A Performance Study of an Implementation of the Push-Relabel Maximum Flow Algorithm in Apache Spark's GraphX

Ryan P. Langewisch
Advised by Dinesh P. Mehta
Background Motivation

- “Big Data” has pushed parallel computing to be more and more necessary.
Background Motivation

• “Big Data” has pushed parallel computing to be more and more necessary.

• As a result, parallel programming technologies have been developed (e.g. MapReduce)
Background Motivation

- “Big Data” has pushed parallel computing to be more and more necessary.

- As a result, parallel programming technologies have been developed (e.g. MapReduce)

- Many algorithmic solutions to problems need to be revisited in parallel.
Apache Spark

- Utilizes the MapReduce paradigm
Apache Spark

- Utilizes the MapReduce paradigm
- Accessible and open-source
Apache Spark

- Utilizes the MapReduce paradigm
- Accessible and open-source
- Built in Scala, based on “Resilient Distributed Datasets”, or RDDs
Resilient Distributed Datasets (RDDs)

- Data partitioning abstraction
Resilient Distributed Datasets (RDDs)

- Data partitioning abstraction
- Achieves fault-tolerance through lineage
Resilient Distributed Datasets (RDDs)

- Data partitioning abstraction
- Achieves fault-tolerance through lineage
- Allows caching of data between parallel operations
GraphX

- Spark's API for graphs and graph-parallel computation.
GraphX

- Spark's API for graphs and graph-parallel computation.

- Allows for data to be viewed as both a graph and a collection simultaneously.
GraphX

- Spark's API for graphs and graph-parallel computation.

- Allows for data to be viewed as both a graph and a collection simultaneously.
Simple GraphX Example

• What if we wanted to find the oldest follower of each person in the graph?
// Find the oldest follower for each user
val oldestFollower: VertexRDD[(String, Int)] =
    userGraph.aggregateMessages[(String, Int)](
        // Map Function
        edge => edge.sentToDst((edge.srcAttr.name, edge.srcAttr.age)),

        // Reduce Function
        (a, b) => if (a._2 > b._2) a else b)

Maximum-Flow Problem

What is the maximum flow that can be pushed from the source vertex to the sink vertex?

![Graph diagram]

The graph shows the network with capacities on the edges. The source vertex is labeled 's' and the sink vertex is labeled 't'. The capacities are indicated on the edges as follows:

- Edge from 's' to 'p': 2/3
- Edge from 'p' to 'o': 3/3
- Edge from 'o' to 'q': 3/3
- Edge from 'q' to 't': 2/2
- Edge from 'p' to 'r': 0/2
- Edge from 'r' to 't': 3/3
- Edge from 's' to 't': 1/4

The maximum flow problem asks for the maximum amount of flow that can be pushed from the source to the sink, respecting the capacities of the edges.
Push-Relabel Algorithm

- Solution that is more inherently parallelizable than alternatives such as Ford-Fulkerson
Push-Relabel Algorithm

- Solution that is more inherently parallelizable than alternatives such as Ford-Fulkerson
- Utilizes the concept of “preflow”
Push-Relabel Algorithm

- Solution that is more inherently parallelizable than alternatives such as Ford-Fulkerson
- Utilizes the concept of “preflow”
- Labeling mechanism monitors which vertices are eligible to push excess flow
Push-Relabel Example

Source: http://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm
Push-Relabel Example
Push-Relabel Example
Project Goal

“Implement a solution to the maximum-flow problem in GraphX, targeting the Push-Relabel algorithm as our approach.”
GraphX Pregel API

- GraphX provides a Pregel operator recommended for iterative algorithms
GraphX Pregel API

- GraphX provides a Pregel operator recommended for iterative algorithms
- Requires 3 user-define functions:
GraphX Pregel API

- GraphX provides a Pregel operator recommended for iterative algorithms

- Requires 3 user-define functions:
  - “Send Message” function
GraphX Pregel API

- GraphX provides a Pregel operator recommended for iterative algorithms

- Requires 3 user-define functions:
  - “Send Message” function
  - “Merge Message” function
GraphX Pregel API

