Recap

- Byzantine generals problem
  - They must decide on a common plan of action.
  - But, some of the generals can be traitors.
- Requirements
  - All loyal generals decide upon the same plan of action (e.g., attack or retreat).
  - A small number of traitors cannot cause the loyal generals to adopt a bad plan.
- Impossibility result
  - In general, with less than $3f + 1$ nodes, we cannot tolerate $f$ faulty nodes.

Today

- Distributed Graph Processing
- Google's Pregel system
  - Inspiration for many newer graph processing systems: Piccolo, Graph, GraphLab, PowerGraph, LFGraph, X-Stream, etc.

What Graphs?

- Large graphs are all around us
- Internet Graph: vertices are routers/switches and edges are links
- World Wide Web: vertices are webpages, and edges are URL links on a webpage pointing to another webpage
  - Called “Directed” graph as edges are uni-directional
- Social graphs: Facebook, Twitter, LinkedIn
- Biological graphs: DNA interaction graphs, ecosystem graphs, etc.

What Graph Analysis?

- Need to derive properties from these graphs
- Need to summarize these graphs into statistics
- E.g., find shortest paths between pairs of vertices
  - Internet (for routing)
  - LinkedIn (degrees of separation)
- E.g., do matching
  - Dating graphs in match.com (for better dates)
- PageRank
  - Web Graphs
  - Google search, Bing search, Yahoo search: all rely on this
- And many (many) other examples!

What Are the Difficulties?

- Large!
  - Human social network has 100s Millions of vertices and Billions of edges
  - WWW has Millions of vertices and edges
- Hard to store the entire graph on one server and process it
  - Slow on one server (even if beefy!)
- Need a distributed solution
Example

- Are C and D connected?
- Can we get all connected pairs?

Typical Graph Processing

- Works in iterations
- Each vertex assigned a value
- In each iteration, each vertex:
  - Gathers values from its immediate neighbors (vertices who join it directly with an edge). E.g., @A: B→A, C→A, D→A,…
  - Does some computation using its own value and its neighbors values.
  - Updates its new value and sends it out to its neighboring vertices. E.g., A→B, C, D, E
- Graph processing terminates after: i) fixed iterations, or ii) vertices stop changing values

One Possible Way

- Each vertex has two Boolean variables.
  - Cflag: true/false
  - Dflag: true/false
- Goal
  - Cflag: Am I connected to C?
  - Dflag: Am I connected to D?
  - If both are true at any of the vertices, C and D are connected.
  - Initially all false, except at C and D.
- Every iteration:
  - Propagates its values to neighboring vertices.
  - Collects the values from other vertices.
  - Updates the variables with OR, i.e., if any one of the values is true, it’s true.
- Iterate N times, where N is the length of the longest path in the entire graph.

Recall MapReduce?

Map phase
- Input (key, value): Computation: Output list of (intermediate key, intermediate value)
- Shuffle: Grouping based on intermediate keys.
- Reduce: Grouping based on intermediate keys.
  - Each intermediate key will get a list of intermediate values.
  - Update each of the Cflag & Dflag variables with OR, i.e., if any one of the values is true, it’s true.
- Run MapReduce N times, where N is the length of the longest path. Goal for each iteration:
  - Propagate each vertex’s Cflag & Dflag values to their neighboring vertices.
  - Collect Cflag & Dflag values from other vertices.
  - Update each of the Cflag & Dflag variables with OR, i.e., if any one of the values is true, it’s true.
- Map
  - Input (key, value): Computation: Output list of (intermediate key, intermediate value)
- Shuffle: Grouping based on intermediate keys.
  - Each intermediate key will get a list of intermediate values.
- Reduce
  - Input (intermediate key, list of intermediate values): Computation: Output (final key, final value)
MapReduce

• Run MapReduce N times
  • Map
    – Input (key, value): (vertex id, (current Cflag & Dflag values))
    – Computation: nothing.
    – Output list of (intermediate key, intermediate value): list of
      (neighboring vertex id, current Cflag & Dflag values)
      • For each neighboring vertex, propagate Cflag & Dflag values
  • Shuffle
    – Grouping based on vertex ids.
  • Reduce
    – Input (intermediate key, intermediate value): (vertex id, list of
      neighbors’ current Cflag & Dflag values)
    – Computation: OR
    – Output (key, value): (vertex id, updated Cflag & Dflag values)

Pros and Cons

• Pros
  – Well-known
  – The system is there.
  – It works.
• Cons
  – Not quite a good fit
  – Need to re-think in terms of keys, values, maps, and
    reduces.
• Question
  – Can we provide a system that is a better fit for graph
    processing?

Bulk Synchronous Parallel Model

• Originally by Valiant (1990)

Better Programming Model

• Vertex-centric programming
  • In each iteration, each vertex performs Gather-
    Apply-Scatter for all its assigned vertices
    – Gather: get all neighboring vertices’ values
    – Apply: compute own new value from own old value and
      gathered neighbors’ values
    – Scatter: send own new value to neighboring vertices

Google Pregel

• Gives simple APIs for easier graph processing
  • Vertex-centric programming
  • Developer’s code subclasses Vertex class
  • Implements Compute() method.
  • Vertex class allows a developer to:
    – Get/set vertex values
    – Get/set outgoing edge values (e.g., for weighted graphs)
    – Send/receive messages to any vertex
  • Gather-apply-scatter
    – Gather is done automatically (with scatter from the previous
      iteration)
    – Apply is done by Compute()
    – Scatter is done by explicit message passing

Page Rank Example

```
class PageRankVertex
{
  public Vertex<double, void, double> {
    public:
      virtual void Compute(MessageIterator* msg) {
        if (compute_step() != 1) {
          double sum = 0;
          for (msg->Dose(); msg->Next())
            sum += msg->Value();
            mutable_value() = 0.15 / NumVertices() + 0.85 * sum;
    }
    if (compute_step() < 30) {
      const int n = GetNumEdges();
      SendMessageToAllNeighborsGetValue() / n;
    } else {
      GetDefaultValue();
    }
  }
}
```
Google Pregel

- Pregel uses the master/worker model
  - Master (one server)
    » Maintains list of worker servers
    » Monitors workers; restarts them on failure
    » Provides Web-UI monitoring tool of job progress
  - Worker (rest of the servers)
    » Processes its vertices
    » Communicates with the other workers
- Naturally captures a graph structure (vertex per worker)
- Persistent data is stored as files on a distributed storage system (such as GFS or BigTable)
- Temporary data is stored on local disk

Summary

- Lots of (large) graphs around us
- Need to process these
- MapReduce not a good match
- Distributed Graph Processing systems: Pregel by Google

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).