Distributed Graph processing
with BSP, Pregel and DataFrames

Pelle Jakovits

30 November 2018, Tartu
Outline

• Distributed Graph processing
• Bulk Synchronous Parallel model
• Using BSP for graph processing
  – Pregel model
  – Pregel based Graph processing frameworks
• Pregel-like Graph processing
  – Spark GraphX
  – Spark GraphFrames
Reminder: Graphs!

\[ G = (V, E) \]

- **V** represents the set of **Vertices** (Nodes)
- **E** represents the set of **Edges** (Links between Nodes)
- Both vertices and edges may contain additional information
  - Vertices have a unique ID and state
  - Edges may be associated with weight, transition cost, type, etc
- Graphs can be:
  - Directed or undirected
  - Cyclic and Acyclic
- Represent relationships in data
Reminder: Graph Problems

- **Social networks**
  - $V$: People, $E$: Relationships (friend, family, colleague)
  - **Problems**: Friend recommendations, detecting communities

- **Web Graphs**
  - $V$: Web pages, $E$: HyperLinks
  - **Problems**: Ranking results in search based on linking frequency

- **Financial networks**
  - $V$: Accounts, $E$: Transitions
  - **Problems**: Detecting fraud rings

- **Product, Review, User networks** (**Yelp** dataset)
  - $V$: Users, Businesses, $E$: Friendships, Reviews
  - **Problems**: Collaborative filtering, personalized product search
Graph processing with MapReduce

• Recall the approach from *Graph Data Processing with MapReduce* lecture
  – Represent input graph as a adjacency matrix or list
  – Graph algorithms as iterative sequence of MapReduce jobs
  – Whole graph data is passed along through the Map-Reduce job chain
    • Map, Shuffle, Reduce
• Can handle very large graphs
• However, not efficient for many graph algorithms
MapReduce and Graph Algorithms

• Algorithms based on graph traversal can be highly iterative
  – States of only a few Vertices may be modified at every iteration
  – Small computation per Vertex

• Vertex centric algorithms would benefit from data locality
  – Vertex state depends only on the state of neighbours
  – Very little computation per vertex

• Map and Reduce functions are essentially stateless
  – Can't keep data in memory between MR iterations
  – Entire graph needs to be passed along to the next stage
  – Each iteration adds a significant overhead
Alternative approach

• Use parallel computing models more suitable for iterative graph algorithms
  – Such as Bulk Synchronous Parallel and Pregel
• Use Frameworks that support caching data in memory between iterations
Bulk Synchronous Parallel model

• **Bulk Synchronous Parallel (BSP)** is a distributed computing model for high-performance iterative computations

• Computations consist of a sequence of Super-steps

• Each super-step consist of 3 stages:
  1. Concurrent local computation
  2. Communication between threads
  3. Barrier synchronisation
BSP model

- Compared to MapReduce it is more suited for graph algorithms
  - Current state is kept in memory between iterations
  - Only updates to the state are synchronized between processes
  - Supports large number of iterations
  - And computationally less intensive iterations
- Message passing is used for data synchronization
- Bulk communication is easier to manage and optimize
- Using barrier synchronisation means that each computation is 100% independent (inside a super-step)
BSP for Graph Processing

• BSP is an old parallel computing model from 80’s
  – Low level libraries: Oxford BSPlib, Padeborn BSP, BSPonMPI
• Google started using BSP model for Large scale Graph data processing in Pregel framework
• Pregel was never made directly available outside Google
• A number of open source Pregel implementations were launched:
  – Graph processing: jPregel & Apache Giraph
  – General purpose: Apache Hama
Pregel model for graph processing

- Based on Bulk Synchronous Parallel model
- Vertices and edges are stored in memory
- Each vertex contains an id and a modifiable state
- Computations consists of sequences of BSP Super-steps
  1. User defined function is applied on each Vertex
  2. Each Vertex generates state update messages to neighbors
  3. Global barrier synchronization
- Vertices can only receive messages from the previous iteration and modify its own state and states of outgoing edges
- Only messages are shuffled between concurrently working processes
- Computation is terminated when there are no more messages sent
Pregel Frameworks

• **Apache Giraph**
  – Direct Pregel implementation in Java
  – Built on top of Apache Hadoop (Yarn, HDFS)

• **Apache Hama**
  – General purpose BSP in Java
  – Pregel model for graph processing
  – Built on top of Apache Hadoop (Yarn, HDFS)

• **Apache Spark**
  – Only partially follows the Pregel model for graph processing:
  – Spark GraphX (Java, Scala)
  – Spark GraphFrames (Python, Scala)
Pregel super-step pseudo code

