LARGE-SCALE GRAPH PROCESSING IN THE BIG DATA WORLD

Dr. Buğra Gedik, Ph.D.
MOTIVATION

- Graph data is everywhere
  - **Relationships** between people, systems, and the nature
  - **Interactions** between people, systems, and the nature

- Relationship graphs
  - Social media: Twitter follower-followee graph, Facebook friendship graph, etc.
  - The web: The link graph of web pages
  - Transportation networks, biological networks, etc.

- Interaction graphs
  - Social media: Mention graph of twitter
  - Telecommunications: Call Detail Records (CDRs)

- Interaction graphs can be summarized to form relationship graphs
APPLICATIONS

- Finding influence for ranking
  - Pages that are influential within the web graph (PageRank)
  - Users that are influential within the Twitter graph (TweetRank)
- Community detection for recommendations, churn prediction
  - If X is in the same community with Y, they may have similar interests
  - If X has churned, Y might be likely to churn as well
- Diffusion for targeted advertising
  - Start with known users in the graph with known interests, diffuse to others
- Regular path queries and graph matching for surveillance
BIG DATA

- The increase in the **Volume**, **Velocity**, and **Variety** of data has passed a threshold such that existing data management and mining technologies are insufficient in managing and extracting actionable insight from this data.

- Big Data technologies are new technologies that represent a paradigm shift, in the areas of platform, analytics, and applications.

- Key features
  
  *Scalability* in managing and mining of data
  
  Analysis and mining *with low-latency and high throughput*

  Non-traditional data, including semi-structured and unstructured
Graphs pose challenges in processing and management

RDBMS are inadequate for graph analytics
  - Traditional graph algorithms require traversals (e.g., BFS, DFS)
  - Traversals require recursive SQL: difficult to write, costly to execute

Large-scale graphs require distributed systems for scalability

Management vs. Processing
  - Management: CRUD operations (Create, Read, Update, Delete)
  - Processing: Graph analytics (BFS, Connected Components, Community Detection, Clustering Coefficient, PageRank, etc.)

Systems may support one or both
  - This talk focuses on graph processing systems, with a focus on distributed ones
DISTRIBUTED GRAPH PROCESSING SYSTEMS

- Graph data stays on the disk, typically in a distributed file system
  - E.g., graph data is on HDFS, in the form of list of edges
- To perform a graph analytic, the graph is loaded from the disk to the memory of a set of processing nodes
- The graph analytic is performed in-memory, using multiple nodes, typically requiring communication between them
- The graph could be potentially morphed during the processing
- The results (which could be a graph as well) are written back to disk
- Overall, it is a batch process
  - E.g., Compute the PageRank over the current snapshot of the web graph
- Advantages: Fast due to in-memory processing, scalable with increasing number of processing nodes
SOME APPROACHES

- **Apache Hadoop & Map/Reduce**
  - Use Map/Reduce framework for executing graph analytics

- **Vertex Programming**
  - A new model of processing specifically designed for graphs
  - Synchronous model
    - Foundational work: Pregel from Google
    - Pregel clones: Apache Giraph and HAMA (more general)
  - Asynchronous model
    - GraphLab, PowerGraph
    - Disk-based variety: GraphChi
HADOOP & MAP/REDUCE

- M/R is a scalable and fault-tolerant distributed data processing framework
- Why not use it for graph processing?
- Can we represent useful graph operations as M/R steps?
- How easy and efficient is it to implement graph analytics as a series of M/R steps?

EXAMPLE: DEGREE COMPUTATION

- **Out-degree computation**
  - **Source Data**: (from_vertex, to_vertex)
  - **Mapper**: (from_vertex, to_vertex) => *key*: from_vertex, *value*: 1
  - **Reducer**: key: vertex, values: [1, 1, …] => (vertex, vertex_degree)

- **Degree computation**
  - **Source Data**: (from_vertex, to_vertex)
  - **Mapper**: (from_vertex, to_vertex) => *key*: from_vertex, *value*: 1
  - *key*: to_vertex, *value*: 1
  - **Reducer**: key: vertex, values: [1, 1, …] => (vertex, vertex_degree)

- What if you want to augment each edge with the degrees of the vertices involved: (u, v) => (u, v, d(u), d(v))
  - We can add one job to add the d(u), another to add d(v)
  - Can we do this using less number of jobs?
EXAMPLE: DEGREE AUGMENTATION (1)
EXAMPLE: DEGREE AUGMENTATION (11)
EXAMPLE: PAGERANK