- GraphX provides a Pregel operator recommended for iterative algorithms

- Requires 3 user-define functions:
  - “Send Message” function
  - “Merge Message” function
  - “Vertex Program” function
GraphX Pregel API Consideration

• Basic Approach
  - Use the **Send Message** step to find possible pushes or relabels in the graph.
GraphX Pregel API Consideration

- Basic Approach
  - Use the **Send Message** step to find possible pushes or relabels in the graph.
  - Use the **Merge Message** step to choose which operations will be executed based on excess.
GraphX Pregel API Consideration

- **Basic Approach**
  - Use the **Send Message** step to find possible pushes or relabels in the graph.
  - Use the **Merge Message** step to choose which operations will be executed based on excess.
  - Use the **Vertex Program** step to update the values of the graph.
GraphX Pregel API Consideration

- Ran into problems with updating both the source and destination of a push.
GraphX Pregel API Consideration

- Ran into problems with updating both the source and destination of a push.
New Approach Visualization

“Surveying” Step

Possible push from A

Other possible pushes from A

Select pushes and store in vertex data

Subtract push amount from excess at A

“Execution” Step

Check push info stored in A

Other pushes to B

Add push to excess at B
Handling Relabeling

- Relabel information also needs to be included in the messages.
Handling Relabeling

- Relabel information also needs to be included in the messages.

- While mapping, find the lowest neighboring height label.
Relabeling Visualization

“Surveying” Step

Height Labels of other neighboring vertices

Height Label of B

Reduce and store smallest height label in vertex data. If greater than or equal to the height label of A, relabel.

“Execution” Step

No actions, relabeling is already complete.
Simple Example

A → B: 5/5
B → C: 0/2
B → E: 0/2
D → E: 0/2
D → C: 0/2
B → D: 0/2
A → D: E-0 L-5
Simple Example: Iteration 1 - “Surveying”

From AB to B – (Map(), 5)
From BC to B – (Map(), 0)
From BD to B – (Map(), 0)
Reduce
B – (Map(), 0)
Simple Example: Iteration 1- “Surveying”

From AB to B – (Map(), 5)
From BC to B – (Map(), 0)
From BD to B – (Map(), 0)
Reduce B – (Map(), 0)
Relabel B to “1”
Simple Example: Iteration 1- “Execution”

No Possible Pushes, all vertices and edges are mapped to their original values.

Resulting Graph:
Simple Example: Iteration 2 - “Surveying”

From AB to B – (Map(), 5)
From BC to B – (Map((3L, true) → 2), 0)
From BD to B – (Map((4L, true) → 2), 0)
Simple Example: Iteration 2 - “Surveying”

From AB to B - (Map(), 5)
From BC to B - (Map((3L, true) → 2), 0)
From BD to B - (Map((4L, true) → 2), 0)

B - (Map((3L, true) ->2, (4L, true) ->2), 0)
Simple Example: Iteration 2 - “Surveying” Vertex Program

\[ B - (\text{Map}((3L, \text{true}) \rightarrow 2, (4L, \text{true}) \rightarrow 2), 0) \]

Loop over possible pushes

Push 1 (Excess at B = 5):  \[5 \geq 2, \text{select push and subtract excess.}\]

Push 2 (Excess at B = 3):  \[3 \geq 2, \text{select push and subtract excess.}\]

Data stored at Vertex B:  \((1, 1, \text{Map}((3L, \text{true}) \rightarrow 2, (4L, \text{true}) \rightarrow 2))\)
Simple Example: Iteration 2 - “Execution”

Vertex Data at B: \((1, 1, \text{Map}((3L, \text{true}) \rightarrow 2, (4L, \text{true}) \rightarrow 2))\)

**Update Edges**
- BC increases its flow by 2
- BD increases its flow by 2

**Update Vertices**
- C updates its excess by 2
- D updates its excess by 2
Simple Example: Iteration 3 - “Surveying”

From AB to B – (Map(), 5)
From BC to C – (Map(), 1)
From BD to D – (Map(), 1)
From CE to C – (Map(), 0)
From DE to D – (Map(), 0)

Reduce

B – (Map(), 5)
C – (Map(), 0)
D – (Map(), 0)

All three vertices relabel
Simple Example: Iteration 3 - “Execution”

No Possible Pushes, all vertices and edges are mapped to their original values.