• At each Super-step $S$:
  – For each vertex $V$:
    1. Read messages received from step $S-1$
    2. Apply user defined function `compute()` on $V$
       – $V$.state = `compute($V$.state, messages)`
    3. Generate new messages for outgoing edges
       – `sendMessage(edge.dst, message)`;
    4. Wait for barrier synchronisation
Example: Maximum value

Superstep 0

Superstep 1

Superstep 2

Superstep 3

Apache Giraph: compute() maximum

```java
public void compute(Iterable<DoubleWritable> messages) {
    double maxDist = Double.MIN_VALUE;
    for (DoubleWritable message : messages) {  // process incoming messages
        maxDist = Math.max(maxDist, message.get());
    }
    if (maxDist > getValue().get()) {  // update current state
        setValue(new DoubleWritable(maxDist));
        for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
            sendMessage(edge.getTargetVertexId(), new DoubleWritable(maxDist));
        }
    }
    voteToHalt();  // vote to halt
}
```

Source: http://giraph.apache.org/intro.html
Fault tolerance

• Vertices are partitioned between concurrent workers
• Fault tolerance is based on checkpointing
  – Workers save state of partitions to persistent storage at checkpoint
  – Vertex, Edge state at the current Super-Step and all incoming messages
• Ping messages to track availability
• On failure, reassign partitions and revert to last checkpoint
  – Restore computation from the latest checkpointed super-step
Graph processing in Spark

• GraphX
  – Only Java and Scala is supported

• GraphFrames
  – GraphX Extension based on DataFrames
    • Scala and Python

• Implemented graph processing algorithms
  – PageRank, Connected components, Label propagation, SVD++, Strongly connected components, Triangle count

• Pregel-like graph processing using `aggregateMessages` primitive
Spark GraphX

• Spark component for distributed graph processing

• Introducing a Graph abstraction over RDD’s
  – a directed multigraph with properties attached to each vertex and edge.

• Graphs are defined by two RDD’s: Vertices and Edges
  – Manipulate Vertices and Edges using RDD transformations
  – Write custom iterative graph algorithms using Pregel API

• GraphX contains a set of fundamental Graph operators
  – Subgraph, joinVertices, aggregateMessages
  – As well as an optimized variant of the Pregel API

• Contains a number of common graph algorithms:
  – PageRank, Connected components, Label propagation, SVD++, Triangle count
Property graph example

Property Graph

Vertex Table

<table>
<thead>
<tr>
<th>Id</th>
<th>Property (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(rxin, student)</td>
</tr>
<tr>
<td>7</td>
<td>(jgonzal, postdoc)</td>
</tr>
<tr>
<td>5</td>
<td>(franklin, professor)</td>
</tr>
<tr>
<td>2</td>
<td>(istoica, professor)</td>
</tr>
</tbody>
</table>

Edge Table

<table>
<thead>
<tr>
<th>SrcId</th>
<th>DstId</th>
<th>Property (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>Collaborator</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Advisor</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Colleague</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>PI</td>
</tr>
</tbody>
</table>

Spark GraphFrames

• DataFrame based extension for Graph processing in Spark.
• Can be used in Scala and Python
• Graph is defined by a Vertex and Edge DataFrames
  – Can be freely manipulated using DataFrame operations
  – Provides many opportunities to manipulate the graph
    • Filter, join, union, …
• Does not have a direct Pregel API method
• Pregel-Like execution can be achieved by iterating over aggregateMessages primitive
Creating GraphFrames

• Users create GraphFrames from two DataFrames:
  – Vertex DataFrame
    • Columns: ID, Attribute1, ..., AttributeN
  – Edge DataFrame
    • Columns: Src, Dst, Attribute1, ..., AttributeN

• Can use any number of attribute columns
  – Allows to specify different node and vertex types
  – Allows to attach arbitrary state to columns
  – Graph Edges and Vertices can be filtered and queried by specific attribute column values
GraphFrames YELP example

• **Edges:**
  – Users and Companies
  – Columns: \texttt{id}, \texttt{name}, \texttt{type}, \texttt{city}

