CS 744: GRAPHX

Shivaram Venkataraman
Fall 2020
- Midterm grades are up!
- Course Project: Check in by Nov 20th
- Extra office hours for projects

项目经理

- Project Proposal
- Mid-semester update
  - HotCRP by Nov 20

- Poster presentation?
  - gather. town?
  - Short videos?

- Final report
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?
- Not all vertices are activated
- Fine-grained parallelism (Async)
- Vertex cuts → partition a graph → reduce communication for natural power-law graphs

What are some shortcomings?
- Integrate with existing dataflow systems
- Static graphs [no changes in structure]
- Fault tolerance
  > Powergraph checkpointing all workers even if one fails, all rollback
GraphX
Can we efficiently map graph abstractions to dataflow engines?

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

Writing out data to disk

Raw data
HTML/XML docs

ETL
Extract
Transform
Load

Transforming documents to graph

Slice

Contemporary Graph Processing Systems

Compute

Power Graph

General purpose computation + Specialized graph computation

Analyze

Report
SYSTEM OVERVIEW

```plaintext
<table>
<thead>
<tr>
<th>General purpose vs specialized</th>
</tr>
</thead>
</table>

General purpose

Layered arch

GAS Pregel API (34)

PageRank (20)  Connected Comp. (20)  K-core (60)  Triangle Count (50)  LDA (220)  SVD++ (110)

**GraphX** (2,500)

Spark (30,000)

Can share cache

Advantages?

→ No data movement or duplication
→ Reuse features from Spark
→ Same machines can run Spark or GraphX
→ View same data as graph or table
```
PROGRAMMING MODEL

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
- Vertices and Edges
  \[
  \text{Vertices} \quad \begin{array}{c|c|c|c}
  \hline
  \text{Id} & \text{S} & \text{D} & \text{E} \\
  \hline
  1 & 2 & 3 & 0.5 \\
  \hline
\end{array}
\]

Triplets
  \[
  \text{Triplet} = (S.I, D.I, E, S.V, D.V)
  \]

Select rows from EDGES
and join VERTEX \ E.S = V.ID
and join VERTEX \ E.D = V.ID
mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[((Id, M)]

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)
```scala
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
- create triplets \( \{s \text{ID}, s \text{V}, e \text{D}, e \text{V} \} \)
  - like a shuffle but taking into account graph structure
  - Power-law graphs

Multicast join
Using routing table
- \( \{1, 2, 3\} \) is used twice in Edge Partition A
- \( 1, 2, 3 \) are only vertices used in A

Join strategy
- create triplets \( \{s \text{ID}, s \text{V}, e \text{D}, e \text{V} \} \)
  - like a shuffle but taking into account graph structure
  - Power-law graphs
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

\[
\begin{align*}
\text{Map} & : 0 \rightarrow 1 \\
& : 0 \rightarrow 2 \\
& : 1 \rightarrow 3 \\
\end{align*}
\]

\[
\begin{align*}
f \text{ only uses source state} & \quad \iff \quad t, \text{source, state} = 1 \\
& \text{return } 1; \\
& \text{return } 0 \\
\end{align*}
\]
SCALABILITY VS. ABSOLUTE PERFORMANCE

GraphX
3x from 8 to 32 machines

PowerGraph
2.6x from 8 to 32
DISCUSSION

https://forms.gle/Urs8PFDnmaud5uZo7
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?

- Single core perf vs perf of 128 cores!!
- Application is not suitable for prog. model?
  - Comm, sync overheads!
  - Reducing effectiveness of compute
- Single node vs no overheads from distribution
- Graph fits in RAM / single SSD vs Big Data?
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?

QR would be worse in a single threaded case

1st stage is compute heavy  
(Page Rank: compute is addition & scalar multiply)

Tree reduce takes very little time  
Comm. overheads are low

Distributed computation is good!
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: PyTorch BigGraph
Project check-ins by Nov 20th