Graph-Processing Systems
(focusing on GraphChi)
Recall: PageRank in MapReduce (Hadoop)

Input:
- adjacency matrix $H$

Map Phase:
- HDFS
- (a, [c])
- (b, [a])
- (c, [a, b])

Shuffle Phase:
- $(c, PR(a) / \text{out}(a)), (a, [c])$
- $(a, PR(b) / \text{out}(b)), (b, [a])$
- $(a, PR(c) / \text{out}(c)), (c, [a, b])$
- $(b, PR(c) / \text{out}(c))$

Reduce Phase:
- $\text{PR}(a) = 1 - l/N + l \sum \text{PR}(y)/\text{out}(y)$
- $(a, \text{PR}(a)/\text{out}(a))$

Write to local storage
Write to HDFS
Iterate
Traditional frameworks are poorly suited for graphs

- From a programming point of view:
  - Do not map neatly to the “flat” map/reduce paradigm

- From a performance point of view
  - Graphs have poor locality of memory access
  - Usually do very little work per vertex
  - Have changing degree of parallelism over course of execution
  - Do very little (often localised work) over and over again.
This presentation

● Focus on GraphChi but:
  ○ Highlight the “tension” between edge-centric vs vertex centric programming
  ○ Highlight the challenges of non-distributed vs distributed approaches
Pregel (2010): “Think like a vertex”

- Define computation as a sequence of message exchanges amongst vertices.
- Impose a structure on program execution (Bulk Synchronous Parallelism):
  - Split execution into supersteps: at each step, every vertex receives messages sent in previous superstep (can only receive messages from adjacent nodes).
  - Within each step, vertices compute in parallel each executing the same user-defined function.
- Vertices can store state (unlike MapReduce). Not edges.
- Make the vertex the unit of partitioning of computation for different machines:
  - Vertices compute in parallel each executing the same user-defined function.
Pregel (2010): “Think like a vertex”

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();
        }
    }
};
But ... real graphs follow a power-law distribution

Source: PowerGraph (OSDI’12)
And power law graphs introduce challenges

- **Work balance**
  - Work imbalance for highly connected vertices as storage/communication linear in the degree of the node

- **Partitioning**
  - Natural graphs difficult to partition to minimise communication and maximise work balance
  - Random hashing works badly

- **Communication/Storage:**
  - Communication asymmetry + high amount of storage required to store the adjacency matrix

- **No parallelism possible within individual vertices**
PowerGraph (2012): it’s all about edges, not vertices

- Introduce GAS programming (Think Like a Vertex)

  ```java
  interface GASVertexProgram(u) {
    // Run on gather_nbrs(u)
    gather(D_u, D_(u,v), D_v) → Accum
    sum(Accum left, Accum right) → Accum
    apply(D_u, Accum) → D_u^new
    // Run on scatter_nbrs(u)
    scatter(D_u^new, D_(u,v), D_v) → (D_u^new, Accum)
  }
  ```

- BUT eliminate degree dependence of vertex-program by decomposing GAS to factor vertex-programs over edges
  - Program in a vertex centric way, but implement edge-centric code
  - (I find this super-cool)
PageRank in GAS

```java
PageRank

// gather_nbrs: IN_NBRS
gather(D_u, D_{u,v}, D_v):
    return D_v.rank / \#outNbrs(v)

sum(a, b): return a + b

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

// scatter_nbrs: OUT_NBRS
scatter(D_u, D_{u,v}, D_v):
    if (|D_u.delta| > \epsilon) Activate(v)
    return delta
```
GraphChi (2012): All you need is a Macbook Mini

- Partitioning a graph is hard (especially for power law graphs).
  - Would it be possible to instead do advanced graph partitioning on a single computer?

