A PERFORMANCE STUDY OF AN IMPLEMENTATION OF THE PUSH-RELABEL MAXIMUM FLOW ALGORITHM IN APACHE SPARK'S GRAPHX

by

Ryan P. Langewisch

© **Copyright by Ryan P. Langewisch, 2015**

All Rights Reserved

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Computer Science).

Golden, Colorado

Date ________________________

Signed: ________________________

Ryan P. Langewisch

Signed: ________________________

Dr. Dinesh Mehta

Thesis Advisor

Golden, Colorado

Date ________________________

Signed: ________________________

Dr. Atef Elsherbeni

Professor and Interim Department Head

Department of Electrical Engineering and Computer Science

ABSTRACT

GraphX is an API for graph computation built upon Apache Spark, a fast and generalized engine for large-scale data processing in the cloud. While the popularity of Spark and GraphX is growing, the relatively young technology has yet to explore the breadth of graph problems that exist in the field. In order to examine and gain insights into the capabilities of GraphX, this thesis approaches the framework with the intention of implementing a solution to the Maximum Flow Problem, a complex graph problem without a trivial distributed approach. Specifically, the implementation is to be based on the serial Push-Relabel algorithm. An original MapReduce-based approach to the problem is presented, as well as an implementation of the approach in GraphX. In addition to the implementation, experimentation and deployment to an Amazon EC2 cluster allowed observations on caching and checkpointing intervals to be made.

TABLE OF CONTENTS

ABSTRACT . . . iii
LIST OF FIGURES . . . vi
LIST OF TABLES . . . vii
ACKNOWLEDGMENTS . . . viii
CHAPTER 1 INTRODUCTION . . . 1
CHAPTER 2 RELATED WORK . . . 4
2.1 MapReduce . . . 4
2.2 Apache Spark . . . 4
2.3 GraphX . . . 5
2.3.1 Graph Construction in GraphX . . . 6
2.3.2 Simple GraphX Example . . . 7
2.3.3 GraphX Pregel API . . . 9
2.4 Maximum-Flow Problem . . . 12
2.5 Push-Relabel Algorithm . . . 13
CHAPTER 3 IMPLEMENTATION . . . 15
3.1 Algorithm Description . . . 15
3.2 Pregel API Consideration . . . 18
3.3 GraphX Implementation . . . 19
3.3.1 Initialization . . . 19
3.3.2 *Surveying* Step . . . 21

3.3.3 *Execution* Step 27

3.4 Simple Example 31

3.5 Algorithm Correctness 39

CHAPTER 4 IMPLEMENTATION VARIATIONS 41

4.1 Checkpointing 41

4.2 Caching 42

4.3 Restricted Active Set 43

CHAPTER 5 EXPERIMENTATION 46

5.1 Datasets 47

5.2 Caching vs. Non-caching results 49

5.3 Checkpointing Intervals Results 50

5.4 Scaling and Amazon Inconsistencies 52

CHAPTER 6 POSSIBLE FUTURE WORK 54

6.1 Scaling and Verification of Approach 54

6.2 Algorithm Optimization 55

REFERENCES CITED 57

APPENDIX A - IMPLEMENTATION CODE 60

APPENDIX B - DATASET FILE TYPE (.BK) 75

APPENDIX C - GRAPH GENERATION SCRIPTS 77

LIST OF FIGURES

Figure 2.1 Simple graph construction in GraphX 6

Figure 2.2 GraphX code to find the oldest follower of each person in the graph. . . . 8

Figure 2.3 Single source shortest path solution using Pregel API 10

Figure 2.4 Example of a solution to the maximum-flow problem 12

Figure 3.1 Graph Initialization Code 19

Figure 3.2 *Surveying* MapReduce Code 22

Figure 3.3 *Surveying* Vertex Program Code 25

Figure 3.4 *Execution* Update Edges Code 27

Figure 3.5 *Execution* Update Vertices Code 29

Figure 3.6 *Execution* Vertex Program Code 30

Figure 3.7 Simple Example - Initialized Graph. 31

Figure 3.8 Simple Example - After Iteration 1. 33

Figure 3.9 Simple Example - After Iteration 2. 35

Figure 3.10 Simple Example - After Iteration 3. 36

Figure 3.11 Simple Example - After Iteration 4. 38

Figure 5.1 Information on Amazon's C3 instance type. 47

Figure 5.2 Iteration run times for cached implementation on single-line dataset with checkpointing interval of 50. 51

LIST OF TABLES

Table 5.1 Base implementation vs. Caching implementation. 49

Table 5.2 Different checkpointing intervals on caching implementation. 51

ACKNOWLEDGMENTS

I would like to thank Dr. Dinesh Mehta for his guidance and collaboration throughout the development of this thesis. Additional thanks to the other members of my thesis committee (Dr. Bo Wu, Dr. Arun Ravindran, and Dr. Bharat Joshi) for their willingness to give of their time to provide feedback, as well as Ankur Dave for sharing some of his expertise on GraphX.

CHAPTER 1
INTRODUCTION

The rapid development of computer technology in recent years has not only improved the efficiency and capability of both hardware and software, but has also introduced the concept of "big data." While no single definition of "big data" is universally accepted, the term emphasizes the scale of the data that is being produced. This massive increase in size changes questions such as, "can this problem be solved?" into questions like, "can this problem be solved quickly?" or "how can the data be stored?" No longer is a solution that guarantees correctness sufficient; time and resources are just as critical. Certainly a large amount of optimization in manipulating "big data" can come in the form of more efficient algorithms that use novel approaches to achieve smaller time complexities. But the optimization that is possible from an algorithmic approach is bounded both by lower-bound proofs for a particular problem, as well as the physical limitations that bound the speed of a single processor. Fundamentally, a signal within a computer cannot travel faster than the speed of light [1]. So what happens if a "big data" problem takes unreasonably long to solve, even with optimal algorithmic methods implemented on cutting edge technology? The answer is simple: parallel computing.

The concept here is not difficult to understand. Two computers should be able to complete a task more quickly than one computer, just as splitting a task between two people should yield faster results than working alone. But using the word "simple" to describe this approach is misleading. Not all problems are inherently parallelizable, and even those that are may require a significantly different approach than the known solutions that have been developed in a serial environment. This means that countless problems that have been "solved" in computing must be reevaluated as the data scales and parallel computing is the only reasonable solution. Developing parallel solutions can be a very difficult task, largely

because humans do not naturally solve problems in parallel. There is an extra layer of complexity that parallelism puts in play that makes a programmer's job even more challenging. On top of that, there are issues that arise simply by multiplying the amount of hardware that is in play. Data transfer can be limited by network bandwidth, and the sheer number of individual processors makes it more likely for a machine to fail. Instead of just considering how to correctly implement an algorithm, serious consideration must be given to whether it requires too much data transfer on the network and how to ensure correctness even if a node fails.

Despite all these complications, parallel computing is quickly becoming the only viable way to approach many "big data" problems. As a result, it is natural that a lot of effort has gone into developing technologies that support problem solving on distributed systems. Google introduced the programming model known as "MapReduce" to provide an easy to use parallel paradigm that hides many of the complex details such as fault-tolerance and data distribution [2]. Since its release, MapReduce has been used as the basis for several parallel programming frameworks that aim to make the approach more accessible. Apache Hadoop [3] quickly gained popularity as an open-source framework that provided MapReduce functionality, as well as HDFS[1] [4], a highly fault-tolerant distributed file system. While Hadoop has proven very effective for batch processing, it shows some limitations when dealing with repeated access of data or operations independent of the two-step MapReduce paradigm. Apache Spark, originally developed at the University of California in Berkeley, addresses these limitations by utilizing new data abstractions known as resilient distributed datasets, or RDDs [5]. This data storage format allows Spark to keep data in memory, greatly benefiting the performance of iterative algorithms that have a cyclic data flow [6]. The implementation of RDDs in Spark also allows fast response times to data queries, providing the flexibility to work with the data without utilizing MapReduce. Spark has rapidly gained popularity, already being adopted and used in production by large companies such as Amazon, Yahoo!,

[1]Hadoop Distributed File System

and eBay [7]. As of the summer of 2015, IBM has also dedicated huge amounts of support to Apache Spark calling it "potentially the most significant open source project of the next decade." [8]

The rapid growth in the popularity of Spark, paired with the strong correlation between "big data" and graph analytics, has resulted in efforts to develop optimized graph processing functionality in Spark. GraphX [9], a thin layer on top of the Spark framework, provides graph processing capabilities competitive with specialized graph processing systems (e.g. Pregel [10]) while maintaining the wide range of computing options supported by Spark. Spark's ability to increase the performance of iterative algorithms complements the fact that much of graph processing is iterative in nature. For example, PageRank [11] relies on iterative neighborhood aggregation for each node until the process converges to a stable solution (or it reaches a specified number of iterations). Spark's GraphX allows PageRank to be implemented succinctly in a high-level programming language, taking advantage of the in-memory data storage, which ultimately leads to improved performance.

While GraphX has been shown to be effective in implementing several fundamental graph processing problems, many of the problems (e.g. PageRank [11]) have a natural compatibility with parallelization, lacking interference among operations involving shared neighboring nodes. In this thesis, we are proposing the analysis of a more complex, yet fundamental graph problem, the maximum flow problem, in the context of a GraphX implementation. Specifically, we are looking to implement an approach based on the push-relabel maximum flow algorithm in GraphX. In the remainder of this thesis, I will give more background information and detail on the technologies being used, the push-relabel algorithm and our parallel implementation, and various observations made based on experimentation.

CHAPTER 2

RELATED WORK

In order to have a reasonable understanding of the proposed problem, it is important to understand the technologies being used, as well as the algorithmic approach that we selected as the basis for our parallel implementation.

2.1 MapReduce

Much of recent work in the area of cluster computing and parallel computation builds on the approach introduced by MapReduce [2]. In brief, MapReduce is a two-step paradigm for solving problems over distributed systems. First, the data is represented as key-value pairs, that are spread over a distributed system. Then a *Map* method is applied to all of the key-value pairs. This is simply a user-defined function that converts each single key-value pair into any number of new key-value pairs. The important distinction here is that it does not matter where the data is located in the system because each map operation is completely independent, and therefore inherently parallelizable. Once the key-value pairs have been mapped, the resulting pairs are then sent to "reducer" nodes based on their keys. These nodes then apply some user-defined *Reduce* method that operates on pairs with the same key. After the reduction, the process may be finished or require more MapReduce iterations depending on the problem. MapReduce provides a highly fault-tolerant and efficient approach to solving problems in parallel, and many problems can be adjusted to be solved using the paradigm.