Resulting Graph:
Simple Example: Iteration 4 - “Surveying”

From AB to B – (Map((1L, false) → 5), 5)
From BC to C – (Map(), 1)
From BD to D – (Map(), 1)
From CE to C – (Map((5L, true) → 2), 0)
From DE to D – (Map((5L, true) → 2), 0)

Reduce

B – (Map((1L, false) → 5), 5)
C – (Map((5L, true) → 2), 0)
D – (Map((5L, true) → 2), 0)
Simple Example: Iteration 4 - “Surveying” Vertex Program

Messages

B – (Map((1L, false) → 5), 5)  →  Excess of 1  →  (0, 6, Map((1L, false) → 1))

Stored Vertex Data
Simple Example: Iteration 4 - “Surveying” Vertex Program

Messages

B - (Map((1L, false) → 5), 5)

Excess of 1

 Stored Vertex Data

(0, 6, Map((1L, false) → 1))

C - (Map((5L, true) → 2), 0)

Excess of 2

(0, 1, Map((5L, true) → 2))
### Simple Example: Iteration 4 - “Surveying” Vertex Program

<table>
<thead>
<tr>
<th>Messages</th>
<th>Stored Vertex Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>B - (Map((1L, false) → 5), 5)</td>
<td>(0, 6, Map((1L, false) → 1))</td>
</tr>
<tr>
<td>Excess of 1</td>
<td></td>
</tr>
<tr>
<td>C - (Map((5L, true) → 2), 0)</td>
<td>(0, 1, Map((5L, true) → 2))</td>
</tr>
<tr>
<td>Excess of 2</td>
<td></td>
</tr>
<tr>
<td>D - (Map((5L, true) → 2), 0)</td>
<td>(0, 1, Map((5L, true) → 2))</td>
</tr>
<tr>
<td>Excess of 2</td>
<td></td>
</tr>
</tbody>
</table>
**Simple Example: Iteration 4 - “Surveying” Vertex Program**

<table>
<thead>
<tr>
<th>Messages</th>
<th>Stored Vertex Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>B - (Map((1L, false) → 5), 5)</td>
<td>Excess of 1 → (0, 6, Map((1L, false) → 1))</td>
</tr>
<tr>
<td>C - (Map((5L, true) → 2), 0)</td>
<td>Excess of 2 → (0, 1, Map((5L, true) → 2))</td>
</tr>
<tr>
<td>D - (Map((5L, true) → 2), 0)</td>
<td>Excess of 2 → (0, 1, Map((5L, true) → 2))</td>
</tr>
</tbody>
</table>

Pushed flow is subtracted from the excess at vertices B, C, and D.
Simple Example: Iteration 4 - “Execution”

B - (0, 6, Map((1L, false) → 1))
C - (0, 1, Map((5L, true) → 2))
D - (0, 1, Map((5L, true) → 2))

Edges (CE, DE, AB) and Vertices (A, E) Update
Simple Example: Iteration 5

- No excess in the graph (excluding the source and sink) leads to no messages.
Simple Example: Iteration 5

- No excess in the graph (excluding the source and sink) leads to no messages.
- Main execution loop terminates, and the maximum flow has been found.
Checkpointing

- RDD lineage grows with each iteration, eventually causing a stack overflow.
Checkpointing

- RDD lineage grows with each iteration, eventually causing a stack overflow.

- A checkpoint saves the RDD to an HDFS file and truncates the lineage entirely.
Checkpointing

- RDD lineage grows with each iteration, eventually causing a stack overflow.

- A checkpoint saves the RDD to an HDFS file and truncates the lineage entirely.

- Implemented by simply calling the checkpoint method after a set number of iterations.
Caching

- RDDs are not cached in memory by default.
Caching

- RDDs are not cached in memory by default.

- When an uncached RDD is accessed again, it must be recomputed.
Caching

- RDDs are not cached in memory by default.