• **Vertices:**
  – User \texttt{Reviewed} Company
  – User \texttt{is friend with} a User
  – Columns: \texttt{src}, \texttt{dst}, \texttt{relationship}, \texttt{stars}
User Vertices

```python
business = spark.read.option("inferSchema", True).json("business")
businessV = business.withColumn("type", F.lit("company")) .
    .select(F.col("business_id").alias("id"),
            "name", "type", "city")
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>type</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>V90fC_aF-_DNYzQvUtblww</td>
<td>Jayde Fuzion</td>
<td>company</td>
<td>Henderson</td>
</tr>
<tr>
<td>zrDi4gEaUi64lAMfJU51dw</td>
<td>Houston's Restaurant</td>
<td>company</td>
<td>Scottsdale</td>
</tr>
<tr>
<td>PChG1Dm0A6AXIXkXGVK8Fw</td>
<td>Sasa Sushi</td>
<td>company</td>
<td>Las Vegas</td>
</tr>
<tr>
<td>wk6aHP-vxv9dhmmJVDWnPg</td>
<td>Crews &amp; Tangos</td>
<td>company</td>
<td>Toronto</td>
</tr>
<tr>
<td>zsYYAnHEs5qzsfazSsGkBw</td>
<td>Diya Eyebrow Threading</td>
<td>company</td>
<td>Las Vegas</td>
</tr>
<tr>
<td>lmCRn3mZ89ThvozQ3gtaw</td>
<td>Michael Kors</td>
<td>company</td>
<td>Chandler</td>
</tr>
</tbody>
</table>
Company Vertices

```python
user = spark.read.option("inferSchema", True).json("user")
userV = user.withColumn("type", F.lit("user")).withColumn("city", F.lit"
.select(F.col("user_id").alias("id"),
        "name", "type", "city")
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>type</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZcsZdHLiJGVvDHVjeTYYnQ</td>
<td>Stacey X Joe</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td>ZcXOWHZtluEliljctm5ibw</td>
<td>Tress</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td>ZckGMvFYMUbmooEg0L6Estw</td>
<td>Smita</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td>ZczGdiKox062ZA9tgpv34g</td>
<td>Mike</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td>Zcoy9912EmxHsbtN8UVeRQ</td>
<td>Donovan</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td>ZcBoDxrxz4t3Pcs96_QIIbw</td>
<td>Zubin</td>
<td>user</td>
<td></td>
</tr>
</tbody>
</table>
Review Edges

```python
review = spark.read.option("inferSchema", True).json("review")
reviewE = review.withColumn("relationship", F.lit("reviewed"))
    .select(F.col("user_id").alias("src"),
            F.col("business_id").alias("dst"),
            "relationship", "stars")
```

<table>
<thead>
<tr>
<th>src</th>
<th>dst</th>
<th>relationship</th>
<th>stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZcgHNMGUk-p-J9hRqgoDx6A</td>
<td>1ForN8iXqYZ_dZZcOkvVeA</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>ZcSQVbrIjIxuWvSAMN9kfQ</td>
<td>5Vt4PqzWwObmQeuhgMNsDQ</td>
<td>reviewed</td>
<td>1</td>
</tr>
<tr>
<td>Zc899qRSiZuefMhERJ8fYg</td>
<td>4mYS-4U0jTKgsf0tX1_IkQ</td>
<td>reviewed</td>
<td>4</td>
</tr>
<tr>
<td>Zc899qRSiZuefMhERJ8fYg</td>
<td>b1UvPlDmVtN0ATBVfe6wBw</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Zc899qRSiZuefMhERJ8fYg</td>
<td>N8FdrK10y_E_mU0j8VSFUa</td>
<td>reviewed</td>
<td>4</td>
</tr>
<tr>
<td>ZcvKb_1gXb4D7ISGRTPjsQ</td>
<td>JUY-6k02E-Zf5QZXhnKabA</td>
<td>reviewed</td>
<td>5</td>
</tr>
</tbody>
</table>
user = spark.read.option("inferSchema", True).json("user")
friendE = user.withColumn("relationship", F.lit("friend")) \
  .withColumn("stars", F.lit("")) \
  .select(F.col("user_id").alias("src"), 
    F.explode(F.split("friends", ",\")).alias("dst"), 
    "relationship", "stars")

<table>
<thead>
<tr>
<th>src</th>
<th>dst</th>
<th>relationship</th>
<th>stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZcYNpemiHw20OpQxkVbrVg</td>
<td>zUKKJhPFlnFRrVfi6hYk6g</td>
<td>friend</td>
<td></td>
</tr>
<tr>
<td>ZcYNpemiHw20OpQxkVbrVg</td>
<td>ELXw24LymsSMhHCgNjGJ7Q</td>
<td>friend</td>
<td></td>
</tr>
<tr>
<td>ZcysmZjtubs-8jZKJwjJDig</td>
<td>Nnki4lJhXJ2H1ew_wTr1mQ</td>
<td>friend</td>
<td></td>
</tr>
<tr>
<td>ZczGdiKox062ZA9tgpv34g</td>
<td>8OVdLDDkbYuPguafWhTznw</td>
<td>friend</td>
<td></td>
</tr>
<tr>
<td>ZczGdiKox062ZA9tgpv34g</td>
<td>sASwKlOPa81UijLYVSjqfg</td>
<td>friend</td>
<td></td>
</tr>
</tbody>
</table>

