PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

Authors: Joseph Gonzalez, Yucheng Low, et al

Presenter: Ian Limarta

Outline

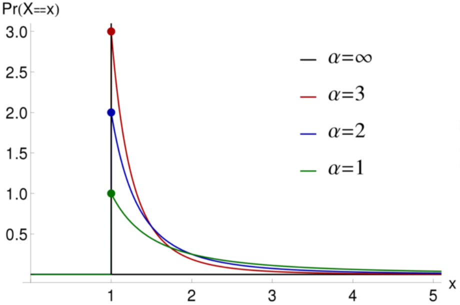
Setup: Why PowerGraph?

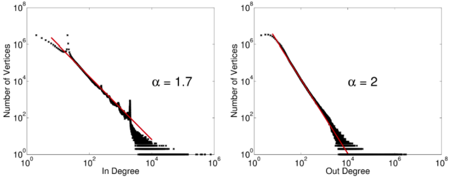
- Many graph frameworks deal with distributed data over large graphs (e.g. Pregel, GraphLib, etc).
- Thematic idea: Process over vertices to compute local neighborhood data and then pass data to other vertices in future steps (possibly across the network).
- Often fails to scale well for vertices with large degrees. These frameworks scale as the degree increases.
- The *lack* of frameworks which incorporated natural *power law* characteristics seen in many real life graphs motivated the development of **PowerGraph**.

Setup: Influence of Power Laws

Power Law

$\mathbf{P}[d] \propto d^{-\alpha}$ where α is an exponent which controls skewness.





(a) Twitter In-Degree (b) Twitter Out-Degree

- A higher α means more skewness or fewer outlier vertices.
- Typically $\alpha \approx 2$. $\alpha_{twitter} \approx 1.8$

Aside: Graph Parallel Abstractions

- Let $G(V, E)$ be a sparse and $N(v)$ be the adjacent vertices of v
- Each vertex $v \in V$ has the same executing *vertex program*, Q. Each $Q(v)$ may execute in parallel with other vertices' programs.
- Each $v \in V$ has associated data D_v
- Each edge $e \in E$ has data $D_{(u,v)}$

Aside: Graph Parallel Abstractions

- Let $G(V, E)$ be a sparse and $N(v)$ be the adjacent vertices of v
- Each vertex $v \in V$ has the same executing *vertex program*, Q. Each $Q(v)$ may execute in parallel with other vertices' programs.
- Each $v \in V$ has associated data D_v
- Each edge $e \in E$ has data $D_{(u,v)}$

Gather, Apply, Scatter (GAS) Abstraction

Each vertex v undergoes the following steps:

1. Gather phase: Collect information of $N(v)$ and combines information into an aggregate statistic.
2. Apply phase: Apply the aggregate statistic to v
3. Scatter phase: Forward the changes in v to the adjacent edges.

Aside: Graph Parallel Abstractions

Gather, Apply, Scatter (GAS) Abstraction

Each vertex v undergoes the following steps:

1. Gather phase: Collect information of $N(v)$ and combines information into an aggregate statistic.
2. Apply phase: Apply the aggregate statistic to v
3. Scatter phase: Forward the changes in v to the adjacent edges.

```
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)
}
```

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if *cached accumulator* a_u *is empty* **then**
  foreach *neighbor v in gather_nbrs(u)* **do**
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
  end
end
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
  if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  else $a_v \leftarrow$ Empty
end

Application: Graph Coloring

- Given a graph G, color using $c = 1, 2, \ldots$ such that no two vertices share the same color.

Application: Graph Coloring

- Given a graph G, color using $c = 1, 2, \ldots$ such that no two vertices share the same color.

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Figure 1: Greedy Graph Coloring

Application: Graph Coloring

- Given a graph G, color using $c = 1, 2, \ldots$ such that no two vertices share the same color.

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Figure 2: Greedy Graph Coloring

- What is `Activate()`?

GAS's Termination Rules

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Figure 3: Greedy Graph Coloring

- A vertex remains activate until its vertex program terminates. Becomes inactive.
- Any vertex, including itself, can call `Activate(v)` to start a new execution of GAS.
- The graph procedure ends when all vertices are inactive.

PowerGraph Abstraction

- PowerGraph implements the GAS abstraction and rules via `gather`, `sum`, `apply`, and `scatter`.
- Can formulate Pregel and other libraries in terms of PowerGraph abstraction
- Unlike Pregel and PowerGraph, provides a caching method called **delta caching**

What is new then?

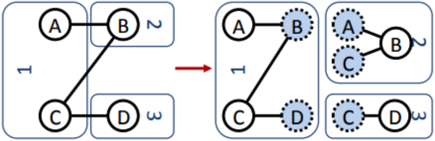
- Often large graphs require many machines.
- Occasionally some vertices require *multiple* machines!

What is new then?

- Often large graphs require many machines.
- Occasionally some vertices require *multiple* machines!
- In Pregel, we must have a copy of the vertex in each machine.

What is new then?

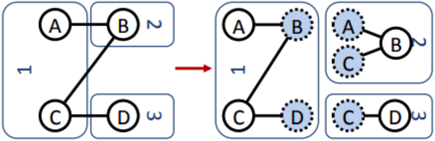
- Often large graphs require many machines.
- Occasionally some vertices require *multiple* machines!
- In Pregel, we must have a copy of the vertex in each machine.



(a) Edge-Cut

What is new then?

- Often large graphs require many machines.
- Occasionally some vertices require *multiple* machines!
- In Pregel, we must have a copy of the vertex in each machine.



