PowerGraph

Presented by: Omar Obeya
New Problem

Problem
Many natural graphs are power law graphs. This means degree unbalance can impede distributed computation.

Challenges
1 – Work Balance
2 – Communication
3 – Partitioning
4 – Storage
5 – Computation
Solution: PowerGraph

Essence of the solution:

1 – Decouple different types of operations (read-only, write to adjacent nodes, changing node data).

2 – Use smart partitioning strategies to decrease communication.

3 – Shared memory; data not need to be moved.

4 – Analysis with respect to power law graph.
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,v}^{new}, Accum)
}
Delta Caching

– Avoids re-gather-ing of data of unchanged neighbors.

- Optional

- Not always possible

– Useful for power law graphs.

```c
// 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|>\varepsilon) Activate(v)
  return delta
```
Interface Comparison

```c
// gather_nbgs: IN_NBRS

```gather
```c
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_nbgs);
```gather
```c
// scatter_nbgs: OUT_NBRS

```scatter
```c
void GraphLabPageRank(Scope scope):
    float accum = 0;
    foreach (nbr in scope.in_nbgs):
        accum += nbr.val / nbr.nout_nbrs();
    vertex.val = 0.15 + 0.85 * accum;
```scatter
```c
Message combiner(Message m1, Message m2):
    return Message(m1.value() + m2.value());
```

```c
// gather_nbgs: 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_nbgs: OUT_NBRS

scatter(D_u, D_(u,v), D_v):
    if(|D_u.delta|>ε) Activate(v)
    return delta
```
Partition Design Choices

1 – Put each edge on one machine.

2 – Put replicas of vertices on different machines.

3 – Elect one replica as master and others as mirrors, maintain consistency in a centralized fashion.

4 – Minimize replicas to minimize communication and duplication of data.
Vertex Cut vs. Edge Cut

Figure 4: (a) An edge-cut and (b) vertex-cut of a graph into three parts. Shaded vertices are ghosts and mirrors respectively.
## Vertex Cut vs. Edge Cut

<table>
<thead>
<tr>
<th>Vertex Cut</th>
<th>Edge Cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Minimizes vertex cuts - replicas in powerGraph</td>
<td>1 – Hard to compute with power law graphs.</td>
</tr>
<tr>
<td>2 – Efficient to compute</td>
<td>2 – Even if computed, not suitable for PowerGraph.</td>
</tr>
<tr>
<td></td>
<td>3 – When random, most edges will be cut.</td>
</tr>
</tbody>
</table>
Communication

Figure 5: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulator is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.
Vertex Cut Computation

**Random**

Assign each edge to a different machine in parallel.

**Greedy**

Case 1: If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

Case 2: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.
Implementations

1 – Synchronized

2 – Asynchronized

3 – Asynchronized and Serializable
Parallel Locking

**Async Serializable**
*PowerGraph*

1 – Use Parallel Locking
2 – Extend Chandy-Misra Solution
3 – Each mirror attempts to acquire its own locks.

**GraphLab**

1 – Sequential Locking
2 – Use Dijkstra
3 – Suitable only when nodes degrees are small.
Comparison with Pregel and GraphLab

(a) Power-law Fan-In Runtime  
(b) Power-law Fan-Out Runtime  
(c) Power-law Fan-In Comm.  
(d) Power-law Fan-Out Comm.
1. An analysis of the challenges of power-law graphs in distributed graph computation and the limitations of existing graph parallel abstractions (Sec. 2 and 3).
2. The PowerGraph abstraction (Sec. 4) which factors individual vertex-programs.
3. A delta caching procedure which allows computation state to be dynamically maintained (Sec. 4.2).
4. A new fast approach to data layout for power-law graphs in distributed environments (Sec. 5).
5. An theoretical characterization of network and storage (Theorem 5.2, Theorem 5.3).
6. A high-performance open-source implementation of the PowerGraph abstraction (Sec. 7).
7. A comprehensive evaluation of three implementations of PowerGraph on a large EC2 deployment using real-world MLDM applications (Sec. 6 and 7).

1 – Achieved the five goals, with minimal trade-offs.
2 – Thorough analysis
3 – The research is built on assuming natural graphs are power laws.


Questions

- The paper makes use of power law, what about other properties in natural graphs?
- How does the nature of the algorithm impacts the framework?
- How does PowerGraph compares with GraphLab and Pregel?