2.2 Apache Spark

Apache Spark is a fast and general engine for large-scale data processing [12]. The motivation for Spark's development stemmed from the fact that while MapReduce and its various implementations have been effective in cluster computing, the existing systems do not have

efficient support for applications where the working data set is reused across parallel operations [6]. Many algorithms involve iterative operations on a data set, providing opportunity for huge performance increase if the data could be retained in memory between iterations. Spark addresses this observation by introducing an abstraction known as a resilient distributed dataset (RDD) [6]. RDDs represent a read-only collection of objects partitioned over a set of machines, and achieve fault tolerance by using the concept of *lineage* [6]. In other words, even if a partition of an RDD is lost, there is enough information about the derivation of the RDD to recreate the missing partition. The main advantage that RDDs provide is that they, in addition to retaining the fault-tolerance of the MapReduce paradigm, can be cached in each parallel core's memory, preventing repeated disk access between iterations of parallel operations [5].

Spark is implemented in Scala [13], and provides an implementation of RDDs, along with parallel operations that can be performed on RDDs. Evaluation of the Spark platform boasts results upwards of 100x faster than Hadoop MapReduce while maintaining generality and accessibility [12]. Whereas Hadoop primarily focused on batch processes, the utilization of RDDs in Spark allows the data to be queried in real-time, allowing data analysis and manipulation that goes beyond the two-step MapReduce model. Interactive queries in Spark can yield sub-second response times on data sets as large as 39GB [6]. Spark's flexibility in data manipulation, as well as increased performance as a result of RDDs, makes it a very attractive option for distributed computing and contributes to its rapid rise in popularity in the last two years.

2.3 GraphX

GraphX is Spark's API for graphs and graph-parallel computation [14]. GraphX provides primitives that efficiently express graph computation, while retaining the computation flexibility of Spark's data-parallel framework [9]. These primitives not only make graph computation in Spark more user friendly and efficient, but even allow graph-specific abstractions such as Pregel [10] to be implemented in less than 20 lines of code [9]. In this section, we

will examine some of the basic syntax and approaches used for solving problems in GraphX.

2.3.1 Graph Construction in GraphX

As emphasized in previous sections, Spark is built upon the use of RDDs for storing and manipulating data. It makes sense then that the process of constructing a graph in GraphX involves converting the dataset into corresponding RDDs. If constructing a graph manually, the process may involve creating the data in Scala arrays, and then using Spark's *parallelize* method to convert the array data into an RDD. Figure 2.1 shows initial graph data being declared in Scala arrays, the conversion to RDDs using the parallelize method, and then the creation of the property graph using the constructed vertex and edge RDDs [15].

```
// Initialize vertex and edge arrays
val vertexArray = Array(
  (1L, ("Alice", 28)),
  (2L, ("Bob", 27)),
  (3L, ("Charlie", 65))
)
val edgeArray = Array(
  Edge(2L, 1L, 7),
  Edge(2L, 3L, 2),
  Edge(3L, 2L, 4)
)

// Convert arrays into RDDs
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

// Create the property graph
val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

Figure 2.1: Simple graph construction in GraphX

The Scala syntax used by Spark and GraphX can be a little confusing to those unfamiliar with the language, so there are a couple of points to clarify. First, the capital L tagged on the numbers when creating the vertex and edge arrays is simply explicitly casting the value as the `Long` type, rather than the `Int` type. This value is the ID of each vertex, which is referenced when creating the edges between vertices. It is also worth noting that it is common to specify the type signature of a value in Scala on assignment. For example, consider the line that converts the vertex array to an RDD:

```
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
```

The `RDD[(Long, (String, Int))]` portion of the line is simply specifying the exact type signature for the value. In this case, we are assigning an RDD where each element is an array with two values: an ID of type Long, and another array that specifies the person's name and age as a *String* and *Int*.

While constructing a graph manually can be useful for testing purposes and visualizing small manageable examples, it would not be the approach to use in any realistic situation. When I discuss evaluation later in the thesis, I will explain how we selected or generated datasets in order to run various tests. It should be noted that the primary goal here was not to show our implementation to be competitive (though that would have been a welcome side effect), but rather to analyze how a complex and different style of algorithm might be implemented in GraphX and what complications that may introduce.

2.3.2 Simple GraphX Example

In order to understand how one might use GraphX to operate on a data set, it may be most useful to jump into an example. Consider if we wanted to find the oldest follower of each vertex in the simple graph constructed in the previous section. To accomplish this, we can use the `aggregateMessages` method. This method operates on the Triplets[2] in the map, which are the fundamental parallel building blocks used by GraphX. A Triplet is simply an edge represented with three parts: the source vertex, the edge, and the destination vertex. The `aggregateMessages` method uses the MapReduce paradigm to apply a *Map* and *Reduce* method to every Triplet in the graph. The *Map* method generates messages targeting either the source or destination vertex of the Triplet, and then the *Reduce* method takes all of the incoming messages to a vertex and condenses them to a single message. Using this method to find each vertex's oldest follower might look like the code shown in Figure 2.2:

[2] A Triplet consists of a source vertex, an edge, and a destination vertex, along with their associated data. It should be noted that graphs in GraphX are directed graphs, and the ordering of this Triplet reflects the direction of the edge. One way to visualize a Triplet might be a stick with a circle on one end, and a triangle on the other. The circle is the source node, the stick is the edge, and the triangle is the destination node. In other words, a Triplet contains all the information about a directed edge, and a directed graph could be represented solely as a set of Triplets.

```
// Find the oldest follower for each user
val oldestFollower: VertexRDD[(String, Int)] = graph.aggregateMessages[(String, Int)](
  // For each edge send a message to the destination vertex with the attribute of the source vertex
  edge => edge.sentToDst((edge.srcAttr.name, edge.srcAttr.age)),
  // To combine messages, take the message for the older follower
  (a, b) => if (a._2 > b._2) a else b
)
```

Figure 2.2: GraphX code to find the oldest follower of each person in the graph. [15]

It is important to recognize that the `=>` operator in Scala is specifying a function declaration. The left side of the `=>` operator specifies the function arguments, and the right side specifies the function body. With the function declaration `edge =>` $\{\ldots\}$, `edge` is the function parameter (in this case an EdgeContext[3], as defined by the `aggregateMessages` type signature) and it uses that parameter to generate any messages. In the example above, the `aggregateMessages` method is being passed two functions as arguments, a map function and a reduce function respectively. These functions could have just as easily been declared before the call to `aggregateMessages`, and then passed in as values.

In this example, the map function is defined as:

```
edge => edge.sentToDst((edge.srcAttr.name, edge.srcAttr.age)),
```

For every edge in the graph, this map function will send a single message to the destination vertex (the "end" of the directed edge). It should be noted that this map method can send any number of messages to both the source and destination vertices, but in this case only a single message is required. Each of these messages contains an array with both the name and age attributes of the source vertex. Conceptually, this means that each vertex is going to receive a message for every incoming edge from another vertex. The *Reduce* function is then the following[4]:

```
(a, b) => if (a._2 > b._2) a else b
```

The job of the *Reduce* method is to take all key-value pairs with the same key and combine them into a single key-value pair. The method itself is written to handle only two

[3]An EdgeContext is essentially a Triplet with additional messaging functions
[4]The `._2` notation is used to access elements of Scala tuples. `._2` accesses the second element in the tuple.

messages at a time, but it will be repeated until all of the messages have collapsed into a single message. In this case, we are only concerned with the *oldest* follower, so the reduce method simply looks at the "age" attribute contained in the messages, and keeps the larger age. Applying this over all messages sent to that vertex will result in the largest age of all incoming edges, which is the solution to our problem. This is a simple example, but gives insight into the GraphX syntax and how problems can be solved using the MapReduce approach.

2.3.3 GraphX Pregel API

GraphX provides a Pregel [10] operator that they recommend for algorithms with an iterative structure [16]. The Pregel operator requires three functions to be defined:

- A "Vertex Program" method that specifies what each vertex should do with the data it receives as a result of each MapReduce step.
- A "Send Message" method that maps each Triplet into messages intended for either the source or destination vertex of the Triplet.
- A "Merge Message" method that reduces all messages to a vertex into a single value (note that the "single value" could in fact be a collection that holds multiple values).

The Pregel operator essentially applies the "Send Message" function to all Triplets in the graph in order to generate messages, uses the "Merge Message" method to reduce all messages destined for the same vertex, and then applies the "Vertex Program" at each vertex using the reduced message data. It repeats this process until an iteration yields no new messages, at which point it returns.

Figure 2.3 shows an example of how the Pregel operator could be used to solve the single source shortest path problem:

We will break down this example to understand the details of how the problem is being solved. First, the `mapVertices` method is used to set the values of each vertex to infinity,

```
// Initialize the graph such that all vertices except the root have distance infinity.
val sourceId: VertexId = 0
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // Merge Message
)
```

Figure 2.3: Single source shortest path solution using Pregel API [16]

except for the source vertex, which is set to zero. This is accomplished by using a simple conditional inside of the mapping function[5]. Then the Pregel operation is used on the resulting graph. One may notice that the Pregel operator is passed two separate lists of arguments. The first list, `(Double.PositiveInfinity)`, contains configuration parameters, while the second list (the remainder of the code) contains the three user-defined functions that have been described. In this case, `Double.PositiveInfinity` is being set as the initial message for each vertex. The meat of the operation comes with the three argument functions. The first function defined is the "Vertex Program" function[6]:

```
(id, dist, newDist) => math.min(dist, newDist)
```

The general arguments for this method are the vertex ID, the current vertex value, and the incoming value that results from the generation and reduction of messages to that vertex. In this case, the value represents the distance of the shortest path that has been found. The "Vertex Program" function simply always takes the smaller value, as that will equate to the shorter path. Next, the "Send Message" method is defined:

[5]The underscore used in the mapping function parameters is a Scala symbol indicating that a portion of the argument information is not required. In this case, `mapVertices` naturally passes the vertex ID and vertex data as a tuple, but since the vertex data is not needed by the mapping function, it can be optimized by excluding it through use of an underscore.

[6]It may seem out of order for the Vertex Program to be defined first, but it is due to the fact that the Vertex Program is actually the first function executed inside the main loop, applying any predefined messages that are used to initialize the graph.