- When an uncached RDD is accessed again, it must be recomputed.

- Especially important with iterative algorithms.
Caching

- RDDs are not cached in memory by default.
- When an uncached RDD is accessed again, it must be recomputed.
- Especially important with iterative algorithms.
- **Simply call the cache method on the graph.**
Restricted Active Set

- Only a small percentage of the vertices are eligible for an operation at one time.
Restricted Active Set

- Only a small percentage of the vertices are eligible for an operation at one time.

- Was able to specify an RDD that specified which Triplets should be included.
Restricted Active Set

- Only a small percentage of the vertices are eligible for an operation at one time.

- Was able to specify an RDD that specified which Triplets should be included.

- Ended up not improving performance.
Restricted Active Set

• Only a small percentage of the vertices are eligible for an operation at one time.

• Was able to specify an RDD that specified which Triplets should be included.

• Ended up not improving performance.
  - Number of operations remains the same
Restricted Active Set

• Only a small percentage of the vertices are eligible for an operation at one time.

• Was able to specify an RDD that specified which Triplets should be included.

• Ended up not improving performance.
  – Number of operations remains the same
  – Could provide benefit with costly methods
Experimentation

- Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).
Experimentation

- Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).

- The scale of “big data” wasn't feasible for the scope of this project.
Experimentation

- Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).

- The scale of “big data” wasn't feasible for the scope of this project.
  - Would require distributed storage
Experimentation

- Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).

- The scale of “big data” wasn't feasible for the scope of this project.
  - Would require distributed storage
  - Cluster would need to be scaled
Experimentation

• Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).

• The scale of “big data” wasn't feasible for the scope of this project.
  - Would require distributed storage
  - Cluster would need to be scaled
  - More difficult to find truly large datasets
Experimentation

• Used Amazon Elastic MapReduce services to run on a cluster (2 c3.xlarge instances).

• The scale of “big data” wasn't feasible for the scope of this project.
  – Would require distributed storage
  – Cluster would need to be scaled
  – More difficult to find truly large datasets

• Aimed to verify correctness and observe the effects of the variations mentioned.
Datasets (.bk files)

- Single-line → Contrived graph of 500 chained vertices
  - 499 edges

- Parallel-5-5 → Contrived graph branching at factor of 5
  - 3900 edges

- Parallel-12-5 → Contrived graph branching at factor of 12
  - 271440 edges

- RMF-wide → Smallest of benchmarks obtained online.
  - 93178 edges
## Caching vs. Non-caching Results

<table>
<thead>
<tr>
<th></th>
<th>single-line (s)</th>
<th>RMF-wide 200 iter. (s)</th>
<th>parallel-5-5 (s)</th>
<th>parallel-12-5 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>750.736</td>
<td>413.838</td>
<td>16.207</td>
<td>118.728</td>
</tr>
<tr>
<td>Cache</td>
<td>522.527</td>
<td>306.654</td>
<td>13.432</td>
<td>92.042</td>
</tr>
</tbody>
</table>

Speedup: 1.44x 1.35x 1.21x 1.29x
Caching vs. Non-caching Results

<table>
<thead>
<tr>
<th></th>
<th>single-line (s)</th>
<th>RMF-wide 200 iter. (s)</th>
<th>parallel-5-5 (s)</th>
<th>parallel-12-5 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>750.736</td>
<td>413.838</td>
<td>16.207</td>
<td>118.728</td>
</tr>
<tr>
<td>Cache</td>
<td>522.527</td>
<td>306.654</td>
<td>13.432</td>
<td>92.042</td>
</tr>
</tbody>
</table>

Speedup: 1.44x 1.35x 1.21x 1.29x

- Clearly improves performance, possibly having a larger impact as the problem is scaled.
Caching vs. Non-caching Results

<table>
<thead>
<tr>
<th></th>
<th>single-line (s)</th>
<th>RMF-wide 200 iter. (s)</th>
<th>parallel-5-5 (s)</th>
<th>parallel-12-5 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>750.736</td>
<td>413.838</td>
<td>16.207</td>
<td>118.728</td>
</tr>
<tr>
<td>Cache</td>
<td>522.527</td>
<td>306.654</td>
<td>13.432</td>
<td>92.042</td>
</tr>
</tbody>
</table>

Speedup: 1.44x 1.35x 1.21x 1.29x

- Clearly improves performance, possibly having a larger impact as the problem is scaled.
- Both contrived parallel datasets complete in the same number of iterations.
Checkpointing Intervals Results

- Expectation was that more frequent checkpointing would always hurt performance.

