Acknowledgements

- **Credits**
  
  - *Part of the course material is based on slides provided by the following authors*
    - Pietro Michiardi, Jimmy Lin
What’s a graph?

- \( G = (V,E) \), where
  - \( V \) represents the set of vertices (nodes)
  - \( E \) represents the set of edges (links)
  - Both vertices and edges may contain additional information

Different types of graphs:
- Directed vs. undirected edges
- Presence or absence of cycles

Graphs are everywhere:
- Hyperlink structure of the web
- Physical structure of computers on the Internet
- Interstate highway system
- Social networks

Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank
Graphs and MapReduce

- A large class of graph algorithms involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph

- Key questions:
  - How do you represent graph data?
    - Adjacency matrix
    - Adjacency list
  - How do you traverse a graph in MapReduce?

Adjacency Matrices

- Represent a graph as an $n \times n$ square matrix $M$
  - $n = |V|$
  - $M_{ij} = 1$ means a link from node $i$ to $j$

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks

- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space
Adjacency Lists

- Take adjacency matrices... and throw away all the zeros

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks

- Disadvantages:
  - Much more difficult to compute over inlinks

```
1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3
```

Agenda

- Graph algorithms in MapReduce
  - Parallel breath-first search
  - PageRank

- Graph Analysis Beyond Mapreduce
  - Pregel
Parallel Breath-First Search

Single-Source Shortest Path (SSSP)

- **Problem**
  - Find shortest path from a source node to all target nodes

- **Solution on a single machine**
  - Dijkstra algorithm using a global priority queue
    - Maintains a globally sorted list of nodes by current distance

```
1: \textbf{Dijkstra}(G, w, s)
2: \hspace{1em} d[s] \leftarrow 0
3: \hspace{1em} \textbf{for all} \hspace{0.5em} v \in V \hspace{1em} \textbf{do}
4: \hspace{2em} d[v] \leftarrow \infty
5: \hspace{1em} Q \leftarrow \{V\}
6: \hspace{1em} \textbf{while} \hspace{0.5em} Q \neq \emptyset \hspace{1em} \textbf{do}
7: \hspace{2em} u \leftarrow \textbf{ExtractMin}(Q)
8: \hspace{2em} \textbf{for all} \hspace{0.5em} v \in u.\textbf{AdjacencyList} \hspace{0.5em} \textbf{do}
9: \hspace{3em} \textbf{if} \hspace{0.5em} d[v] > d[u] + w(u, v) \hspace{0.5em} \textbf{then}
10: \hspace{4em} d[v] \leftarrow d[u] + w(u, v)
```
Example: SSSP - Dijkstra’s Algorithm

SSSP on large instances

- How to solve this problem in parallel?
  - “Brute-force” approach: breadth-first search (BFS)

- Parallel BFS: intuition
  - Flooding
  - Iterative algorithm in MapReduce
  - Try to mimic message passing style algorithms
Parallel Breadth-First Search: Pseudo code

```
1: class Mapper
2:    method Map(nid n, node N)
3:        d ← N.Distance
4:        Emit(nid n, N)  // Pass along graph structure
5:    for all nodeid m ∈ N.AdjacencyList do
6:        Emit(nid m, d + 1)  // Emit distances to reachable nodes
1: class Reducer
2:    method Reduce(nid m, [d1, d2, ...])
3:        d_min ← ∞
4:        M ← Ø
5:    for all d ∈ counts [d1, d2, ...] do
6:        if IsNode(d) then
7:            M ← d
8:        else if d < d_min then
9:            d_min ← d
10:    M.Distance ← d_min
11:    Emit(nid m, node M)
```

Parallel Breadth-First Search

- **Assumptions**
  - Connected, directed graph
  - Data structure: adjacency list
  - Distance to each node is stored alongside the adjacency list of that node

- **The pseudo-code**
  - We use n to denote the node id (an integer)
  - We use N to denote the node adjacency list and current distance
  - The algorithm works by mapping over all nodes
  - Mappers emit a key-value pair for each neighbor on the node’s adjacency list
    - The key: node id of the neighbor
    - The value: the current distance to the node plus one
    - If we can reach node n with a distance d, then we must be able to reach all the nodes connected to n with distance d + 1
Parallel Breadth-First Search

