CS 744: GRAPHX

Shivaram Venkataraman
Fall 2019
- Midterm grades are up!
- Course Project: Check in meetings Thu, Mon
**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?
- Fine-grained parallelism
- Less communication vertex cuts
- Not all are activated

What are some shortcomings?
- Static graphs
- Fault Tolerance
- Cross-graph analytics

Summary
GraphX

Can we efficiently map graph abstractions to dataflow engines?

Scalability! But at what COST?
When should we distribute graph processing?
SYSTEM OVERVIEW

Advantages:

Layering:

- Leverage lower-level
- Cluster reuse
- Codebase sharing
- View data: graph / table

GraphX (2,500)

Spark (30,000)

PageRank (20)
Connected Comp. (20)
K-core (60)
Triangle Count (50)
LDA (220)
SVD++ (110)
GAS Pregel API (34)
```java
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]
}
```

**Constructor**

- Vertext & Edge Collection

**Triplets**

- S.ID, D.ID, E, V.S, V.D

**Select from the edge table**

- and join vertex E

- and join vertex E

- dest: V.ID
mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)]

map: Triplet → Message → Apply

sum: Combine → Sum Power graph

Dest vertices → Final value of message associated with it
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 TRIPPLETS VIEW

Join strategy
- Send vertices to the edge site
  - Default: Use FS partitions
  - Number of vertices << 1E1

Multicast join
Using routing table

- A: 1, 2, 3
- B: 1

Incremental / Partial materialization
OPTIMIZING MR TRIPLETS

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
SCALABILITY VS. ABSOLUTE PERFORMANCE

GraphX
3x from 8 to 32 machines

PowerGraph
2.6x from 8 to 32
DISCUSSION

https://forms.gle/ARaU8Ce9XCpkZznn6
Consider a single-threaded PageRank implementation as shown and the performance comparison shown in the corresponding table. What could be some reasons for this performance gap?

- Graph X Slowest \rightarrow \text{immutable data lineage tracking} \rightarrow \text{Twitter}
- Single threaded \rightarrow \text{Avoid SSD lookup} \rightarrow \text{Graph fits in memory!}
  \hspace{1cm} 100M vertices | 400MB
  \hspace{1cm} 4B edges | 4GB
  \hspace{1cm} \approx 5GB
- Overhead of "distribution" \rightarrow \text{High!} \rightarrow \text{Sync in Pregel}
- Will it get better or worse every iteration? \text{PAGERANK has not much compute}
Now consider a distributed QR decomposition workload shown in Figure below with corresponding performance breakdown. How would you expect a single-thread implementation to perform here?
What are some workload properties that could explain the difference?
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: Weld
Project check-in meetings