<table>
<thead>
<tr>
<th></th>
<th>single-line (s)</th>
<th>RMF-wide 200 iter. (s)</th>
<th>parallel-5-5 (s)</th>
<th>parallel-12-5 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 iterations</td>
<td>456.626</td>
<td>287.349</td>
<td>15.505</td>
<td>94.515</td>
</tr>
<tr>
<td>25 iterations</td>
<td>427.684</td>
<td>314.942</td>
<td>13.251</td>
<td>92.612</td>
</tr>
<tr>
<td>50 iterations</td>
<td>522.527</td>
<td>341.530</td>
<td>13.432</td>
<td>92.042</td>
</tr>
</tbody>
</table>
Checkpointing Intervals Results

<table>
<thead>
<tr>
<th></th>
<th>single-line (s)</th>
<th>RMF-wide 200 iter. (s)</th>
<th>parallel-5-5 (s)</th>
<th>parallel-12-5 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 iterations</td>
<td>456.626</td>
<td>287.349</td>
<td>15.505</td>
<td>94.515</td>
</tr>
<tr>
<td>25 iterations</td>
<td>427.684</td>
<td>314.942</td>
<td>13.251</td>
<td>92.612</td>
</tr>
<tr>
<td>50 iterations</td>
<td>522.527</td>
<td>341.530</td>
<td>13.432</td>
<td>92.042</td>
</tr>
</tbody>
</table>

- Expectation was that more frequent checkpointing would always hurt performance.

- Results seem to be unpredictable, and the middle (25 iterations) option appears to be best overall.
Checkpointing Intervals Results

- Results show that iterations tend to grow longer in between checkpointing intervals.
Checkpointing Intervals Results

Results show that iterations tend to grow longer in between checkpointing intervals.

May indicate that there is some balance between the cost of checkpointing and the cost of increased lineage.
Scaling and Amazon Inconsistencies

- All of these observations must be taken with a grain of salt...
Scaling and Amazon Inconsistencies

- All of these observations must be taken with a grain of salt...

- Amazon clusters are extremely convenient, but appear to vary widely in performance.
Scaling and Amazon Inconsistencies

- All of these observations must be taken with a grain of salt...

- Amazon clusters are extremely convenient, but appear to vary widely in performance.

- Prevented any meaningful data collection in the area of scaling the cluster.
Scaling and Amazon Inconsistencies

- All of these observations must be taken with a grain of salt...

- Amazon clusters are extremely convenient, but appear to vary widely in performance.

- Prevented any meaningful data collection in the area of scaling the cluster.

“The performance of Amazon machine instances is sometimes fast, sometimes slow, and sometimes absolutely abysmal."

- Blog article “Benchmarking Amazon EC2: The wacky world of cloud performance”
Possible Future Work

- Scaling and verification of approach
Possible Future Work

- Scaling and verification of approach
  - How does the solution perform in a truly “big data” context?
Possible Future Work

• Scaling and verification of approach
  - How does the solution perform in a truly “big data” context?
  - What is the impact that scaling the cluster has on performance?
Possible Future Work

• Scaling and verification of approach
  – How does the solution perform in a truly “big data” context?
  – What is the impact that scaling the cluster has on performance?

• Algorithm optimization
Possible Future Work

• Scaling and verification of approach
  – How does the solution perform in a truly “big data” context?
  – What is the impact that scaling the cluster has on performance?

• Algorithm optimization
  – Possibility of condensing MapReduce steps
Possible Future Work

• Scaling and verification of approach
  – How does the solution perform in a truly “big data” context?
  – What is the impact that scaling the cluster has on performance?

