Hi!

CS 744: GRAPHX

Shivaram Venkataraman
Fall 2021
- Midterm grades today?
- Course Project: Check in by Nov 30th

Canvas = 1 page update of what you have done, what are roadblocks/challenges.
Scalable Storage Systems

Datacenter Architecture

Resource Management

Computational Engines

Applications

Machine Learning

SQL

Streaming

Graph

Powergraph

Applications

Machine Learning

SQL

Streaming

Graph

Powergraph

Computational Engines

Scalable Storage Systems

Resource Management

Datacenter Architecture

Applications

Machine Learning

SQL

Streaming

Graph

Powergraph

Computational Engines

Scalable Storage Systems

Resource Management

Datacenter Architecture
**POWERGRAPH**

**Programming Model:**
Gather-Apply-Scatter

**Better Graph Partitioning**
with vertex cuts

**Distributed execution**
(Sync, Async)

What is different from dataflow system e.g., Spark?

→ Specialized partitioning
   → lower communication

→ API was more graph specific
   → easy to express many algorithms

What are some shortcomings?

→ Fault tolerance
   → checkpoint of all vertices
GraphX
Can we efficiently map graph abstractions to dataflow engines?

Scalability! But at what COST?
When should we distribute graph processing?
**MOTIVATION**

- Specialized approach
  - Write out to files in-between systems
  - Modularity → update parts of "pipeline"

**Diagram Notes**

- Extract, Transform, Load
- ETL → Slice → Compute → Analyze
- GraphLab
- XML input data
- Adjacency list
- Connected components
- Visualize or evaluate
SYSTEM OVERVIEW

**Advantages?**

- **Hierarchical implementation**
- **Simplifies implementation**
- **Reuse!**
- **Fault tolerance etc.**
- **Base abstractions might not be suitable and need retrofit**

---

**Unified approach**

GraphX (2,500)

Spark (30,000)

GAS Pregel API (34)

PageRank (20)

Connected Comp. (20)

K-core (60)

Triangle Count (50)

LDA (220)

SVD++ (110)

**ETL**

**Evaluation/Analysis**

**Scheduling**

**Straggler mitigation**
class Graph[V, E] {
    // Constructor
    def Graph(v: Collection[(Id, V)],
          e: Collection[(Id, Id, E)])
    // Collection views
    def vertices: Collection[(Id, V)]
    def edges: Collection[(Id, Id, E)]
    def triplets: Collection[Triplet]
    // Graph-parallel computation
    def mrTriplets(f: (Triplet) => M,
                   sum: (M, M) => M): Collection[(Id, M)]
    // Convenience functions
    def mapV(f: (Id, V) => V): Graph[V, E]
    def mapE(f: (Id, Id, E) => E): Graph[V, E]
    def leftJoinV(v: Collection[(Id, V)],
                  f: (Id, V, V) => V): Graph[V, E]
    def leftJoinE(e: Collection[(Id, Id, E)],
                  f: (Id, Id, E, E) => E): Graph[V, E]
    def subgraph(vPred: (Id, V) => Boolean,
                 ePred: (Triplet) => Boolean)
        : Graph[V, E]
    def reverse: Graph[V, E]
}
mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)]

Gather: Edge → Accumulator
(C, S, D, E)

Sum: Sum

Return: Collection with messages aggregated at destination vertex

map operation after mrTriplets

val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
  if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(mapUDF, reduceUDF)
def Pregel(g: Graph[V, E],
    vprog: (Id, V, M) => V,
    sendMsg: (Triplet) => M,
    gather: (M, M) => M): = {

    g.mapV((id, v) => (v, halt=false))

    while (g.vertices.exists(v => !v.halt)) {
        val msgs: Collection[(Id, M)] =
            g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
                .mrTriplets(sendMsg, gather)
            g = g.leftJoinV(msgs).mapV(vprog)
    }

    return g.vertices
}
IMPLEMENTING TRIPLETs VIEW

Join strategy
Send vertices to the edge site
More edges than vertices, parallelize

Multicast join
Using routing table

Routing Table minimizes vertex state sent across the network

Join, map, group By for m Triplet

→ More edges than vertices, parallelize on * edges

Bitmask tracks active vertices

Not active anymore
SCALABILITY VS. ABSOLUTE PERFORMANCE

- GraphX: 3x from 8 to 32 machines
- PowerGraph: 2.6x from 8 to 32 machines

Graph showing runtime (s) for 10 iterations against number of nodes.

Notes:
- Slower but "better" scalability
- "better"
COST: CONFIGURATION THAT OUT-PERFORMS SINGLE THREAD

\[ C\#, \text{ single threaded program} \]

```rust
fn PageRank20(graph: GraphIterator, alpha: f32) {
    let mut a = vec![0f32; graph.nodes().len()];
    let mut b = vec![0f32; graph.nodes().len()];
    let mut d = vec![0f32; graph.nodes().len()];

    graph.map_edges(|x, y| { d[x] += 1; });

    for iter in 0..20 {
        for i in 0..graph.nodes().len() {
            b[i] = alpha * a[i] / d[i];
            a[i] = 1f32 - alpha;
        }

        graph.map_edges(|x, y| { a[y] += b[x]; });
    }
}
```

<table>
<thead>
<tr>
<th>scalable system</th>
<th>cores</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphLab [10]</td>
<td>128</td>
<td>249s</td>
</tr>
<tr>
<td>GraphX [10]</td>
<td>128</td>
<td>419s</td>
</tr>
<tr>
<td>Single thread (SSD)</td>
<td>1</td>
<td>300s</td>
</tr>
<tr>
<td>Single thread (RAM)</td>
<td>1</td>
<td>275s</td>
</tr>
</tbody>
</table>
DISCUSSION

https://forms.gle/u4TvMumnH7yBHd3b8
What are some reasons why GraphX or GraphLab or Naiad might be slower than a single thread implementation of PageRank?

- Communication overhead between nodes
  - Single thread = no communication
- Load balancing → one core/machine be slow and lead to overall slowdown
- Memory locality → all of this data fits in memory of one machine (n billion vertex & bytes = 4GB)
- GraphX implemented in Scala/JVM/Python, having C++/C++ overheads
How would you expect a single-thread QR implementation to perform?

- What is being computed:
  - Scalar addition/multiplication
  - Low compute

- First map stage is compute intensive ⇒ distributing makes more sense
GraphX: Combine graph processing with relational model

COST
- Configuration that outperforms single-thread
- Measure scalability AND absolute performance
  - Computation model of scalable frameworks might be limited
  - Hardware efficiency matters
- System/Language overheads
Next class: Marius
Project check-ins by Nov 29th
Filtered Index Scanning
- Store edges clustered on source vertex id
- Filter triplets using user-defined predicate

Automatic Join Elimination
- Some UDFs don’t access source or dest properties
- Inspect JVM byte code to avoid joins