- The pseudo-code (continued)
  - After shuffle and sort, reducers receive keys corresponding to the destination node ids and distances corresponding to all paths leading to that node
  - The reducer selects the shortest of these distances and update the distance in the node data structure

- Passing the graph along
  - The mapper: emits the node adjacency list, with the node id as the key
  - The reducer: must distinguish between the node data structure and the distance values

- MapReduce iterations
  - The first time we run the algorithm, we “discover” all nodes connected to the source
  - The second iteration, we discover all nodes connected to those
  - Each iteration expands the “search frontier” by one hop
  - How many iterations before convergence?

- This approach is suitable for small-world graphs
  - The diameter of the network is small
Parallel Breadth-First Search

- Checking the termination of the algorithm
  - Requires a “driver” program which submits a job, check termination condition and eventually iterates
  - In practice:
    - Hadoop counters
    - Side-data to be passed to the job configuration

- Extensions
  - Storing the actual shortest-path
  - Weighted edges (as opposed to unit distance)

Summary

- The graph structure is stored in an adjacency lists
  - This data structure can be augmented with additional information

- The MapReduce framework
  - Maps over the node data structures involving only the node’s internal state and it’s local graph structure
  - Map results are “passed” along outgoing edges
  - The graph itself is passed from the mapper to the reducer
    - This is a very costly operation for large graphs!
  - Reducers aggregate over “same destination” nodes

- Graph algorithms are generally iterative
  - Require a driver program to check for termination
Graph algorithm: PageRank

- What is PageRank
  - It’s a measure of the relevance of a Web page, based on the structure of the hyperlink graph
  - Based on the concept of random Web surfer

- Formally we have:
  \[ P(n) = \alpha \left( \frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)} \]
  - \(|G|\) is the number of nodes in the graph
  - \(\alpha\) is a random jump factor
  - \(L(n)\) is the set of out-going links from page \(n\)
  - \(C(m)\) is the out-degree of node \(m\)
PageRank in Details

- PageRank is defined recursively, hence we need an iterative algorithm
  - A node receives “contributions” from all pages that link to it

- Consider the set of nodes \( L(n) \)
  - A random surfer at \( m \) arrives at \( n \) with probability \( 1/C(m) \)
  - Since the PageRank value of \( m \) is the probability that the random surfer is at \( m \), the probability of arriving at \( n \) from \( m \) is \( P(m)/C(m) \)

- To compute the PageRank of \( n \) we need:
  - Sum the contributions from all pages that link to \( n \)
  - Take into account the random jump, which is uniform over all nodes in the graph

PageRank: Example

[Graphs showing iteration 1 and iteration 2 with values for each node indicated]
PageRank: pseudo-code

1: class MAPPER
2:    method MAP(nid n, node N)
3:        p ← N.PAGERANK/[N.ADJACENCYLIST]  ▷ Pass along graph structure
4:        Emit(nid n, N)  
5:    for all nodeid m ∈ N.ADJACENCYLIST do
6:        Emit(nid m, p)  ▷ Pass PageRank mass to neighbors
1: class REDUCER
2:    method REDUCE(nid m, [p₁, p₂, ...])
3:        M ← ∅
4:        for all p ∈ counts [p₁, p₂, ...] do
5:            if ISNODE(p) then
6:                M ← p  ▷ Recover graph structure
7:            else
8:                s ← s + p
9:        M.PAGERANK ← s  ▷ Sum incoming PageRank contributions
10:        Emit(nid m, node M)

PageRank: Example
PageRank in MapReduce

- **Sketch of the MapReduce algorithm**
  - The algorithm maps over the nodes
  - For each node computes the PageRank mass the needs to be distributed to neighbors
  - Each fraction of the PageRank mass is emitted as the value, keyed by the node ids of the neighbors
  - In the shuffle and sort, values are grouped by node id
    - Also, we pass the graph structure from mappers to reducers (for subsequent iterations to take place over the updated graph)
  - The reducer updates the value of the PageRank of every single node

