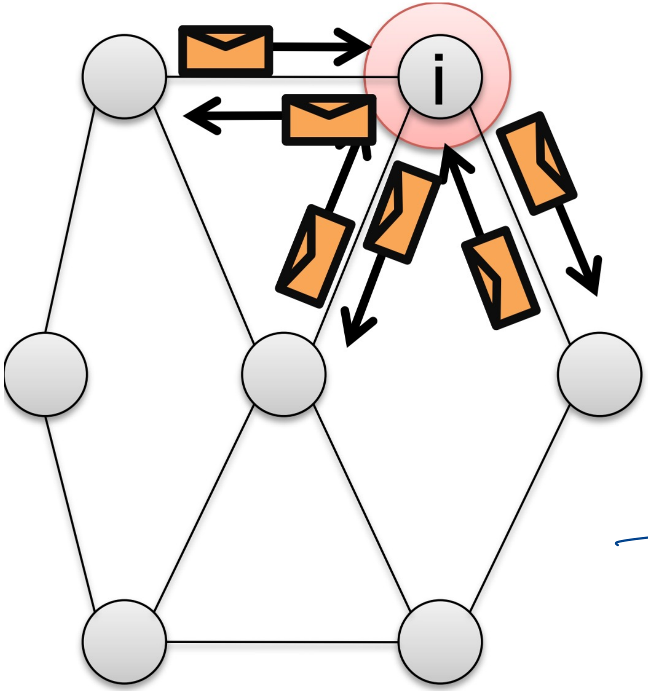
ML

Apply ML on graph structured data

PREGEL: PROGRAMMING MODEL

~ 2008-2009

"Think like a vertex"



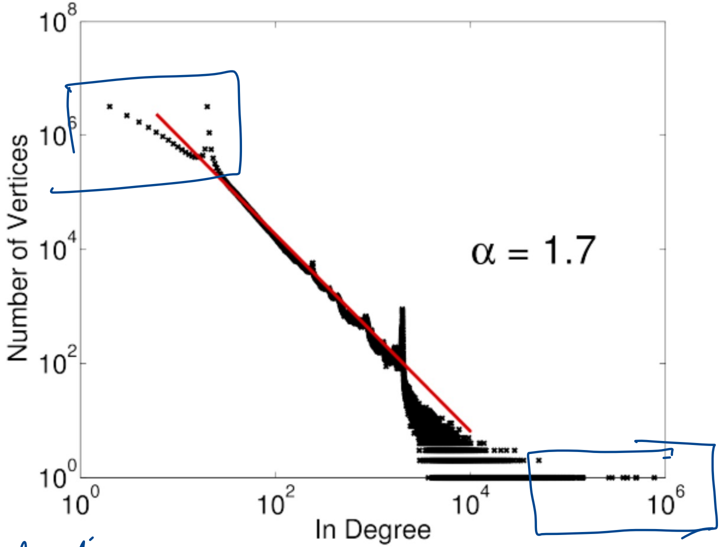

```
Message combiner(Message m1, Message m2):
    return Message(m1.value() + m2.value());

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);
```

1. Express graph algorithms,
2. Recv inputs from nbrs → Comp on the vertex → Send msg to nbrs

NATURAL GRAPHS

1. Exp. distribution of nbr hood sizes
   - few vertices very large degree
   - large ~~x~~ number with very small

2. Lack of symmetry
   - Imbalance in work done per vertex
   - Imbalance ~~le~~ load to → low utilization
     ↳ stragglers



(a) Twitter In-Degree

POWERGRAPH

Programming Model:
Gather-Apply-Scatter

→ builds on think like a vertex

Sync / Async execution

→ Single machine

Better Graph Partitioning
with vertex cuts

→ Distributed

GATHER-APPLY-SCATTER

Gather: Accumulate info from nbrs

Apply: Accumulated value to vertex

Scatter: Update adjacent edges

Gather → input vertex, edge, states
returns accumulator

Apply → return / update vertex state

Scatter → takes update vertex state
and can update nbrs.