- Goal of GraphChi is to maximize sequential access when loading graph into memory (500x speedup for sequential vs random)

- Execute on individual subgraphs, loading them efficiently from disk
  - Introduce the concept of “parallel sliding window” (PSW) to achieve this
GraphChi (2012): Programming Model

Like Pregel, vertex-centric computation model

Algorithm 1: Typical vertex update-function

1. Update(vertex) begin
2.   `x[]` ← read values of in- and out-edges of vertex;
3.   vertex.value ← f(x[]);
4.   foreach edge of vertex do
5.     edge.value ← g(vertex.value, edge.value);
6. end
7. end


1. typedef: VertexType float
2. Update(vertex) begin
3.     var sum ← 0
4.     for e in vertex.inEdges() do
5.       sum += e.weight * neighborRank(e)
6. end
7. vertex.setValue(0.15 + 0.85 * sum)
8. broadcast(vertex)
9. end
GraphChi (2012): Parallel Sliding Window

- Three Steps:
  - Loading a subgraph from disk (by using shards + execution intervals)
  - Updating the vertices and edges
  - Writing the updated values to disk

- Pre-processing of graph necessary when loaded for the first time (to determine shards/execution internals)
  - Compute in-degree of each vertex (full pass over data) + partition vertex accordingly into shards using prefix sum, explicitly writing out the vertices to file + a file with their in/out degree. Requires 3 full (sequential) pass over data
GraphChi (2012): Loading a subgraph

- Partition vertices into **shards** (must fit in memory)
- Each shard contains edges with destination in that shard.
- Edges are sorted by source address.
- Execution internal - process one vertex at a time
  - Load corresponding shard into memory, then iterate over all other shards to read out-edges (will be sequential as sorted by source address)
GraphChi (2012): Parallel Updates & Scheduling

- Executes user-defined update function for each vertex in parallel
- Enforce external determinism by executing vertices with endpoints in the same interval in sequential order
- Selective scheduling: focus computation to where most needed by flagging vertices to be updated with higher priority
GraphChi (2012): Results

<table>
<thead>
<tr>
<th>Application &amp; Graph</th>
<th>Iter.</th>
<th>Comparative result</th>
<th>GraphChi (Mac Mini)</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagerank &amp; domain</td>
<td>3</td>
<td>GraphLab [51] on AMD server (8 CPUs) 87 s</td>
<td>132 s</td>
<td>-</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>5</td>
<td>Spark [48] with 50 nodes (100 CPUs): 486.6 s</td>
<td>790 s</td>
<td>[42]</td>
</tr>
<tr>
<td>Pagerank &amp; V=105M, E=3.7B</td>
<td>100</td>
<td>Stanford GPS, 30 EC2 nodes (60 virt. cores), 144 min</td>
<td>approx. 581 min</td>
<td>[41]</td>
</tr>
<tr>
<td>Pagerank &amp; V=1.0B, E=18.5B</td>
<td>1</td>
<td>Piccolo, 100 EC2 instances (200 cores) 70 s</td>
<td>approx. 26 min</td>
<td>[40]</td>
</tr>
<tr>
<td>Webgraph-BP &amp; yahoo-web</td>
<td>1</td>
<td>Pegasus (Hadoop) on 100 machines: 22 min</td>
<td>27 min</td>
<td>[24]</td>
</tr>
<tr>
<td>ALS &amp; netflix-mm, D=20</td>
<td>10</td>
<td>GraphLab on AMD server: 4.7 min</td>
<td>9.8 min (in-mem)</td>
<td>[31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40 min (edge-repl.)</td>
<td></td>
</tr>
<tr>
<td>Triangle-count &amp; twitter-2010</td>
<td>-</td>
<td>Hadoop, 1636 nodes: 423 min</td>
<td>60 min</td>
<td>[43]</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>1</td>
<td>PowerGraph, 64 x 8 cores: 3.6 s</td>
<td>158 s</td>
<td>[21]</td>
</tr>
<tr>
<td>Triange-count &amp; twitter-2010</td>
<td>-</td>
<td>PowerGraph, 64 x 8 cores: 1.5 min</td>
<td>60 min</td>
<td>[21]</td>
</tr>
</tbody>
</table>

Great results but extremely high pre-processing cost!