```
triplet => {  // Send Message
  if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
    Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
  } else {
    Iterator.empty
  }
}
```

This function takes a Triplet, and returns zero or more messages. The Scala *Iterator* type is used to facilitate the variable number of messages. For this problem, the method checks to see if the value of the source vertex (`triplet.srcAttr`), plus the value of the edge (`triplet.attr`), is less than the current value of the destination vertex (`triplet.dstAttr`). If so, that means that a shorter path has been found that utilizes the source vertex and edge of the Triplet. In this case, we want to send a message to the destination vertex so that it knows to update its value. If the combined path is not shorter, then we simply return an empty Iterator which will yield no messages. Finally, the "Merge Message" method is defined:

```
(a,b) => math.min(a,b)
```

This method simply takes two incoming messages, and combines them to ensure that only one value is ultimately sent to the vertex program. For this problem, we want to take the smaller of the two values, as each message represents a possible shorter path. This method will come into play if on the same iteration, multiple neighbors of a vertex provide shorter paths than the vertex's current path.

These functions will be repeatedly applied until the graph converges to a state where no new messages are being produced. This means that no vertex in the graph is capable of finding a shorter path. At this point, the problem is solved and the `sssp` value will contain the desired results.

2.4 Maximum-Flow Problem

The maximum-flow problem involves finding the largest possible flow from a single source to a single sink in a directed graph with edges of limited capacity. Figure 2.4 gives an example instance of this problem, with s denoting the source, and t denoting the sink.

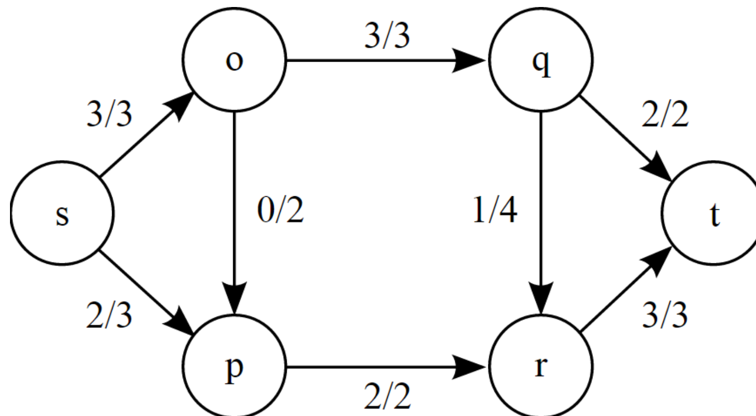


Figure 2.4: Example of a solution to the maximum-flow problem [17].

More formally, let $G = (V, E)$ be a directed graph where V is a set of vertices and E is a set of edges. Additionally, G is a flow network, with a source vertex s and a sink vertex t. A positive capacity is defined as $c(v, w)$ for each edge (v, w). A flow f on G is a function on each pair of vertices in V that satisfies the following constraints [18]:

1. Capacity Constraint: $f(v, w) \leq c(v, w)$ for all $(v, w) \in V \times V$
2. Antisymmetry Constraint: $f(v, w) = -f(w, v)$ for all $(v, w) \in V \times V$
3. Flow Conservation Constraint: $\sum_{u \in V} f(u, v) = 0$ for all $v \in V - \{s, t\}$

The value of a flow f is the net flow into the sink [18]:

$$|f| = \sum_{v \in V} f(v, t)$$

The *maximum flow* is the flow that has its value maximized.

2.5 Push-Relabel Algorithm

The Push-Relabel algorithm [18] is one of several approaches to solving the maximum-flow problem. We are interested in this approach in particular because the operations have a more inherently parallel structure than other approaches, such as the Ford-Fulkerson algorithm [19] which relies on iteratively finding augmenting paths in the residual graph[7]. The Push-Relabel algorithm has been selected for parallelized approaches in the past, as found in the papers *A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic*[20] and *An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Non-blocking Global Relabeling Heuristic*[21], supporting that the algorithm is a reasonable choice for a distributed approach. Instead of satisfying the flow conservation constraint throughout its execution, the Push-Relabel algorithm uses the concept of a *preflow*. The *preflow* pushes a maximum amount of flow out from the source, without worrying if the flow is all capable of reaching the sink. The algorithm then pushes as much of this preflow as possible towards the sink. Once no more of the excess flow can reach the sink, it pushes the remaining excess back to the source. Once the preflow becomes a valid flow that satisfies the conservation of flow at every vertex, the maximum flow has been found.

To accomplish this, the algorithm employs a vertex labeling convention that indicates whether a vertex is capable of pushing excess to another vertex. Conceptually, the label of a vertex can be thought of as a topographical height, and excess always flows from a higher vertex to a lower vertex. On each iteration, a vertex can either push excess along an edge, or relabel its own labeling value. A push is only possible if a vertex with excess flow has a label exactly one greater than a neighboring vertex. If a vertex with excess is unable to push, then it is eligible for a relabel. A relabel is possible if a vertex with excess has a label that is less than or equal to all of its neighboring vertices' labels. If true, the vertex relabels

[7] The residual graph is identical to the original graph, except that each edge capacity is defined as the original edge capacity minus the current flow on that edge. Any edges with zero capacity are removed from the residual graph.

itself to be exactly one greater than the smallest label of vertices connected by a residual edge. This puts the vertex in a position to push during the next iteration. The algorithm is initiated by setting the source vertex's label to n (the number of vertices) and pushing a maximum preflow out from the source. As the excess is pushed through the network, the vertices receiving the preflow will subsequently relabel themselves in order to push the excess further towards the sink. At some point, there will be no available path to the sink (no residual edges), and the relabeling will cause the excess to be pushed backwards until it eventually reaches the source. When there is no longer excess flow in the graph, the result is in fact the maximum flow.

The Push-Relabel algorithm shows potential for parallelism due to the fact that push and relabel operations in different areas of the graph can be executed simultaneously without any dependence on each other. Compare this with the Ford-Fulkerson [19] approach that relies on finding augmenting paths. An augmenting path is simply a path from the source to the sink that has the capacity for increased flow. Because an augmenting path can arbitrarily involve any number of vertices in the graph, the approach is not very naturally parallelizable. Pushing more flow along an augmenting path changes the graph, and as a result, it is not possible to search for additional augmenting paths until the current iteration has been resolved. This is a very sequential approach, and highlights why the Push-Relabel algorithm appears to be a better candidate for our needs.

CHAPTER 3
IMPLEMENTATION

In the serial implementation of the Push-Relabel algorithm, there may be many possible valid operations at any given time. For example, several vertices in different parts of the graph could be eligible to push flow or relabel during the same execution step. In the single-threaded algorithm, one of these operations is picked arbitrarily in each step, as there is no necessary ordering of operations to ensure correctness. The intuition with a parallel adaptation of this algorithm is to execute *all* possible operations in each step. On large, branching graphs, there is the potential for the number of parallel operations to be extremely high, presenting an opportunity for increased performance by a distributed approach.

3.1 Algorithm Description

Conceptually, the challenge presented was to convert the Push-Relabel algorithm into a structure of iterative MapReduce steps. When trying to construct a MapReduce approach, I found it useful to think from the perspective of one of the single edges that will have the map function applied to it. Each edge needs to be able to answer the question, "what are the possible operations that I have available this step?" and then "which of these operations should be executed?" The detail that complicates making these decisions is the fact that multiple outgoing edges from the same vertex share the same reservoir of excess flow. For example, consider a vertex with an excess of 10 that has three outgoing edges, each with a capacity of 5. When considering what pushes are possible, all three edges would be valid pushes. But if all three edges choose to execute a push, that requires an excess of 15, which is not available from the source vertex. How might the algorithm select pushes, ensuring that the operations do not exceed the available excess?

My first intuition (in an attempt to fit the iterative MapReduce model used by the Pregel API) was to have each edge send a message if a push was possible during the mapping step,

and then use the reduce step to select the messages one by one and keep track of the remaining excess. This approach almost works, but for each push, both the source vertex and the destination vertex need to be updated. The problem is that once the possible pushes are reduced at the source vertex, there is no means to send a message to the destination vertex to update the excess value according to the push. On the other hand, we cannot send a push message to the destination vertex during the mapping phase because it doesn't know if that particular push will be chosen. This creates a sort of "chicken and egg" scenario. In order to know which pushes will be executed, we need to wait until each vertex reduces the operations based on its limited excess. But to actually update values based on the push, we would need to use the map step to send messages to both vertices involved. For this reason, I decided that a solution with a single MapReduce step was not feasible without significant modification.

These insights led to the approach that I pursued in my implementation. The primary problem with my initial approach was that there was no way to both select valid operations and also execute them on the graph in a single MapReduce step. So what if the first MapReduce step simply focused on picking operations and storing the information in the graph, and then a second MapReduce step could actually update the graph? Following this line of thinking, I developed an approach that involves two distinct steps within each iteration: a *Surveying* step that selects a set of operations, and then an *Execution* step that applies those operations and updates the graph. Once operations are selected during the *Surveying* step, the information about the selected pushes is stored in the vertices of the graph. Then in the *Execution* step, the Triplets are able to use the information embedded at each vertex to update all of the graph values accordingly.

In addition to dealing with push operations, it is also necessary that a vertex knows if it is eligible for a relabel. Since the goal of the initial *Surveying* step is to get a list of all possible operations at each vertex, I needed to find a way to package the push and relabel information together in the messaging. My solution was for each generated message to be a

tuple, with one element corresponding to push operations, and the other corresponding to relabel operations. While the push portion of the message included information about the destination vertex ID, push amount, and push direction, the relabel portion simply tracked an integer of the lowest neighboring height label. Since a relabel is only possible if the vertex has a height label smaller than all of its neighbors, knowing the smallest neighboring height label is sufficient to validate the operation.

All of these steps will be clarified by an example later in this chapter, but for now the responsibilities of the two main iteration steps can be summarized as follows:

- Surveying Step
  - *Map* - For each edge, send a message to either vertex that has excess (could be one vertex, both, or neither). The message includes information for a legal push operation if it exists, as well as the height label of the neighboring vertex. This step checks for a push in either direction along the edge, as there could be a residual edge in either direction. Note that there will never be two simultaneous pushes in opposite directions on the same edge, as the prerequisite for having a source height label of one greater than the destination height label cannot be satisfied in both directions.
  - *Reduce* - For each pair of messages, concatenate the possible pushes into a single collection, and take the smaller of the two height labels.
  - *Vertex Program* - Select possible pushes from the messaged collection one by one until no excess is remaining. Store the information about the selected pushes in the vertex data, and discard the unselected operations. If no possible pushes exist, and the smallest neighboring height label is greater than that of the vertex, mark the vertex for a relabel[8].

[8] Note that the vertex program is only executed on vertices that received at least one message. Since messages are only sent if a vertex has excess, this prevents vertices with no excess from relabeling.

- Execution Step
  - *Map* - For each edge, check the vertex data at each vertex to see if a push was selected along that edge. If so, send a message to each vertex with the information to modify its excess. Again, both the source vertex and destination vertex of the edge must be checked as there could be a push in either direction along the edge. The vertex that contains the applicable data becomes the "source" for purposes of the push, whether it is the actual source of the edge or not.
  - *Reduce* - Sum the modifications to excess from each message.
  - *Vertex Program* - Apply the aggregate excess modification to the vertex data.

It should be noted that the `aggregateMessages` method in GraphX only allows for messages to either vertex in a Triplet. In addition to updating the vertices on a push, we also need to update the values of the edge to accurately represent the state of the residual graph. In order to accomplish this, an additional MapReduce step is used specifically for the edges. This step involves the same process of using the push information embedded at the vertices to determine if the edge should be updated. I will detail this more in section 3.3, but it is worth clarifying that the *Execution* step technically contains two MapReduce operations: one to update the vertices, and one to update the edges.

The *Surveying* and *Execution* steps are repeated continuously until no messages are generated by the *Surveying* step. This means that no internal vertices contain any excess, which is the terminating condition for the Push-Relabel algorithm.

3.2 Pregel API Consideration

When considering how to implement this approach in GraphX, it initially seemed to make sense to use the Pregel API. The documentation of GraphX recommends that the Pregel API is used for any iterative computation, as it handles some of the behind the scenes details of caching and persisting RDDs. As explained in section 2.3.3, the parameters for

the Pregel function call include three core user functions: a "Vertex Program" method, a "Send Message" method, and a "Merge Message" method. It then iteratively repeats these functions until no messages are generated. As explained in the description of my approach in section 3.1, the proposed solution requires two separate sets of these three functions that are executed on each iteration. As a result, there is no way to implement the approach using the Pregel API. Since the internal structure of the Pregel function call still very closely matches what I wanted to accomplish, I referenced and extended the source code of the Pregel class to create a version that had two distinct MapReduce cycles per iteration. The implementation of the Pregel API at the time of this work also seemed to lack any built-in checkpointing functionality, which (as I will explain in section 4.1) ends up being essential for problems requiring large numbers of iterations.

3.3 GraphX Implementation

In this section I will walk through my implementation of the proposed algorithm in GraphX. Certain details of the implementation that are not directly tied to the algorithmic approach may be omitted in this section, but the full code excerpts are made available in appendix A.

3.3.1 Initialization

The first part of the implementation initializes the graph and other variables before it enters into the main execution loop, as shown in Figure 3.1.