state for source

dest

edge state

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

Accumulator

```
sum(a, b): return a+b
```

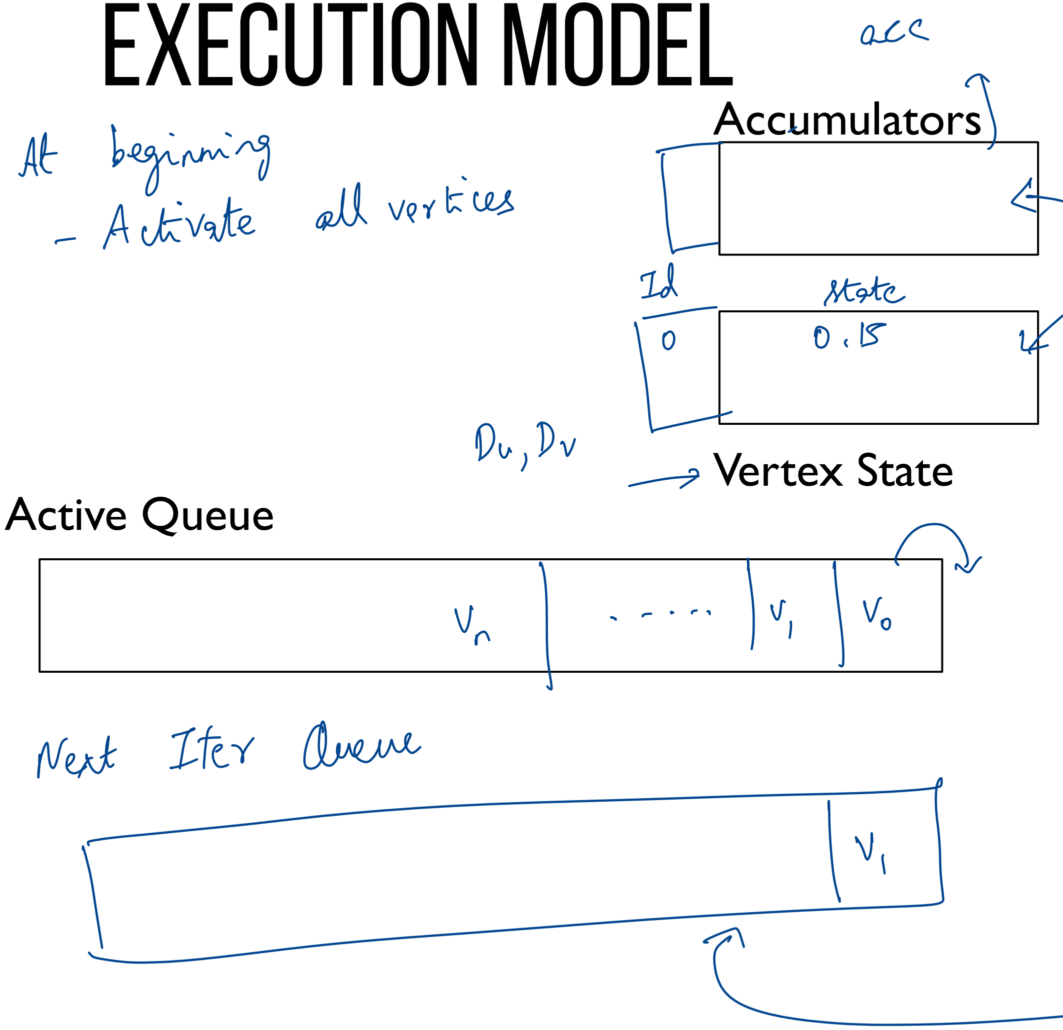
Vertex

accumulated results of gather

```
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
```

EXECUTION MODEL

At beginning
- Activate all vertices



Gather

gather (V_0)

[list all nbrs (V_0)]

get their state

accumulate

Apply

apply (V_0)

read acc for V_0

read state for V_0

Comp. ← write back updated

Scatter

activate nbrs in next iteration

CACHING

Accumulators

Persist across iters (reuse)