• Algorithm optimization
  – Possibility of condensing MapReduce steps
  – Compare data structure selections
Possible Future Work

• Scaling and verification of approach
  - How does the solution perform in a truly “big data” context?
  - What is the impact that scaling the cluster has on performance?

• Algorithm optimization
  - Possibility of condensing MapReduce steps
  - Compare data structure selections
  - Explore manual uncaching
Open Questions
// Define types
type VertexPushMap = Map[(VertexId, Boolean), Int]
type EdgeData = (Int, Int)
type VertexData = (Int, Int, VertexPushMap)
type SurveyMessage = (VertexPushMap, Int)

// Initialize the graph
var activeMessages = 1
var iteration = 1

// Build graph
val vertexArray = vertexBuffer.toArray
val edgeArray = edgeBuffer.toArray
val vertexRDD: RDD[(VertexId, VertexData)] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[EdgeData]] = sc.parallelize(edgeArray)
var graph = Graph(vertexRDD, edgeRDD)
// "Surveying" MapReduce step
val eligiblePushesRDD = graph.aggregateMessages[SurveyMessage] (  
  // Map: Send message if vertex has excess
  edgeContext => {  
    // Make sure not to push from sink or source
    if (edgeContext.srcId != sinkId && edgeContext.srcId != sourceId) {  
      // If a residual edge exists from source to destination
      if (edgeContext.attr._2 > edgeContext.attr._1) {  
        // If source has an excess
        if (edgeContext.srcAttr._1 > 0) {  
          // If source has height one greater than destination
          if (edgeContext.srcAttr._2 == (edgeContext.dstAttr._2 + 1)) {  
            // Push is possible, send message to source containing push information
            val pushAmount = math.min(edgeContext.attr._2 - edgeContext.attr._1, edgeContext.srcAttr._1)
            edgeContext.sendToSrc((Map((edgeContext.dstId, true) -> pushAmount), edgeContext.dstAttr._2))  
          } else {  
            edgeContext.sendToSrc((Map(), edgeContext.dstAttr._2))  
          }  
        } else {  
          edgeContext.sendToSrc((Map(), edgeContext.dstAttr._2))  
        }  
      }  
    }  
  }  
  // Reduce: Concatenate into map of all possible pushes, keep track of relabel eligibility
  (a, b) => {  
    (a._1 ++ b._1, math.min(a._2, b._2))  
  })
  
(Repeated in other direction along the edge)
GraphX Code - “Surveying” MapReduce

// "Surveying" MapReduce step
val eligiblePushesRDD = graph.aggregateMessages[SurveyMessage] (  
  // Map: Send message if vertex has excess
  edgeContext => {
    // Make sure not to push from sink or source
    if (edgeContext.srcId != sinkId && edgeContext.srcId != sourceId) {
      // If a residual edge exists from source to destination
      if (edgeContext.attr._2 > edgeContext.attr._1) {
        // If source has an excess
        if (edgeContext.srcAttr._1 > 0) {
          // If source has height one greater than destination
          if (edgeContext.srcAttr._2 == (edgeContext.dstAttr._2 + 1)) {
            // Push is possible, send message to source containing push information
            val pushAmount = math.min(edgeContext.attr._2 - edgeContext.attr._1, edgeContext.srcAttr._1)
            edgeContext.sendToSrc((Map((edgeContext.dstId, true) -> pushAmount), edgeContext.dstAttr._2))
          } else {
            edgeContext.sendToSrc((Map(), edgeContext.dstAttr._2))
          }
        }
      }
    }
  }
)

(Repeated in other direction along the edge)

// Reduce: Concatenate into map of all possible pushes, keep track of relabel eligibility
(a, b) => {
  (a._1 ++ b._1, math.min(a._2, b._2))
}