```
// 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)
```

Figure 3.1: Graph Initialization Code

`activeMessages` serves as the loop condition, and is initialized to a trivial value of 1 to allow entrance to the `while` loop. The `iteration` variable is simply used for tracking the number of iterations and providing that information in the output. The `iteration` variable will also come into play when I talk about checkpointing in section 4.1. At this point, all of the vertex and edge information is contained in `ArrayBuffer` variables that were used to read in data from an external file. A description of the file formats used has been provided in appendix B and the code for parsing them into an `ArrayBuffer` is included in the full code in appendix A. The first step in initializing the graph is to convert both the vertex and edge arrays to RDDs. This is done using the `parallelize` method provided by the `SparkContext` class. You may notice that the RDD type declarations contain the types `VertexData` and `EdgeData`. These are two user defined types that are declared earlier as follows:

```
type VertexPushMap = Map[(VertexId, Boolean), Int]
type EdgeData = (Int, Int)
type VertexData = (Int, Int, VertexPushMap)
type SurveyMessage = (VertexPushMap, Int)
```

These types are non-trivial, so I will break them down to explain their role in the algorithm.

- **VertexPushMap** - This is the type used to store push information in each vertex after the *Surveying* step, as well as the "push" portion of the messages that are generated during the *Surveying* step. It is a map in which each key is a tuple containing the ID of the destination vertex in the push, and a boolean flag that specifies whether the push is in the direction of the directed edge[9]. Each value in the map is the amount of excess being pushed. So for example, the map may contain the following key-value pair:

  `(1L, true) -> 5`

[9]The direction of the push is important due to the fact that there could be two directed edges between the same two vertices, but in opposite directions. If the direction of the push is not specified, both edges will execute the push in the *Execution* step, adding erroneous flow into the system.

This represents an excess of 5 being pushed in the direction of the edge towards the vertex with an ID of 1L.

- **EdgeData** - This type simply represents the flow on an edge as a tuple. The first `Int` is the active flow, and the second `Int` is the edge capacity. This data encapsulates all the information of the residual graph as the residual capacity in the opposite direction is simply equal to the current flow in the direction of the edge, while the capacity of the residual edge in the direction of the edge is the capacity minus the current flow.

- **VertexData** - This type represents the data at each vertex as a tuple. The first `Int` is the excess flow and the second `Int` is the current height label. The third element of the tuple is the `VertexPushMap` type that was defined earlier, and is where the selected pushes are stored in between the *Surveying* and *Execution* steps.

- **SurveyMessage** - This is the type used for the messaging in the *Surveying* step. The tuple simply includes the push information as a `VertexPushMap`, and the neighboring height label as an `Int`.

Defining these types explicitly proves to be extremely valuable during development in GraphX, as it allows you to change the types of edge data, vertex data, message data, etc. without having to update the scattered references throughout the rest of the code.

Once the RDDs are formed, we can instantiate the graph with the two RDDs as parameters. At this point, the graph has been created and we are ready to begin the core execution loop. The rest of the code is nested inside of a simple while loop that terminates once no new messages are generated:

```
while (activeMessages > 0) { ... }
```

3.3.2 *Surveying* Step

The first step within the main loop is the *Surveying* MapReduce operation as shown in Figure 3.2. This portion of code is actually simpler than it may appear at first. The bulk of

```
// "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))
          }
        }
      }
    }

    // Make sure not to push from sink or source
    if (edgeContext.dstId != sinkId && edgeContext.dstId != sourceId) {
      // If a residual edge exists from destination to source
      if (edgeContext.attr._1 > 0) {
        // If destination has an excess
        if (edgeContext.dstAttr._1 > 0) {
          // If destination has height one greater than source
          if (edgeContext.dstAttr._2 == (edgeContext.srcAttr._2 + 1)) {
            // Push is possible, send message to destination containing push information
            val pushAmount = math.min(edgeContext.attr._1, edgeContext.dstAttr._1)
            edgeContext.sendToDst((Map((edgeContext.srcId, false) -> pushAmount), edgeContext.srcAttr._2))
          } else {
            edgeContext.sendToDst((Map(), edgeContext.srcAttr._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))
  }
)
```

Figure 3.2: *Surveying* MapReduce Code

the map method is simply conditionals that narrow down whether there is an eligible push on that edge. You will then notice that the entire section of conditional checks is repeated a second time. This is because it needs to check whether a push is possible in either direction along the edge. For a push to be possible, the following criteria must be met:

- The push cannot come out of the source or the sink. The algorithm begins with all of the edges out of the source saturated completely, so there is never another time that either the source or sink should push excess outward.
- A residual edge must exist in the direction of the push. This simply means the edge has capacity to increase flow in the direction of the push[10].
- The source vertex of the push must currently house excess flow. If this condition is met, a message is guaranteed to be sent as any vertex with excess needs to either push or relabel.
- The height label of the source vertex must be exactly one greater than the height label of the destination vertex. If this is true, the push information is sent as a single element of a map in the message. Otherwise, an empty Map is sent to indicate there is no legal push.

If all of these criteria are met, a message will be sent to the vertex that will serve as the source in the context of the push. In addition to including the information for possible push operations in the generated messages, the height label of the neighboring vertex is also included to account for possible relabel operations. In the cases where possible pushes do exist, the height label will be irrelevant, but it is packaged in the message either way simply to keep a consistent message data type.

The reduce method is very simple. It concatenates[11] the Maps from each of the messages (meaning that the empty Maps from messages without eligible pushes effectively disappear),

[10]Keep in mind that "increasing" the flow in the opposite direction of the edge actually means reducing the flow on that edge.
[11]The concatenation operation for Maps in Scala is `++`.

and then takes the minimum of the two height label values. Once all edges have been mapped and reduced in this manner, `eligiblePushesRDD` will contain every vertex that received at least one message. Each vertex will have a single message containing a Map of all possible pushes with the corresponding information, as well as the value of the lowest height label of all of its neighboring vertices.

The message information in `eligiblePushesRDD` is then used to select push operations and store the push information in each vertex. The code for this vertex program is shown in Figure 3.3. This is accomplished using the `outerJoinVertices` method, which joins the vertices in the graph with an external RDD using a user-defined mapping function to merge the results. In this case, the mapping method needs to view the message data held in the RDD and appropriately update values and store data in the vertex. For any given vertex, there are three possible options. Either no message was received at that vertex (for all vertices without excess), the message indicates that the vertex is eligible for a relabel, or the message indicates that the vertex is eligible to push. As can be seen in the code sample in Figure 3.3, the map method within `outerJoinVertices` contains three parameters: `id`, `data`, and `msg`. `id` is simply the ID of the vertex that is being mapped. `data` contains the data that is currently stored in the vertex. `msg` contains the message from `eligiblePushesRDD` if a message exists for that vertex. The Scala `Option` type simply accounts for the fact that a vertex may not have an associated message. The map method needs to return the new data to be stored at that vertex, matching the `VertexData` type that was defined earlier. Note that in Scala, the `return` keyword is implied and the last expression in the method is taken to be the return value.

First, the method checks if the `msg` parameter is empty, indicating that neither a push nor a relabel is available. In this case, the original vertex data for excess and height label from the `data` parameter are stored back in the vertex, along with an empty map of eligible pushes. It is important that an empty map is stored as opposed to the original value from the `data` parameter because otherwise pushes from a previous iteration could be executed again

```
// Store results in graph, only keeping as much as excess can support
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)
    }
  }
}
```

Figure 3.3: *Surveying* Vertex Program Code

erroneously. If the message is not empty, the method then checks to see if the height label in the message is greater than or equal to the height label of the vertex. Remember that the height label in the message represents the *lowest* height label of all the vertex's neighbors. If the vertex's height label is less than or equal to this value, then it is eligible for a relabel. The data is then stored just as in the case with an empty message, except the height label is set to one greater than the message's height label. If the message is neither empty nor indicating eligibility for a relabel, then the vertex needs to select pushes to be executed. The basic structure of this process is to first create an empty Map, and then iterate over the message's pushes as long as there is excess remaining. For each push selected, the excess value is decreased accordingly. It is likely that the excess will not divide evenly among the push operations, and this is handled by using the minimum of the push capacity and the remaining vertex excess. For example, if there is only an excess of 5 left at the vertex and a push with a residual capacity of 10 is available, the push will still be executed, but only with a flow of 5.

You will also notice that the portion of code that iterates over the `VertexPushMap` is surrounded by a `breakable` block. In Scala, there is no `break` keyword as is commonly found in non-functional languages. This means that to achieve the same effect, you need to restructure your code to use recursion or another similar technique. In more recent versions of Scala, a `break` construct was added as an external library and can be used given that the code is surrounded by the `breakable` block. In my implementation, this was the simplest way to achieve my goal. The reasoning is that if all of the excess flow has been distributed, it is a waste of time to continue iterating through the rest of the push operations in the map. Rather, we can just break out of the loop and return the vertex data right away.

After the *Surveying* step is complete, the graph will have subtracted excess from every vertex assigned to push, as well as stored the push information so that the destination vertices can add the pushed excess during the *Execution* step. At this point, the graph is in a somewhat "invalid" state, in that flow has been subtracted, without adding it to the

destination vertices or updating the values of the residual edges. That said, correctness will be ensured during the *Execution* step because for every subtraction of excess, a destination vertex will read the source vertex data and add the missing excess at that point. Similarly, the edges will each view the neighboring vertex information and adjust accordingly.

3.3.3 *Execution* Step

During this step, the graph needs to use the data stored in each vertex from the *Surveying* step to adjust vertex and edge values, bringing the graph back into a valid, flow-conserving state by the end of the iteration. This involves two specific operations: updating the excess of vertices that received flow via a push, and updating the residual edges on which the pushes took place. In GraphX, the `aggregateMessages` method only allows for messaging to each of the vertices, but not to the edges. In other words, there is no direct MapReduce operation that allows both vertices and edges to be updated simultaneously. For this reason, we first use `mapTriplets`, a method that updates edge values exclusively, and then use `aggregateMessages` to update the vertices. The code used to update the edge values can be viewed in Figure 3.4.