26/45
Create the GraphFrame

• Union different types of Edge and Vertex DataFrames together

```python
all_vertices = userV.union(businessV)
all_edges = reviewE.union(friendE)
```

• Create the GraphFrame object

```python
g = GraphFrame(all_vertices, all_edges)
```

• Can check the current values of Graph DataFrames

```python
g.vertices.show(10)
g.edges.show(10)
```
GraphFrames Example

*Shobeir* is going to *Las Vegas* and would like a recommendation for a good restaurant based on what his friends have liked

• Lets use Breath First Search (BFS) to find shortest path between the user and businesses in *Las Vegas*
  – Using *friendship* and *review* type edges
  – Where reviews have maximum stars
Breath First Search (BFS)

- BFS is implemented as a query in Spark GraphFrames API
- User specifies:
  - Source Vertex filter
  - Destination vertex filter
  - Edge filter: what types of edges are allowed
  - Maximum path length
- Returns a DataFrame that contains paths between source and destination

```
paths = graphFrame.bfs(SourceFilter, DestinationFilter, EdgeFilter, MaxPathLength)
```
BFS example

- Find path from user to businesses through friend or review relationships (review stars = 5).

```g.bfs("name = 'Shobeir'",
    "city = 'Las Vegas'",
    EdgeFilter = "stars = 5 or relationship = 'friend'",
    MaxPathLength = 4)```

- BFS Returns a DataFrame that contains paths from source to destination
  - First column from is the source vertex
  - Second column e0 is Edge
  - Last column to is destination vertex
  - Additional Vertex (vX) and Edge (eX) between second and last columns for each additional link in the path
Querying BFS result

```python
paths.select("from.name",
          "e0.relationship",
          "v1.name",
          "e1.relationship", "e1.stars",
          "to.name", "to.city")
```

<table>
<thead>
<tr>
<th>From</th>
<th>e0</th>
<th>v1</th>
<th>e1</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>relationship</td>
<td>name</td>
<td>relationship</td>
<td>stars</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
<tr>
<td>Shobeir</td>
<td>friend</td>
<td>Eric</td>
<td>reviewed</td>
<td>5</td>
</tr>
</tbody>
</table>
Other available Graph operations

- **Connected components** - Maps each vertex to a connected component
  - `g.connectedComponents()`
- **Strongly connected components** - Find components, where every vertex is reachable from every other vertex
  - `g.stronglyConnectedComponents(maxIter=10)`
- **Label Propagation Algorithm (LPA)** - Find communities: dense internal structure, sparse connections between groups
  - `g.labelPropagation(maxIter=5)`
- **Shortest paths** - Find shortest paths to a set of landmark vertices
  - `g.shortestPaths(landmarks=['a', 'd'])`
- **Triangle count** - Find number of triangles passing through each vertex
  - `g.triangleCount()`
Neighborhood Aggregation

- The core of many Vertex-centric graph algorithms is aggregating information from the Vertex neighbors.
- Iterative graph algorithms iteratively perform Neighborhood Aggregation, each result modifying the current state.
  - BFS Shortest paths, PageRank, connected components.
- Pregel model can be viewed as Iterative Neighborhood Aggregation.

- `aggregateMessages()` operation is the main BSP-Like operation for designing user defined graph algorithms in Spark.
  - Implemented in both GraphX and GraphFrames.
aggregateMessages()

df = gf.aggregateMessages(aggColFun, messageToSrc, messageToDs)

• Generates messages from Edge to connected Vertices, based on:
  – Edge DataFrame column values
  – Source and Destination Vertex DataFrame column values

• Aggregation column function combines all messages into a single column value

• Result is a DataFrame containing two columns
  – Vertex id, message aggregation

• Result can be joined into the original graph using DataFrame joins to update the current state of Vertices

• Iterate over aggregateMessages() to achieve BSP like Super-step cycle
Custom Shortest paths example

• Implement BFS based shortest path search using aggregateMessages()
• Source user Vertex: Eric
• Find shortest path distance from Eric to all reachable nodes
• Prepare the initial state of the GraphFrame
  – Add current distance column dist to vertices
  – Set Eric’s distance to 0, all other distances to -1

filtVert = g.vertices.withColumn('dist',
    F.when(F.col("name") == 'Eric', F.lit(0))
    .otherwise(F.lit(-1)))

gx = GraphFrame(filtVert, g.edges)
Breath First Search
Pregel Super-Step Cycle

# Super-step cycle
for iter_ in range(numIter):

    messages = gx.aggregateMessages(
        F.min(AM.msg).alias("newD"),
        sendToDst = F.when(AM.src['dist'] >= 0, AM.src['dist'] + 1))
    messages = messages.filter(F.col("newD").isNotNull())

    newVert = gx.vertices.join(F.broadcast(messages), "id", how = 'left_outer') \
        .withColumn("dist", upd_dist_udf("newD", "dist")) \
        .drop("newD")

    cachedVert = AM.getCachedDataFrame(newVert)  # Update GraphFrame
    gx = GraphFrame(cachedVert, gx.edges)
Aggregate Messages

- Generates a message for each Edge based on Vertex and Edge state
- Vertex aggregates received messages into a news state update value
- Result is a DataFrame, containing: (id, aggregated_value)