Vertex State

Active Queue

Avoid running gather for that vertex

Delta caching

Cache accumulator value for vertex

Re use acc computed in previous iteration

Optionally scatter returns a delta

Accumulate deltas

If few nbrs have changed

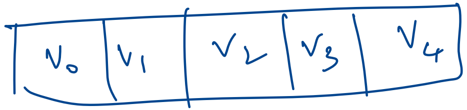
SYNC VS ASYNC

Sync Execution

Gather for all active vertices,

followed by Apply, Scatter

Barrier after each minor-step



Gather (V_0)
Gather (V_1) } minor step

Apply (V_0)

Scatter (V_0)

All the reads happen before updates

Async Execution

Execute active vertices,

as cores become available

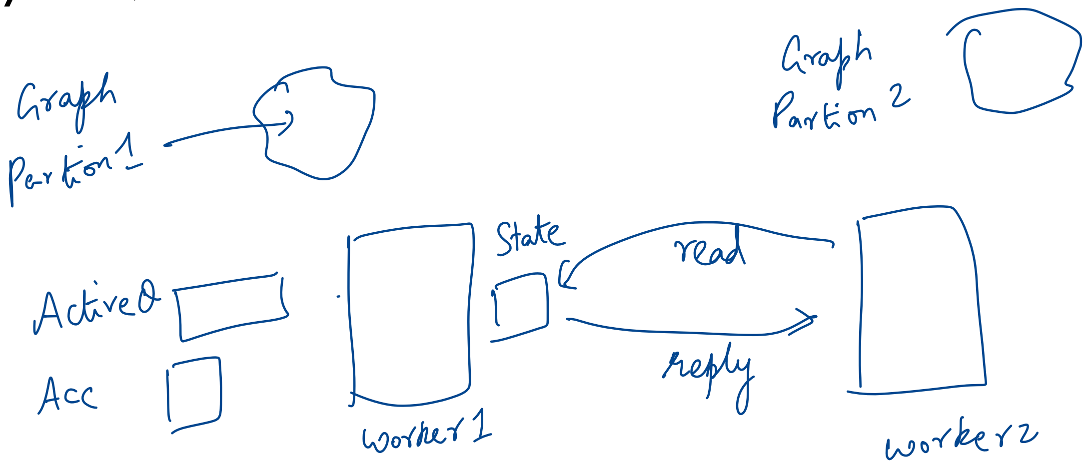
No Barriers! Optionally serializable

Async + Serializable

- Ensure that connected vertices are not processed concurrently

DISTRIBUTED EXECUTION

Symmetric system, no coordinator



Partition graph across machines

Communicate to spread updates, read state

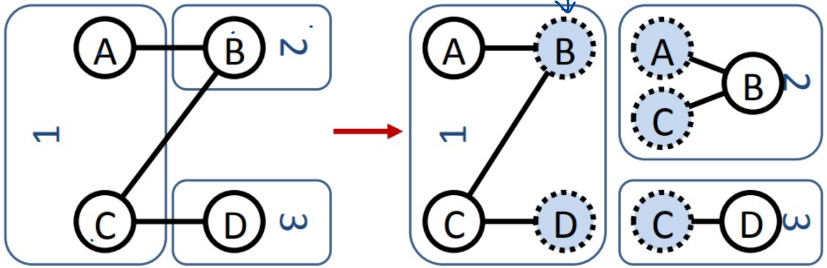
GRAPH PARTITIONING

"read-only" replica

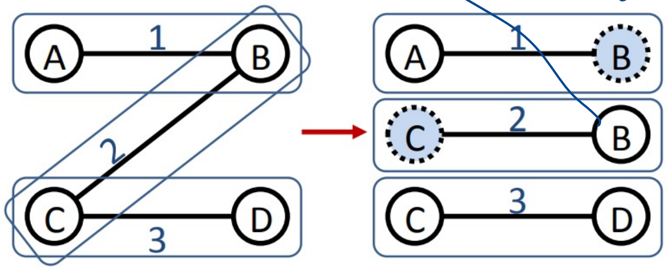
ghost vertices

primary

ghost vertices



(a) Edge-Cut



(b) Vertex-Cut

- Assign a vertex to a machine
  - Implies edges are cut across machines. Minimize this
- Induce imbalance for natural graphs