```
// Update edge values based on selected pushes
graph = graph.mapTriplets[EdgeData](
  (edgeTriplet: EdgeTriplet[VertexData, EdgeData]) => {
    if (edgeTriplet.srcAttr._3.contains((edgeTriplet.dstId, true))) {
      // Push from source to destination
      val pushAmount: Int = edgeTriplet.srcAttr._3((edgeTriplet.dstId, true))
      (edgeTriplet.attr._1 + pushAmount, edgeTriplet.attr._2)
    } else if (edgeTriplet.dstAttr._3.contains((edgeTriplet.srcId, false))) {
      // Push from destination to source
      val pushAmount: Int = edgeTriplet.dstAttr._3((edgeTriplet.srcId, false))
      (edgeTriplet.attr._1 - pushAmount, edgeTriplet.attr._2)
    } else {
      // No push
      edgeTriplet.attr
    }
  }
)
```

Figure 3.4: *Execution* Update Edges Code

Unlike the `aggregateMessages` method that uses a messaging system to form a new RDD, `mapTriplets` simply takes a mapping function as a parameter that directly updates

the edges with new values. The sole parameter to this mapping function is an `EdgeTriplet`, which is similar to the `EdgeContext` type that was used in `aggregateMessages`, except it does not include the extra messaging functions. There are three different cases that could be true along an edge:

- The source vertex includes the destination vertex's ID in its `VertexPushMap` and the push is in the direction of the edge. In this case, the destination vertex needs to add the pushed flow to its excess.
- Conversely, the destination vertex includes the source vertex's ID in its `VertexPushMap` and the push is in the *reverse* direction of the edge. In this case, the source vertex needs to add the pushed flow to its excess.
- If neither of the above conditions are true, then no push occurred along the edge.

These conditionals indicate why the `Boolean` "push direction" flag is necessary. It ensures that edges of opposite direction, but between the same vertices, do not add excess mistakenly for data in the `VertexPushMap` that resulted from the other edge. In the first case when the push is in the same direction as the edge, the push amount is simply added to the edge flow value. In the second case when the push is in the *reverse* direction of the edge, the push amount is subtracted from the current flow on the edge. Otherwise, the same edge values are retained.

Once the edge values have been updated, the next step is to update the vertices in a similar fashion. The logic is comparable, only using the `aggregateMessages` method rather than `mapTriplets`. As in the *Surveying* step, this process involves using `aggregateMessages` to build an RDD with message information, and then applying `outerJoinVertices` to combine the messaging information with the graph to construct the new values at each vertex. The code used to build the `executedPushesRDD` can be seen in Figure 3.5.

Unlike the `aggregateMessages` call used in the *Surveying* step that had the complex `SurveyMessage` type for its messages, this step only requires each message to use an `Int`.

```
// "Execution" MapReduce step
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
  }
)
```

Figure 3.5: *Execution* Update Vertices Code

This singular value simply represents an adjustment in excess that should occur at the vertex receiving the message. In the mapping function, we see the same logic that was used in the code to update the edge values. The edge simply examines whether the stored information indicates a push in either direction, and if so, a message with the push amount is sent to the vertex receiving the flow. All the reduce function needs to do is add all of the incoming messages into a single sum. For example, if three different vertices are all pushing to the same destination vertex, that vertex will be sent three messages, and the reduce function will combine them into a single message containing the total accumulated flow from all sources. When this MapReduce operation is complete, `executedPushesRDD` will contain all the vertices that received flow, along with the total amount of flow received.

Just as in the *Surveying* step, this RDD needs to be merged with the graph to allow the RDD's messages to modify the actual values at each vertex. This is done quite simply using `outerJoinVertices` and storing the sum of the original vertex excess and the message value from the RDD. As before, Scala's `Option` type is used to account for vertices that are in the graph, but not included in `executedPushesRDD`. This code can be viewed in Figure 3.6. The

`Option` class provides a convenient `getOrElse` method to easily define a value for missing messages, in this case "0".

```
// Add excess to vertices receiving flow
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)
  }
}
```

Figure 3.6: *Execution* Vertex Program Code

At this point, the graph is in a valid state and all pushes that were possible during that iteration have been executed. First, the *Surveying* step selected push operations, adjusted excess at the source vertices, and stored the the push information in the source vertices' data. Then, in the *Execution* step, both the vertices and edges updated their values based on the information stored in the vertices during the *Surveying* step. This reaches the end of the main `while` loop, and it will continue with an additional iteration as long as there were any messages during the *Surveying* step. If not, then all of the remaining excess is in either the source or the sink, and the algorithm has completed.

The RDD method used on `eligiblePushesRDD` to count the number of messages and dictate the continuation of the main loop is, aptly named, `count()`. While this method seems trivial, it actually brings up an important concept that should be understood about RDDs. Operations on RDDs such as `aggregateMessages` are what might be called "lazy" operations. That is, they don't actually get executed when they are called, but rather the stack of RDD operations will be executed once an operation requires the RDD to complete the computations. In this case, `count` is a good example of an operation that requires the RDD to execute the stack of pending operations, and effectively materializes the RDD in memory. This may be important to understand when running performance diagnostics, as measuring the run time of methods such as `aggregateMessages` will appear to be extremely fast, but it is simply because the work has not actually been executed yet. Conversely, a

call to `count` could be observed as taking a significant amount of time, when in reality it is simply catching up on all the operations that have not been forced into execution yet.

3.4 Simple Example

To provide a more concrete explanation of how this parallelized version of the Push-Relabel algorithm might work, this section will step through a simple example. Obviously, the scale of this example is trivial, but it should allow for the algorithm to be conceptualized more visually and, as as result, understood more clearly. Consider the graph shown in Figure 3.7, with vertex A as the source and vertex E as the sink. Each vertex in the figure shows both its excess and height label. It is assumed that the IDs of vertices A-E are 1L-5L respectively, and that the initialization steps before the main execution loop have taken place. This means that all edges out of the source vertex have been saturated, which is why vertex B begins with an excess of 5.

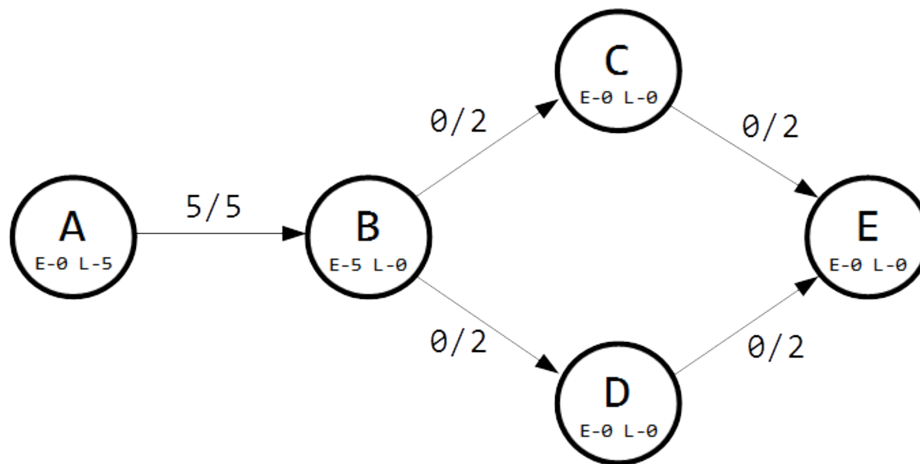


Figure 3.7: Simple Example - Initialized Graph.

During the *Surveying* step of the first iteration, there are three Triplets that meet the criteria to send a message: AB, BC, and BD. As a reminder, a message will be sent as long as a vertex in the Triplet has excess, has capacity left to push the excess along the edge, and the vertex with excess is neither the source vertex nor the sink. At this point, each Triplet

will either send a message with a possible push, or a message with information for a possible relabel. In this particular case, none of the edges are eligible to push as the height label of A is 0, implying that it can only push to a vertex with a height label of -1. As a result, the following three messages are sent to vertex B:

- From AB to B - `(Map(), 5)`
- From BC to B - `(Map(), 0)`
- From BD to B - `(Map(), 0)`

These three messages are then reduced to the single message `(Map(), 0)`, which is simply the concatenation of the three empty Maps, and the minimum of the the three height labels. This single message, stored in `eligiblePushesRDD`, is then joined with the original graph. For each of the vertices in the graph that did not receive a message, the same excess and height label are retained along with an empty `VertexPushMap`. The only vertex that received a message in this first iteration is vertex B. When the vertex program compares the height label of vertex B to the one stored in the message, it is found that they are both equal. This indicates that vertex B is eligible for a relabel, and the height label of the vertex is set to 1 (one greater than the height label of the message).

At this point, the *Execution* step begins. Since the `VertexPushMap` at every vertex is empty, there are no pushes that need to be executed during this step. The MapReduce operation will result in `executedPushesRDD` being empty, and joining it with the original graph will produce the same graph. Likewise, all of the edges will be mapped to their original values. With the conclusion of the *Execution* step, the number of messages in `eligiblePushesRDD` are totaled to equal 1 (only vertex B), which triggers another iteration of the whole execution loop. The state of the graph after the first iteration can be seen in Figure 3.8.

In the *Surveying* step of the second iteration, as in the first iteration, vertex B is the only vertex with any excess. So just as before, edges AB, BC, and BD are going to send

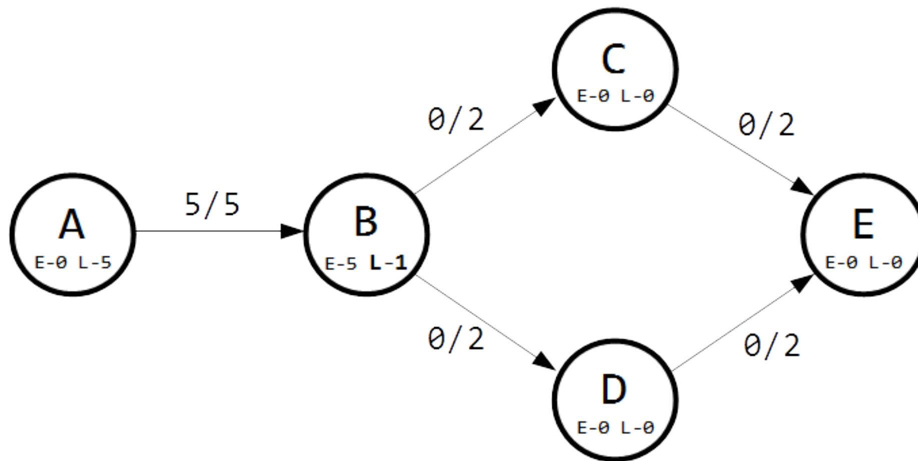


Figure 3.8: Simple Example - After Iteration 1.

messages. Unlike in the first iteration, vertex B now has a height label of exactly one greater than both vertex C and vertex D, allowing for a possible push. This results in the following three messages generated by the mapping function:

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

Each of the pushes maps a tuple (containing the ID of the vertex receiving flow and the push direction on the edge) to the capacity available to push. This is then reduced to the single message: `(Map((3L, true) -> 2, (4L, true) -> 2), 0)`. Again the reduced Map is simply the concatenation of all the Maps from each message to that vertex. In this case, all the messages are destined to B as it is the only vertex with any excess. This message is then joined with the original graph. This time, vertex B is not eligible for a relabel as it's height label of 1 is greater than the minimum neighboring height label of 0 as specified in the message. Rather, it follows the conditional logic into the push selection code where it will loop over the possible pushes found in the message and select them one by one. The push to

vertex C (ID of 3L) has a capacity of 2, and vertex B has an excess of 5. Taking the minimum of 2, it adds that push to the `selectedPushes` Map and subtracts the flow from the available excess. The next push is to vertex D, with another capacity of 2. This time the excess at vertex B is 3, which is still sufficient and the push of 2 is stored in `selectedPushes`. This leaves 1 remaining excess at vertex B. There are no more possible pushes in the message, so the loop terminates and the following data is stored in vertex B:

```
(1, 1, Map((3L, true) -> 2, (4L, true) -> 2))
```

This data includes the excess of 1 that remains at vertex B, the height label that has remained at 1, as well as the Map of selected pushes that will be referenced during the *Execution* step. Moving on to the *Execution* step, the edges are the first to update. The edges AB, BC, and BD will all view the selected pushes in vertex B. Since the `VertexPushMap` includes the IDs of both vertex C and vertex D, edges BC and BD will both increase their flow by 2. To update the vertices, `executedPushesRDD` is first populated using `aggregateMessages`. As when updating the edges, the Triplets BC and BD both view the push information in vertex B and send the following messages:

- From BC to C - `2`
- From BD to D - `2`

The vertex program of the *Execution* step then adds these values to the existing excess at vertices C and D, which at this point is zero in both cases. The value of `activeMessages` is calculated to be 1, and therefore the algorithm will continue with another iteration. The state of the graph after this second iteration can be seen in Figure 3.9.

In iteration 3, we will begin to see some evidence of the benefits of the parallelized approach since there is excess at three different vertices. Another key point to remember is the concept of adjacency in the residual graph. Because the edges BC and BD are both completely saturated, there is no residual edge in that direction. On the other hand, there

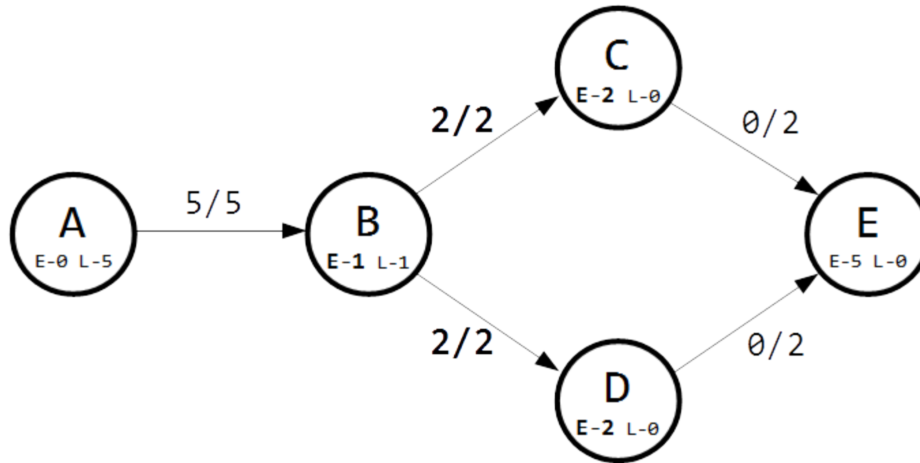


Figure 3.9: Simple Example - After Iteration 2.

now exist residual edges in the *opposite* direction on those edges, CB and DB, because the flow could potentially be reduced by "pushing" excess back to vertex B. Therefore, in the *Surveying* step, the Triplet AB is the only edge that has the potential to send a message to vertex B. In this case, a push is not possible since the height label of 1 is not one greater than the height label of 5. So instead, a message recording the neighboring height level is sent to B. Both vertices C and D also do not find any possible pushes since their height labels are both 0. Similarly, they receive messages that only include height label information. The messages produced from the mapping function in this step are as follows:

- 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)`

Vertices C and D both reduce their two messages to a single message, effectively ignoring the messages from BC and BD since the height labels on the ends of those connections were

larger. `eligiblePushesRDD` then contains the following information:

- Vertex B - `(Map(), 5)`
- Vertex C - `(Map(), 0)`
- Vertex D - `(Map(), 0)`

These messages are joined with the original graph, with each message triggering a relabel at the corresponding vertex. As in iteration 1 when there were no selected pushes, the entire *Execution* step is a trivial process of forming an empty RDD and joining it with the graph to produce the same result. The graph resulting from the relabels of iteration 3 is shown in Figure 3.10.

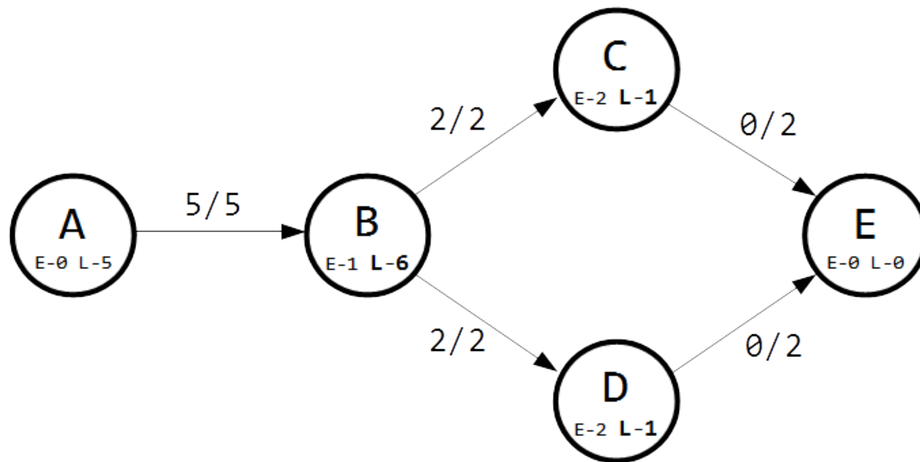


Figure 3.10: Simple Example - After Iteration 3.

Since relabel operations effectively create the required conditions for a push to take place, we can expect an iteration of relabels to be followed by an iteration of pushes. In the *Surveying* step of iteration 4, the exact same messages are generated as in iteration 3, except that now some of them contain information about possible pushes. The initial mapped messages are as follows:

- 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)`

Note that the push information sent to vertex B has set the direction flag of the key to `false` to reflect that the "push" is actually reducing the flow on the directed edge. The messages are then reduced and stored in `eligiblePushesRDD`:

- Vertex B - `(Map((1L, false) -> 5), 5)`
- Vertex C - `(Map((5L, true) -> 2), 0)`
- Vertex D - `(Map((5L, true) -> 2), 0)`

The vertex program selects which pushes will be executed. Each vertex will choose to execute the single push assigned to it, though vertex B will modify the push value to be 1 since the excess available is smaller than the capacity of the push. The following data is stored at each of the three vertices:

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

This vertex data is used by both the `mapTriplets` and `aggregateMessages` calls in the *Execution* step to update the edges and vertices in the graph. Edges AB, CE, and DE simply update their flow values (decreasing the flow in the case of AB). Those same edges then send messages to the vertices whose IDs are found in the `VertexPushMap` of each vertex, and the flow is added during the vertex program. The Triplets that update their values will always

be the same as the Triplets that send messages; the only reason that the separate function calls are necessary is because GraphX does not have a graph aggregation method that allows messaging and updating of both vertex and edge values simultaneously. Once all of the graph values have been updated, the state of the graph matches what is shown in Figure 3.11.

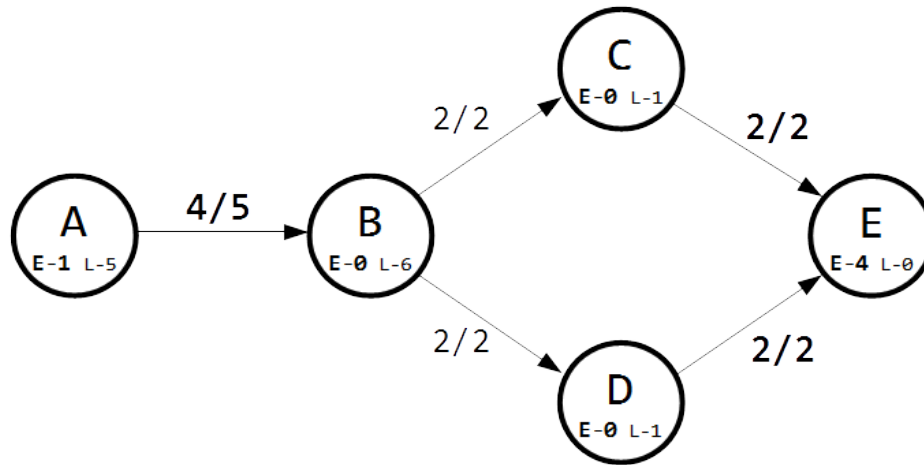


Figure 3.11: Simple Example - After Iteration 4.

You will notice at this point that all of the excess is housed at the source vertex (A) or at the sink (E). This, by definition of the Push-Relabel algorithm, means that the process is complete and the maximum flow has been found. However, after iteration 4, the count of `activeMessages` is still 3, triggering an additional fifth iteration. On this iteration, `eligibleMessagesRDD` will be empty as messages can only be sent when internal vertices (not source and not sink) have remaining excess. The rest of the execution loop will continue with empty message RDDs making no modifications to the graph. At the end of this "stable" iteration, `activeMessages` will then be calculated to be 0, and the main loop will be exited.

While this example is obviously trivial, it also shows how the distributed approach can become beneficial as the graph begins to branch. Whereas in the serial implementation the operations would be arbitrarily executed one after another, this MapReduce approach allows for *every* possible operation to be executed on *every* iteration.

3.5 Algorithm Correctness

For the most part, the algorithm and implementation presented mirror the exact process as the single-threaded Push-Relabel algorithm. The primary difference is that every possible operation is being executed simultaneously, which may raise questions about whether the correctness of the algorithm is retained in all cases. There are two cases that need to be validated in terms of correctness: simultaneous pushes to or from the same vertex, and simultaneous relabels on neighboring vertices. In the case of pushing, it is more intuitive that correctness is maintained. A vertex cannot push more than the excess available, and any simultaneous incoming flow is combined into a single sum in the MapReduce phase of the "Execution" step. As a result, there is no way for simultaneous flows to cause an invalid state.

Relabeling on the other hand, may present more of a concern. Since the criteria for relabeling is that the height label is equal to or less than all neighboring vertices, it is possible that two neighboring vertices could relabel in the same step. This would be the case if the vertices have an equal height label, and all other neighboring vertices have equal or higher height labels. This presents a concern that it may be possible for two neighboring vertices to relabel indefinitely, rather than having one vertex relabel and then push to the other. In addition, there could be any number of vertices connected in this sort of "infinite cycle of relabeling." To show that this case is not possible, consider the following facts:

- In order for a vertex to relabel, there must be excess flow currently at that vertex.
- In order for there to be excess flow at a vertex, there must be a residual path from the source to that vertex. In other words, the flow must have originated at the source, so there must be a path in the residual graph to push the flow back to the source.
- The source vertex is not allowed to relabel.

In order for the relabeling of adjacent vertices to go on infinitely, the height labels of all the vertices must remain equal to or less than all other neighboring vertices. Since the

relabeling of these vertices is pushing the height labels upwards to infinity, it follows that every neighboring vertex must also increase for the process to continue infinitely. We know from the second point that there must be a residual path from the vertices back to the source. Since the source can never relabel, this ensures that the process can never go on infinitely. Eventually, at least one of the relabeling vertices must have a height label greater than a neighboring vertex or the source, at which point a push will occur and break the cycle. As a result, simultaneous adjacent relabeling has no effect on the correctness of the algorithm.

CHAPTER 4

IMPLEMENTATION VARIATIONS

Upon completing the basic implementation and validating it on small examples, there were several variations that were analyzed either out of necessity to run on larger datasets (in the case of checkpointing) or simply in the attempt to improve performance. This chapter will discuss several of these modifications and will set the stage for some of the comparative testing results that will be detailed in the next chapter.

4.1 Checkpointing

Soon after the implementation was completed, I moved towards trying to test it on larger datasets and benchmarks. It quickly became evident that there were some problems that arose even when the graph was relatively small. Not problems in correctness, but rather stack overflow errors that prevented the algorithm from completing. This is due to the way that RDDs in Spark track their associated lineage. While this lineage is necessary to provide fault tolerance by providing a way to recreate lost data, it also means that this lineage grows with more operations done on the same RDD. This is a non-issue on computations that require a relatively small number of iterations, but on high-iteration algorithms the lineage can grow to the point where the serialization of the RDD can cause a stack overflow. This is simply because the lineage is serialized recursively, and as a result it can overflow the stack if the levels of recursion exceed what is available. For situations where this is a problem and simply allocating a larger stack is not a feasible solution, Spark provides checkpointing to allow the computed RDD to be saved to disk and then have the lineage truncated.

In order to implement checkpointing, the first step is to specify a directory for the saved data. The `SparkContext` class provides a method to specify a directory, and I was able to implement it with the following line:

```
sc.setCheckpointDir("hdfs://")
```

Note that `sc` is simply the reference to the `SparkContext` that is provided when running files from the Spark shell. Using the base path of the Hadoop distributed file system that is available on a cluster was the simplest solution. Once the directory is specified, the main execution loop just needs to call the `checkpoint` method after a set number of iterations. I defined this iteration count value in a constant variable named `CHECKPOINT_ITERATION_COUNT` and then added the following code to the end of the main execution loop (right after the *Execution* step):

```
if (iteration % CHECKPOINT_ITERATION_COUNT == 0) {
  graph.cache()
  graph.checkpoint()
}
```

When choosing a value for `CHECKPOINT_ITERATION_COUNT`, I initially set it to 50 as it was shortly after 50 iterations that the `StackOverflowError` would occur without checkpointing. In section 5.3, I will discuss some of the observations made when adjusting the spacing of the checkpointing operation. You will also notice in the code that the method `cache` is called before the actual `checkpoint` method. This is recommended so that the computed RDD is already in main memory, rather than needing to be recomputed to save it to the file. The addition of checkpointing to the base implementation described in section 3.3 allowed for a graph requiring an arbitrarily large amount of iterations to run without encountering any issues with overflowing the stack.

4.2 Caching

One optimization that is recommended in Spark is to manually cache RDDs in memory if they are being used repeatedly. By default, a materialized RDD is never cached in main memory. This means that operations on the same RDD require recomputation from the RDD lineage. Caching keeps the RDD materialized in memory, allowing the recomputation to be avoided. The implementation presented in this thesis involves a lot of repeated operation on the same graph during each iteration, indicating that caching could have a significant impact

on performance. Spark makes this easy to implement, by simply providing a `cache` method that can be called on an RDD. Similarly, the `Graph` class has a `cache` method that applies to both its vertices and edges. To implement caching in the implementation, I simply added `.cache()` to the end of each operation that updated the graph.

In addition to allowing for explicit caching of RDDs, Spark also allows RDDs to be uncached manually. While cached results will simply be evicted in the order of the least recently used when the cache runs out of space, explicitly uncaching data can help the garbage collection to run more efficiently. This is particularly applicable to iterative algorithms, and one of the reasons why it is recommended to use the Pregel API, which handles the memory persistence behind the scenes. As the proposed implementation was unable to be implemented using the Pregel API, all of the caching needed to be done manually. As far as unpersisting[12] data from memory, attempts to mimic the unpersisting pattern contained in the Pregel API did not seem to have the desired effect. This could be due to the more complex structure of each iteration, but I chose to stick with the implementation that cached and let the normal cache eviction handle any uncaching. Section 5.2 will discuss the increase in performance observed when applying caching to the algorithm.

4.3 Restricted Active Set

One aspect of the Push-Relabel algorithm is that only the vertices with excess have the potential to perform an operation. While this can be a large number of vertices if the graph branches significantly, it is still likely that at any given time, a relatively small percentage of the entire graph is eligible to push or relabel. Despite this fact, the MapReduce operations in GraphX will be applied to every Triplet in the graph, which certainly seems like more work than is desired. In previous versions of Spark and GraphX, the `mapReduceTriplets` method (which has been replaced by `aggregateMessages`) provided an optional parameter of an "active set." This allowed you to pass an argument RDD, and then the MapReduce

[12] Note that the `cache` method is equivalent to the `persist` method with the default memory level parameter. There is no "uncache" method provided, so the "persist" terminology is used here.

operation would only be applied to Triplets that were adjacent to the vertices in the RDD[13]. In newer versions of Spark and GraphX, the updated method `aggregateMessages` does not include this option, with the intention of simplifying the interface.

By utilizing the `GraphImpl` class that is not part of the public API, I was able to adjust the MapReduce steps of the algorithm to accept an active set. This code can be seen in Appendix A. This meant that on each iteration, I would only consider Triplets that were adjacent to the vertices that were involved in the active messages of the previous iteration. While I expected that this might have a large impact on performance, it ended up being insignificant and even hurting performance in some cases. I suspect this is because even though the active set prevents the mapping method from being run on all of the Triplets in the graph, every Triplet in the graph still needs to check if it is in the RDD, resulting in the same linear amount of operations. In other words, the number of operations is not actually being reduced, rather it is just ensuring that a single, simple conditional is executed on non-active Triplets rather than the full mapping function. In cases where the mapping function is very intensive, this could make a noticeable impact. But in this case, the mapping function used to build up `eligiblePushesRDD` is relatively tame, only containing a series of conditional checks. So while conceptually I envisioned reducing the operations by a large factor, it in reality only reduced the complexity of some of the operations by an insignificant amount.

That said, it still seems that it should help marginally, and certainly never hurt performance. Some other variables that may be causing this discrepancy are the added complexity to the dependency graph of the RDDs, or simply because it is an unsupported feature excluded from the public API. When I mention the complexity of the RDD dependency graph, I mean that providing an active set as an RDD introduces an additional operation that needs to fit into the RDD lineage. Spark's performance is almost entirely driven by the performance of these RDDs, so it is possible that there are some negative side effects that

[13]The method provided options to specify the direction of edges to be considered, relative to the vertices included in the active set.

outweigh the optimization that the restricted active set is trying to accomplish. The fact that the option to pass an active set has been removed from the public API in recent versions seems to also support the notion that it may not be as useful of an optimization as it might sound conceptually. More concrete answers to these questions would certainly require more testing and validation, but for purposes of this thesis, I concluded my research of the area here and opted to exclude the active set implementation from my experimentation.

CHAPTER 5
EXPERIMENTATION

In approaching experimentation, the goal was not to analyze whether this implementation is competitive with existing solutions to the maximum flow problem, but rather to observe its relative performance when varying the datasets, implementation details, and the cluster setup. Distributed frameworks like Spark require datasets that truly fit in the category of "big data" in order to judge their usefulness in comparison to their single-threaded counterparts. It was not feasible to analyze at that scale for this project, though that certainly could be an avenue to explore in the future. Rather, we utilized the web services provided by Amazon to run modest tests on a small cluster. Amazon Elastic MapReduce [22] allows for the creation of low-cost, on the spot clusters, and conveniently has options to deploy Spark on the cluster without any manual setup. This proved to be a valuable asset as we were able to avoid some of the overhead that comes with manually setting up and maintaining a cluster.

Amazon EMR offers a variety of instance types with which to create a custom cluster. Several of these options were tested, and I settled on using two `c3.xlarge` instances as that particular setup showed good relative performance. The details provided by Amazon on the C3 instance types are shown in Figure 5.1. While you can see that there are larger instance types within the C3 model list, the tier of my account on Amazon only allowed me to scale to a certain threshold. Since the primary intention of running these tests was not to try and be competitive with outside results, the simple cluster options available to me were sufficient to gather data to observe relative results.

Tests were run by initializing a cluster on Amazon, securely copying (`scp`) over the implementations and datasets, and then using SSH to access the cluster and run the code through the Spark shell. Again, this process was made convenient by Amazon's clear instructions and the option to pre-install Spark on the cluster. This pre-installation means that all of the

C3

Features:

- High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge) Processors
- Support for Enhanced Networking
- Support for clustering
- SSD-backed instance storage

Model	vCPU	Mem (GiB)	SSD Storage (GB)
c3.large	2	3.75	2 x 16
c3.xlarge	4	7.5	2 x 40
c3.2xlarge	8	15	2 x 80
c3.4xlarge	16	30	2 x 160
c3.8xlarge	32	60	2 x 320

Figure 5.1: Information on Amazon's C3 instance type. [22]

necessary installation to run Spark is handled by the service, not requiring any configuration on the part of the user. Spark version 1.5.0 was used for all tests. Noting this version may be of importance due to the rapid development of Spark. In other words, it is very possible that the code provided in this thesis becomes incompatible with a version of Spark in the near future[14].

5.1 Datasets