```python
messages = gx.aggregateMessages('F.min(AM.msg).alias("newD"),
                 sendToDst = F.when(AM.src['dist'] >= 0, AM.src['dist'] + 1))
messages = messages.filter(F.col("newD").isNotNull())
```

<table>
<thead>
<tr>
<th>id</th>
<th>newD</th>
</tr>
</thead>
<tbody>
<tr>
<td>wZukjLaf1V2dLRQap_Zriw</td>
<td>1</td>
</tr>
<tr>
<td>JX9XWEV_9_N1EicEF35c9w</td>
<td>1</td>
</tr>
<tr>
<td>rRUMh9qvfW8wA6NUd5Yf8w</td>
<td>1</td>
</tr>
<tr>
<td>82I9SSilxh3zL8F1UQ45Xw</td>
<td>1</td>
</tr>
<tr>
<td>bzqf7gJHx4QDPYupGSuRtg</td>
<td>1</td>
</tr>
<tr>
<td>S-B-ak8Kq0DrkLfE7k3DGg</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Update GraphFrame Vertex State

- Join aggregated messages into Vertices using left outer join
- Update distance column based on aggregated message values

```python
v = gx.vertices
newVert = v.join(F.broadcast(messages), "id", how = 'left_outer') \
    .withColumn("dist", upd_dist_udf("newD", "dist")) \
    .drop("newD")

upd_dist_udf = F.udf(upd_dist, IntegerType())
def upd_dist(new, old):
    if(new == None):
        return old
    if(old < 0):
        return new
    return min(new, old)
```

#UDF for updating Current distance
Update GraphFrame

• Update GraphFrame vertices for the next Super-Step

cachedVert = AM.getCachedDataFrame(newVert)
gx = GraphFrame(cachedVert, gx.edges)
Results

gx.vertices.filter("distance > 0") \ 
.orderBy(F.col("distance").desc()) 
.drop("id")

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>city</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dairy Queen</td>
<td>company</td>
<td>Phoenix</td>
<td>3</td>
</tr>
<tr>
<td>Charity Towing &amp; Recovery</td>
<td>company</td>
<td>Phoenix</td>
<td>3</td>
</tr>
<tr>
<td>P.F. Chang's</td>
<td>company</td>
<td>Phoenix</td>
<td>3</td>
</tr>
<tr>
<td>Yese</td>
<td>user</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Element Carpet Cleaning</td>
<td>company</td>
<td>Peoria</td>
<td>3</td>
</tr>
<tr>
<td>Brian</td>
<td>user</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
BSP/Pregel advantages

• More efficient for graph algorithms compared to MapReduce
  – Supports large number of iterations
  – Supports less computationally intensive iterations
  – Suitable for graph traversal based algorithms where only a few Vertices may be modified at each Super-step

• Takes advantage of In-memory computations

• Good data locality
  – Only messages are synchronized between processes
  – Avoids shuffling data around
BSP/Pregel disadvantages

- Can’t be used when Graph’s Vertices, Edges and state does not fit into the distributed memory of the cluster
- Fault tolerance requires periodical checkpointing
  - Performance overhead when nothing fails
- Global barrier introduces overhead that can become significant as the number of cores increases
- Does not support real-time or interactive graph querying
  - Use Graph databases Neo4J or Amazon Neptune
  - Graph query languages like SparQL
- Manipulating the structure of the graph is somewhat cumbersome
  - Use Apache Spark GraphX and GraphFrames
That’s All

• Next weeks practice session is
  – Distributed graph processing with Spark GraphFrames

• Next week's lecture is
  – Non-Relational databases
References

2. GraphFrame User Guide https://graphframes.github.io/