- Probability of a web surfer being at a particular page under the *random surfer model*

- Random surfer model:
  - The surfer starts from a random page
  - With probability $d$, she continues surfing by following one of the outlinks on the page at random
  - With probability $(1-d)$, she jumps to random page

- Let $p_i$ be the PageRank of page $i$, $N$ be the total number of pages, $M(i)$ be the pages that link to page $i$, and $L(i)$ be the out-degree of page $i
  
  $$p_i = (1-d) / N + d \times \sum_{j \in M(i)} p_j / L(j)$$

- Iterative implementation
  - Start with all pages having a PageRank of $1/N$
  - Apply the formula above to update it each page’s PageRank using page rank values from the last step
  - Repeat fixed number of times or until convergence
  - Note: pages with no outgoing links need special handling (assumed as if they link to all other pages)
PAGERANK M/R STYLE

one iteration

A | PR(A), [B, C]
B | PR(A)/2
C | PR(A)/2

Note: an additional step is needed to handle nodes with no outgoing links

A | PR(A), [B, C]
A | PR(D)

B | PR(B), [D]
B | PR(A)/2
B | PR(C)/2

C | PR(C), [B, D]
C | PR(A)/2

D | PR(D), [A]
D | PR(B)
D | PR(C)/2

B | (1-d)/N + d * (PR(A)/2 + PR(C)/2), [D]

ν | PR(ν), Neig(ν)
A | PR(A), [B, C]
B | PR(B), [D]
C | PR(C), [B, D]
D | PR(D), [A]
VERTEX PROGRAMMING (SYNC.)

- Graph analytics are written from the perspective of a vertex
  - You are programming a single vertex
  - The vertex program is executed for all of the vertices

- There are a few basic principles governing the execution
  - Each vertex maintains its own data
  - The execution proceeds in supersteps
  - At each superstep
    - the vertex program is executed for all vertices
  - Between two supersteps
    - Messages sent during the previous superstep are delivered
SUPERSTEPS

- During a superstep, the vertex program can do the following:
  - Access the list of messages sent to it during the last superstep
  - Update the state of the vertex
  - Send messages to other vertices (these will be delivered in the next superstep)
  - Vote to halt, if done
- Each vertex has access to vertex ids of its neighbors
- Vertex ids are used for addressing messages
- Messages can be sent to neighbor vertices or any other vertex (as long as the vertex id is learnt by some means, such as through messages exchanged earlier)
- The execution continues until no more supersteps can be performed, which happens when:
  - There are no pending messages
  - There are no unhalted vertices
BSP & PREGEL

- Vertex programs can be executed in a scalable manner using the Bulk Synchronous Processing (BSP) paradigm
- Pregel system by Google (research paper, code not available) does that
- Vertices are distributed to machines using some partitioning
  - The default is a hash based partitioning (on the vertex id)
- At each superstep, each machine executes the vertex program for the vertices it hosts (keeps the state for those vertices as well)
- At the end of the superstep, messages that need to cross machine boundaries are transported
- Pregel also supports additional abstractions
  - Aggregations: Reduction functions that can be applied on vertex values
  - Combiners: Reduction functions that can applied to messages destined to the same vertex from different vertices
  - Ability to remove vertices (morphing the graph)

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: Pregel: a system for large-scale graph processing. SIGMOD Conference 2010: 135-146
EXAMPLE: CONNECTED COMPONENTS

- Vertex state
  - Just an int value, representing the id of the connected component the vertex belongs
- The vertex program

```java
if (superstep() == 0)
    getVertexValue() = getVertexId();
    sendMessageToAllNeighbors(getVertexId());
} else {
    int mid = getVertexValue();
    for (msg in getReceivedMessages())
        mid = max(mid, msg.getValue());
    if (mid != getVertexValue()) {
        getVertexValue() = mid;
        sendMessageToAllNeighbors(mid);
    }
    VoteToHalt();
}
```
EXAMPLE: EXECUTION
EXAMPLE: PAGERANK

- Vertex state
  - Just a double value, representing the PageRank

- The vertex program

```java
if (superstep() >= 1) {
    double sum = 0;
    for (msg in getReceivedMessages())
        sum += msg.getValue();
    getVertexValue() = 0.15 / getNumVertices() + 0.85 * sum;
}
if (superstep() < 30) {
    int64 n = getNumOutEdges();
    sendMessageToOutNeighbors(getVertexValue() / n);
} else { VoteToHalt(); }
```