- Assign edge to particular machine
- * When vertex is on many machines, one primary

RANDOM, GREEDY OBLIVIOUS

Assign edges to machines

Three distributed approaches:

Random Placement

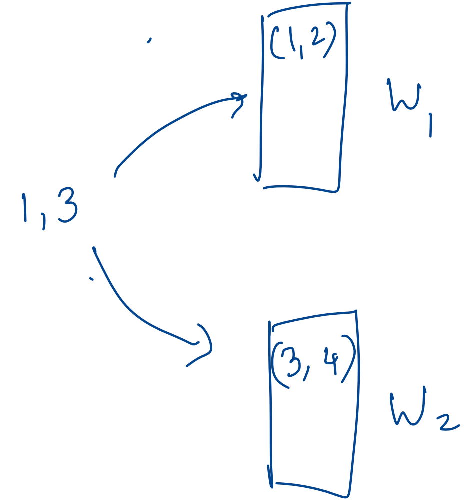
↳ given an edge, choose random machine

Coordinated Greedy Placement

↳ If either vertex is already placed favor those machines

Oblivious Greedy Placement

↳ Avoid coordination, only track vertices present locally



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

SUMMARY

Gather-Apply-Scatter programming model

Vertex cuts to handle power-law graphs

Balance computation, minimize communication

DISCUSSION

https://forms.gle/K7xk2KybTXf3XX3b6

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

① You can activate a subset of vertices across iterations ⇒ Work goes down as iters progress

② Partition in Spark → randomly partitions
smarter partitioning here? → lower communication

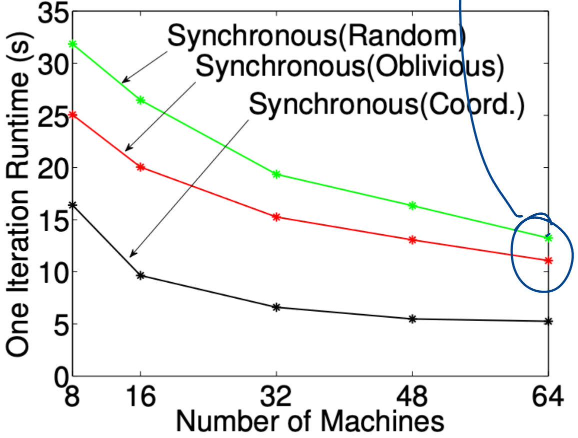
③ Delta caching → computation for active vertices can be lowered.

Partitiong time?

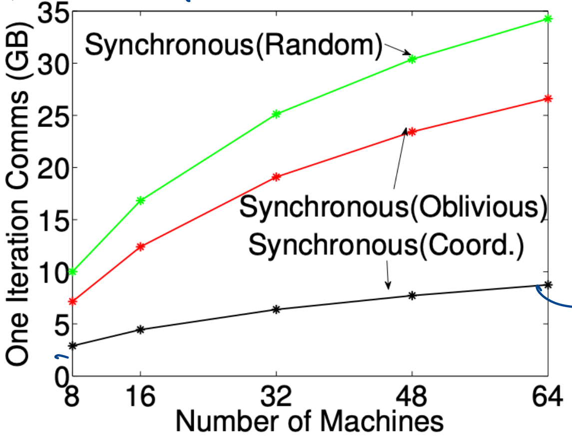
diminishing returns

get close to same

Comm goes up with num machine



(a) Twitter PageRank Runtime



(b) Twitter PageRank Comms

Sub-linear with more machines

NEXT STEPS

Next class: Marius