Maps

Reduce
GraphX Code - “Surveying” Vertex Program

```
graph = graph.outerJoinVertices(eligiblePushesRDD) {
  (id: VertexId, data: VertexData, msg: Option[SurveyMessage]) => {
    // Store empty map if no messages
    if (msg.isEmpty) {
      (data._1, data._2, Map[(VertexId, Boolean), Int]())
    } else if (msg.get._2 >= data._2) {
      // Eligible for relabel
      (data._1, msg.get._2 + 1, Map[(VertexId, Boolean), Int]())
    } else {
      // Add pushes until no excess remains or pushes are exhausted
      var excess = data._1
      val selectedPushes = scala.collection.mutable.Map[(VertexId, Boolean), Int]()

      // Select pushes until flow is gone, break once no flow is remaining.
      breakable {
        msg.get._1.foreach(pushData => {
          val dstId = pushData._1._1
          val forwardPush = pushData._1._2
          val pushAmount = pushData._2
          if (excess > 0) {
            val selectedPushAmount = math.min(pushAmount, excess)
            excess -= selectedPushAmount
            selectedPushes((dstId, forwardPush)) = selectedPushAmount
          } else {
            break
          }
        })
      }

      (excess, data._2, selectedPushes.toMap)
    }
  }
}
```
graph = graph.outerJoinVertices(eligiblePushesRDD) {
  (id: VertexId, data: VertexData, msg: Option[SurveyMessage]) => {
    // Store empty map if no messages
    if (msg.isEmpty) {
      (data._1, data._2, Map[(VertexId, Boolean), Int]())
    } else if (msg.get._2 >= data._2) {
      // Eligible for relabel
      (data._1, msg.get._2 + 1, Map[(VertexId, Boolean), Int]())
    } else {
      // Add pushes until no excess remains or pushes are exhausted
      var excess = data._1
      val selectedPushes = scala.collection.mutable.Map[(VertexId, Boolean), Int]()
      
      // Select pushes until flow is gone, break once no flow is remaining.
      breakable {
        msg.get._1.foreach(pushData => {
          val dstId = pushData._1._1
          val forwardPush = pushData._1._2
          val pushAmount = pushData._2
          if (excess > 0) {
            val selectedPushAmount = math.min(pushAmount, excess)
            excess -= selectedPushAmount
            selectedPushes((dstId, forwardPush)) = selectedPushAmount
          } else {
            break
          } 
        })
      }
      (excess, data._2, selectedPushes.toMap)
    }
  }
}
val executedPushesRDD = graph.aggregateMessages[Int] (  
  // Map: Send push information to vertices that received flow  
  edgeContext => {  
    // Check if destination vertex id is in the source's push map  
    if (edgeContext.srcAttr._3.contains((edgeContext.dstId, true))) {  
      val pushAmount: Int = edgeContext.srcAttr._3((edgeContext.dstId, true))  
      edgeContext.sendToDst(pushAmount)  
    }  
    // Check if source vertex id is in the destination's push map  
    if (edgeContext.dstAttr._3.contains((edgeContext.srcId, false))) {  
      val pushAmount: Int = edgeContext.dstAttr._3((edgeContext.srcId, false))  
      edgeContext.sendToSrc(pushAmount)  
    }  
  },  
  // Reduce: Combine all incoming flow into a single total  
  (a, b) => {  
    a + b  
  }
)
GraphX Code - “Execution” MapReduce

```scala
val executedPushesRDD = graph.aggregateMessages[Int] {
  // Map: Send push information to vertices that received flow
  edgeContext => {
    // Check if destination vertex id is in the source's push map
    if (edgeContext.srcAttr._3.contains((edgeContext.dstId, true))) {
      val pushAmount: Int = edgeContext.srcAttr._3((edgeContext.dstId, true))
      edgeContext.sendToDst(pushAmount)
    }

    // Check if source vertex id is in the destinations's push map
    if (edgeContext.dstAttr._3.contains((edgeContext.srcId, false))) {
      val pushAmount: Int = edgeContext.dstAttr._3((edgeContext.srcId, false))
      edgeContext.sendToSrc(pushAmount)
    }
  },
  // Reduce: Combine all incoming flow into a single total
  (a, b) => a + b
}
```
GraphX Code - “Execution” Vertex Program

// Update excess values
graph = graph.outerJoinVertices(executedPushesRDD) {
    (id: VertexId, data: VertexData, msg: Option[Int]) => {
        // Add pushed flow to vertex
        (data._1 + msg.getOrElse(0), data._2, data._3)
    }
}