Table 1: Experiment graphs. Preprocessing (conversion to shards) was done on Mac Mini.
GraphChi (2012): Drawbacks

- Extremely high cost of pre-processing phase (though graph can be modified incrementally once loaded)
- Vertex-centric model makes it necessary to re-sort the edges in the shard by destination vertex after loading the shard into memory (claim by X-Stream)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Pre-Sort(s)</th>
<th>Runtime(s)</th>
<th>Re-sort(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Twitter pagerank</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Stream (1)</td>
<td>none</td>
<td>397.57 ± 1.83</td>
<td>-</td>
</tr>
<tr>
<td>GraphChi (32)</td>
<td>752.32 ± 9.07</td>
<td>1175.12 ± 25.62</td>
<td>969.99</td>
</tr>
<tr>
<td><strong>Netflx ALS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Stream (1)</td>
<td>none</td>
<td>76.74 ± 0.16</td>
<td>-</td>
</tr>
<tr>
<td>GraphChi (14)</td>
<td>123.73 ± 4.06</td>
<td>138.68 ± 26.13</td>
<td>45.02</td>
</tr>
<tr>
<td><strong>RMAT27 WCC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Stream (1)</td>
<td>none</td>
<td>867.59 ± 2.35</td>
<td>-</td>
</tr>
<tr>
<td>GraphChi (24)</td>
<td>2149.38 ± 41.35</td>
<td>2823.99 ± 704.99</td>
<td>1727.01</td>
</tr>
<tr>
<td><strong>Twitter belief prop.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Stream (1)</td>
<td>none</td>
<td>2665.64 ± 6.90</td>
<td>-</td>
</tr>
<tr>
<td>GraphChi (17)</td>
<td>742.42 ± 13.50</td>
<td>4589.52 ± 322.28</td>
<td>1717.50</td>
</tr>
</tbody>
</table>

- Performance imbalance creeps back if have to create mini-partitions of highly connected nodes => disk bottleneck?
X-Stream (2013): edge-centric GAS for Macbooks

- Use edge-centric GAS to obtain fully sequential access to edges (at the cost of random access to vertices)
  - Assume that number of edges is larger than number of vertex

- Use streaming partitions to load edges and determine, based on destination whether an update needs to be propagated to active vertex.

- Prefer to stream (potentially many) unrelated edges over the cost of edge-random access in GraphChi + cost of creating an index
X-Stream (2013): edge-centric GAS for Macbooks

vertex_scatter(vertex v)
   send updates over outgoing edges of v

vertex_gather(vertex v)
   apply updates from inbound edges of v

while not done
   for all vertices v that need to scatter updates
      vertex_scatter(v)
   for all vertices v that have updates
      vertex_gather(v)

edge_scatter(edge e)
   send update over e

update_gather(update u)
   apply update u to u.destination

while not done
   for all edges e
      edge_scatter(e)
   for all updates u
      update_gather(u)

Figure 1: Vertex-centric Scatter-Gather

Figure 2: Edge-centric Scatter-Gather
Unifying graph processing with general processing (2013 and beyond)

- Naiad (SOSP’13): uses timely dataflow (+ inherent asynchrony, like Pregel) with optional SQL-like GraphLinq

- GraphX (OSDI’14): layer over Spark for graph processing. Recasts graph-specific optimizations as distributed join optimizations and materialized view maintenance

- Musketeer (Eurosys’15): GAS can be expressed in relational algebra by a JOIN (scatter phase) and GROUP-BY (apply phase) placed within a WHILE loop
Going forward

● Need to be careful about when distribution makes sense
  ○ Partitioning is still a hard (unsolved?) problem

● Need to make sure that parallelism does not actually hurt performance
  ○ “Think like a vertex” forces programmer to use label propagation for graph connectivity when union find performs better.

● Do we need special abstractions for graphs, or is something like timely dataflow enough?
  ○ Could timely dataflow ever beat PowerGraph?

● What is the best way to represent a graph on disk?