(a) Edge-Cut

- Can introduce "ghost" edges.

PowerGraph Solution

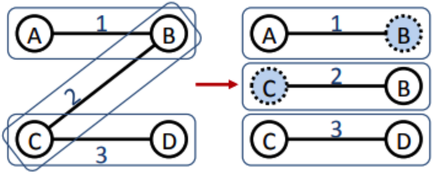
- Use **vertex** cuts instead.

PowerGraph Solution

- Use **vertex** cuts instead.
- Machines store edges. Each edge lies in exactly one machine.
- Vertices may have replicas across machines.

PowerGraph Solution

- Use **vertex** cuts instead.
- Machines store edges. Each edge lies in exactly one machine.
- Vertices may have replicas across machines.



(b) Vertex-Cut

Handling Replicas

Some crucial details:

- Since there are many replicas of vertices, PowerGraph employs the master-follower paradigm to commit changes into vertices.

Handling Replicas

Since there are many replicas of vertices, PowerGraph employs the master-follower paradigm to commit changes into vertices.

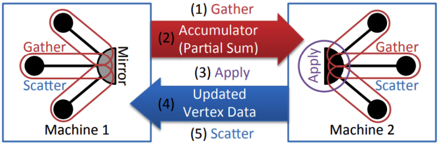


Figure 4: Replicas of a vertex

- Master is randomly selected per set of replicas.
- All changes directed to master.
- Followers are read-only for everyone except the master

Balanced p-way Vertex Cuts

- Since vertices are replicated across machines, can't we run into similar memory and network problems as Pregel?

Balanced p-way Vertex Cuts

- Since vertices are replicated across machines, can't we run into similar memory and network problems as Pregel?
- If we used *edge* cuts, yes. Not so much for *vertex*.
- How do we distribute edges across machines?
- Randomly hash edges to machines $i \in \{1, 2, \ldots p\}$?

Balanced p-way Vertex Cuts

- Since vertices are replicated across machines, can't we run into similar memory and network problems as Pregel?
- If we used *edge* cuts, yes. Not so much for *vertex*.
- How do we distribute edges across machines?
- Randomly hash edges to machines $i \in \{1, 2, \dots p\}$?
- Yes!

Balanced p-way Vertex Cuts

- Let $A(v)$ be the machines that have replicas of vertex v where $A(V) \subset \{1, 2, \ldots, p\}$
- Let $A(e)$ be the machine containing edge $e \in E$.

Analysis

- Let $A(v)$ be the machines that have replicas of vertex v where $A(V) \subset \{1, 2, \ldots, p\}$
- Let $A(e)$ be the machine containing edge $e \in E$.

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)|$$

s.t

$$\max_{m} |\{e \in E \mid A(e) = m\}| < \lambda \frac{|E|}{p}$$

Analysis

- Let $A(v)$ be the machines that have replicas of vertex v where $A(V) \subset \{1, 2, \ldots, p\}$
- Let $A(e)$ be the machine containing edge $e \in E$.

$$\min_A \frac{1}{|V|} \sum_{v \in V} |A(v)|$$

s.t

$$\max_m |\{e \in E \mid A(e) = m\}| < \lambda \frac{|E|}{p}$$

- We say we have a *balanced p-way vertex cut* for the edge assignments corresponding to the solution to this optimization problem.
- Somewhat difficult to solve. Can instead randomly hash edges to machines

Analysis of Edge Hashing

Theorem 1: Randomized Vertex Cuts

A random vertex cut on p machines has expected replication

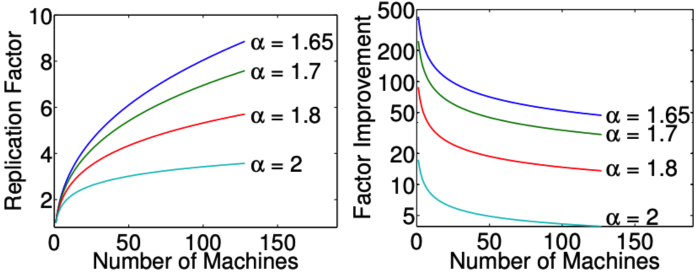
$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v \in V}|A(v)|\right] = \frac{p}{|V|}\sum_{v \in V}\left(1 - \left(1 - \frac{1}{p}\right)^{D(v)}\right)$$

Greedy Edge Hashing

- It turns out that we can derandomize our randomized vertex algorithm
- Guarantees at least as good replica score as the randomized algorithm
- Greedily maximize the conditional replica score given the edge assignments already completed.

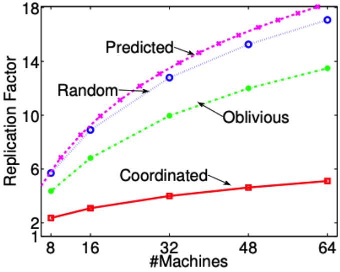
Edge balancing

- The previous theorem shows that as $\alpha \to 0$, the replication factor increases.
- But compared to edge cuts, vertex cuts does *significantly* better

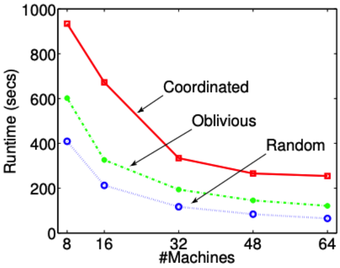


(a) V-Sep. Bound (b) V-Sep. Improvement

Replication factor in real graphs



(a) Replication Factor (Twitter)



(b) Ingress time (Twitter)