- **Implementation details**
  - Loss of PageRank mass for sink nodes
  - Auxiliary state information
  - One iteration of the algorithm
    - Two MapReduce jobs: one to distribute the PageRank mass, the other for dangling nodes and random jumps
  - Checking for convergence
    - Requires a driver program
    - When updates of PageRank are “stable” the algorithm stops
Graph Analysis Beyond MapReduce

Acknowledgements

- Credits
  - Part of the course material is based on slides provided by the following authors
    - Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski (Google, Inc.)
Pregel: A System for Large-Scale Graph Processing

- What is it?
  - Model for fault-tolerant parallel processing of graphs
  - C++ API allowing users to apply this model

- Why use it?
  - Problems solvable with graph algorithms are common
  - The alternatives aren’t very good
    - Develop distributed architecture for individual algorithms
    - Existing distributed platform (e.g., MapReduce)
      - May not be very good at graph algorithms (multiple stages → lots of overhead)

The Pregel model (1/2)

- Master/Worker model
  - Each worker assigned a subset of a directed graph’s vertices

- Vertex-centric model. Each vertex has:
  - An arbitrary “value” that can be get/set.
  - List of messages sent to it
  - List of outgoing edges (edges have a value too)
  - A binary state (active/inactive)
The Pregel model (2/2)

- **Bulk Synchronous Parallel model**
  - Synchronous iterations of asynchronous computation
  - Master initiates each iteration (called a “superstep”)
  - At every superstep
    - Workers asynchronously execute a user function on all of its vertices
    - Vertices can receive messages sent to it in the last superstep
    - Vertices can send messages to other vertices to be received in the next superstep
    - Vertices can modify their value, modify values of edges, change the topology of the graph (add/remove vertices or edges)
    - Vertices can “vote to halt”
  - Execution stops when all vertices have voted to halt and no vertices have messages.
  - Vote to halt trumped by non-empty message queue

Illustration: vertex partitions

http://stochastix.files.wordpress.com/
Loading the graph input

- Master assigns section of input to each worker
- Vertex “ownership” determined by hash(v) mod N
  - N - number of partitions
  - Recall each worker is assigned one or more partitions
  - User can modify this to exploit data locality
- Worker reads its section of input:
  - Stores vertices belonging to it
  - Sends other vertices to the appropriate worker
- Input stored on something like GFS
  - Section assignments determined by data locality

Simple example: find max

```
i_val := val
for each message m
  if m > val then val := m
  if i_val == val then
    vote_to_halt
  else
    for each neighbor v
      send_message(v, val)
```

```
3 6 2 1
6 6 2 6
6 6 6 6
6 6 6 6
```
Combiners

- Sometimes vertices only care about a summary value for the messages it is sent (e.g., previous example)
- Combiners allow for this (examples: min, max, sum, avg)
- Messages combined locally and remotely
- Reduces bandwidth overhead
- User-defined, not enabled by default
Fault Tolerance (1/2)

- At start of superstep, master tells workers to save their state:
  - Vertex values, edge values, incoming messages
  - Saved to persistent storage
- Master saves aggregator values (if any)
- This isn’t necessarily done at every superstep
  - That could be very costly
  - Authors determine checkpoint frequency using mean time to failure model

Fault Tolerance (2/2)

- When master detects one or more worker failures:
  - All workers revert to last checkpoint
  - Continue from there
  - That’s a lot of repeated work!
  - At least it’s better than redoing the whole thing.
Example: PageRank

class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *mutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = getOutEdgeIterator().size();
            sendMessageToAllNeighbors(getValue() / n);
        } else {
            voteToHalt();
        }
    }
};

\[ PR(p; t + 1) = \frac{1 - d}{N} + d \sum_{p | M(p)} \frac{PR(p; t)}{L(p)} \]

http://wikipedia.org

Alternatives to Pregel

- Pregel is a Google project
- Alternative open source project similar in spirit
  - GPS: A Graph Processing System
    - Developed at Stanford Univ.
  - Giraph
    - An Apache project
- Other alternatives, tailored to the specific problem
  - E.g. “Filtering: A Method for Solving Graph Problems in MapReduce”