Surprisingly simple
SYSTEM ISSUES

- **Scalability**
  - Better than M/R for most graph analytics
  - Minimizing communication is key (many papers on improved partitioning to take advantage of high locality present in graph analytics)
  - Skew could be an issue for power graphs where there are a few number of very high degree vertices (results in imbalance)

- **Fault tolerance**
  - Checkpointing between supersteps, say every x supersteps, or every y seconds

- **Example Open Source Systems**
  - Apache Giraph: Pregel-like
  - Apache HAMA: More general BSP framework for data mining/machine learning
GraphLab targets not just graph processing, but also iterative Machine Learning and Data Mining algorithms

Similar to Pregel but with important differences
- It is asynchronous and supports dynamic execution schedules

Vertex programs in GraphLab
- Access vertex data, adjacent edge data, adjacent vertex data
  - These are called the scope
  - No messages as in Pregel, but similar in nature
- Update any of the things in scope as a result of execution
- Return a list of vertices, that will be scheduled for execution in the future
Asynchronous
- Pregel works in supersteps, with synchronization in-between
- Graphlab works asynchronously
  - It is shown that this improves convergence for many iterative data mining algorithms

Dynamic execution schedule
- Pregel executes each vertex at each superstep
- In GraphLab, new vertices to be executed are determined as a result of previous vertex executions

GraphLab also supports
- Configuring the *consistency model*: determines the extent to which computation can overlap
- Configuring the *scheduling*: determines the order in which vertices are scheduled (synchronous, round-robin, etc.)

GraphLab has multi-core (single machine) and distributed versions

EXAMPLE: PAGERANK

Input: Vertex data $\mathbf{R}(v)$ from $S_v$
Input: Edge data $\{w_{u,v}: u \in N[v]\}$ from $S_v$
Input: Neighbor vertex data $\{\mathbf{R}(u): u \in N[v]\}$ from $S_v$

$R_{old}(v) \leftarrow R(v)$  // Save old PageRank
$R(v) \leftarrow \alpha/n$

foreach $u \in N[v]$ do  // Loop over neighbors
[
  $R(v) \leftarrow R(v) + (1 - \alpha) \times w_{u,v} \times R(u)$
]

// If the PageRank changes sufficiently
if $|R(v) - R_{old}(v)| > \epsilon$ then
[ // Schedule neighbors to be updated
  return $\{u: u \in N[v]\}$
]

Output: Modified scope $S_v$ with new $R(v)$

The list of vertices to be added to the scheduler’s list
UNDERSTANDING DYNAMIC SCHEDULING

- A very high level view of the 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
  \( v \leftarrow \text{RemoveNext} (\mathcal{T}) \)
  \( (\mathcal{T}', S_v) \leftarrow f(v, S_v) \)
  \( \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}' \)
Output: Modified Data Graph \( G = (V, E, D') \)
```

- Consistency model adjusts how the execution is performed in parallel
- Scheduling adjusts how the RemoveNext method is implemented
GraphLab also supports
- Global aggregations over vertex values, which are read-only accessible to vertex programs
- Unlike Pregel, these are computed continuously in the background

PowerGraph is a GraphLab variant
- Specially designed for scale-free graphs
  - Degree distribution follows a power law
  - \( P(k) \sim k^{-y} \) \( k \): degree, \( y \): typically in the range 2 < \( y \) < 3
- Main idea is to decompose the vertex program into 3 steps
  - Gather: Collect data from the scope
  - Apply: Compute the value of the central vertex
  - Scatter: Update the data on adjacent vertices
- This way a single vertex with very high-degree can be distributed over multiple nodes (partition edges not vertices)

GraphChi is another GraphLab variant
- It is designed for disk-based, single machine processing
- The main idea is a disk layout technique that can be used to execute vertex programs by doing mostly sequential I/O (potentially parallel)

- Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012: 17-30
- Aapo Kyrola, Guy E. Blelloch, Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI 2012: 31-46
OTHER SYSTEMS

- **GraphX**
  - Built on Spark RDD
  - Supports Graph ETL tasks, such as graph creation and transformation
  - Supports interactive data analysis (kind of like PigLatin of the graph world)
  - Low-level, can be used to implement Pregel and GraphLab

- **Boost Parallel BGP**
  - SPMD approach with support for distributed data structures
  - Many graph algorithms are already implemented
  - No fault-tolerance

- Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica: GraphX: A resilient distributed graph system on Spark. GRADES 2013: 2
QUESTIONS

- ???