Selecting datasets to use for experimentation proved to be a slightly difficult task. Some maximum flow benchmark files were found online [23], but the large size of these files introduced an additional complication that comes with parallel computing, and that is distributing the dataset. Up to this point, all of my testing had been through the Spark shell, through which the implementation would just read in a local file containing the graph information. This means that the entire graph is being read into the memory of the single master node. As I attempted to scale my runs to much larger datasets, it resulted in exhausting the Java heap space and terminating the run. The solution to this would be to store the actual dataset in the Hadoop distributed file system, and then have Spark read it in from the cluster. Due to time constraints and the realization that the experimentation was not

[14]For example, just over the span of the development of this project, Spark deprecated the `mapReduceTriplets` method and introduced `aggregateMessages`.

aiming to be competitive, I settled to simply use smaller datasets that avoided this problem.

Since these tests were intended to provide a window into how the implementation handled different datasets and the relative performance of the variations, we chose to use "contrived" graphs that emphasized the extreme cases of potential input. On one side of the spectrum, a graph that is just a single line with no opportunity for parallelization (the worst case), and on the other side, a graph that branched exponentially providing huge parallel potential (the best case). Two Python scripts were developed to aid with the creation of these contrived graphs, and are included in Appendix C. In the end, I decided on four different datasets:

- **single-line** - This contrived graph was generated by one of the python scripts, and is simply 500 vertices chained together in a single line from source to sink.
- **parallel-5-5** - This contrived graph was also generated by one of the python scripts, and consists of 5 levels of branching where on every level, each vertex branches to 5 additional vertices. In other words, the source branches to 5 vertices, which then branch to 25 vertices, and then to 125, 625, and 3125. The last 3125 vertices then all connect to the sink. The capacities of these edges are set up in such a way that all of edges to the sink in the last level have a capacity of exactly one.
- **parallel-12-5** - This contrived graph was generated just as `parallel-5-5`, but with a branch factor of 12 as opposed to 5. This leads to the vertices at each level growing from 12 vertices connected to the source, all the way to 248832 vertices connected to the sink.
- **RMF-wide** - This is the smallest of the benchmark graphs that was obtained online [23], included to provide a more complex and non-contrived graph for comparison. Note that due to the high number of iterations required for this dataset to complete (and the modest cluster setup being used), measurements on this dataset were made based on the first 200 iterations.

In short, the four datasets are just intended to represent graphs with extremely different compositions so that it can be seen if trends in the data hold true regardless of graph structure. The two contrived parallel datasets, `parallel-5-5` and `parallel-12-5`, are interesting in that they vary drastically in size (3900 edges vs. 271440 edges) yet they take the same number of iterations for the algorithm to complete. Specifically, the algorithm will complete after iterations equal to twice the number of "levels" in the graph. This is because on each level, the vertices will require an iteration to relabel, and another iteration to push. Comparing the results on these datasets should provide some insight into how effectively the parallelization is affecting performance.

5.2 Caching vs. Non-caching results

The first test involved running the base implementation with all four datasets, and comparing the results to the caching implementation on the same four datasets. The results of these tests can be seen in Table 5.1.

Table 5.1: Base implementation vs. Caching implementation.

	single-line (s)	RMF-wide 200 iter. (s)	parallel-5-5 (s)	parallel-12-5 (s)
Base	750.736	413.838	16.207	118.728
Cache	522.527	306.654	13.432	92.042
Speedup	1.437	1.350	1.207	1.290

There are a few observations that could be made here. Firstly, and most obviously, caching clearing has an impact on performance. This is to be expected, and these initial results confirm these expectations with the average run times of the cached implementation being roughly 75% of the run times of the base implementation counterparts. Another interesting observation is that the ratio of improvement from the base implementation to the cached implementation seems to grow with the run time. In other words, the `single-line` dataset which took the longest amount of time saw the highest percentage of improvement due to caching (69.6% the original run time) while the dataset with the smallest run time,

`parallel-5-5`, saw the lowest percentage of improvement due to caching (82.9% the original run time). This trend holds for the other two graphs that fall in between in terms of run time. The explanation for this may have to do with Amdahl's law [24] which bounds the expected improvement of an optimization to the percentage of the system that the optimization applies to. In other words, as the run time grows larger, it is possible that a higher percentage of the total run time is actual computation on the graph and RDDs as opposed to overhead in the system. Since the caching optimization applies directly to computation on the RDDs, this means that the optimization will improve a larger portion of the total run time. Considering that all of these datasets do not even begin to approach the level of "big data", this trend suggests that the benefits of caching on a truly large dataset would be drastic. However, no strong implication can be made without verifying this trend on larger datasets.

The other key observation from this data is the comparison between the `parallel-5-5` dataset and the `parallel-12-5` dataset. Both of these datasets took the same number of iterations to complete. We can see that even though the `parallel-12-5` dataset has roughly seventy times as many edges as the `parallel-5-5` dataset, it only took roughly seven times as long to run in both the base and cached implementation. This emphasizes that the parallelization of the implementation is effective in keeping the run time from scaling up at the same rate as the graph size, as it would if the implementation was executed serially.

5.3 Checkpointing Intervals Results

Another variable that could be adjusted to view its effect on performance is the number of iterations in between each checkpoint. As mentioned in section 4.1, checkpointing is necessary to avoid the stack from overflowing after too many iterations. When implementing checkpointing, I naturally selected the checkpointing interval to be just below what would normally cause the error, in this case, every 50 iterations. The intuition was that checkpointing is an expensive operation since it requires interaction with disk, and therefore it should be done as infrequently as possible. The results as shown in Table 5.2 seem to slightly contradict this notion.

Table 5.2: Different checkpointing intervals on caching implementation.

	single-line (s)	RMF-wide 200 iter. (s)	parallel-5-5 (s)	parallel-12-5 (s)
10 iterations	456.626	287.349	15.505	94.515
25 iterations	427.684	314.942	13.251	92.612
50 iterations	522.527	341.530	13.432	92.042

It seems that checkpointing more often should always hurt performance, but the results seem to be more unpredictable. There may be several reasons for this. One may be that the Amazon cluster is showing inconsistencies as I will discuss in section 5.4. Another possibility is that the RDD lineage is another factor in the run time of each iteration. The purpose of checkpointing is to truncate this lineage, but if this lineage is also slowly degrading performance, then there is a balance between the overhead of checkpointing and the slowdown caused by letting the lineage grow. This speculation seems to be supported by the fact that iteration run times tend to trend upwards in the time in between checkpoints.

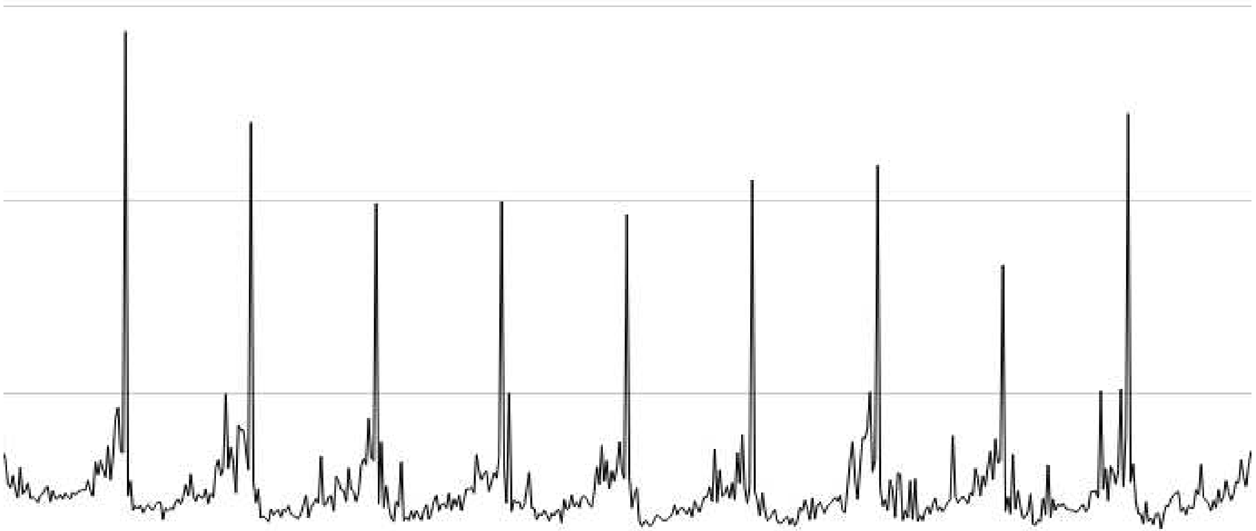


Figure 5.2: Iteration run times for cached implementation on single-line dataset with checkpointing interval of 50.

Consider Figure 5.2 which shows the relative iteration times of the cached implementation when run on the `single-line` dataset with a checkpointing interval of 50 (First 500 iterations

out of a total of 1000). The iterations that include checkpointing are easily identified by the spikes in the graph, but it is also evident that the iteration time trends upwards in between checkpoints, and then drops immediately after checkpointing. All of this seems to indicate that there may be some balance to make between checkpointing too often and incuring too many "spikes" in the graph, and checkpointing too infrequently, causing too much slowdown in the space between checkpoints.

5.4 Scaling and Amazon Inconsistencies

While the experimentation here seems to support several different arguments about the performance of RDDs and iterative computation in GraphX, it should all be taken with a grain of salt. While running the tests, it became evident that the Amazon clusters were not particularly reliable when it comes to performance, in that the overall speed of the cluster would fluctuate between runs and especially when a new cluster was created. Pairing this with the fact that the algorithm presented in this thesis is non-deterministic, that is, the pushes selected are arbitrary since a Map is not an ordered collection, it is hard to make any strong claims using the collected data. That said, the main points discussed here (caching and checkpointing) were patterns that were seen across the board, even if on varying timescales.

Some initial research indicates that I am not the first to observe the inconsistency of running benchmark tests on Amazon's services. I ran across a blog post on InfoWorld.com titled *Benchmarking Amazon EC2: The wacky world of cloud performance* with the subtitle, "The performance of Amazon machine instances is sometimes fast, sometimes slow, and sometimes absolutely abysmal."[25] This echoes my experiences with the service, and emphasizes that while it may be a very convenient option for testing and operating on a cluster without the overhead of maintaining one, it may not be the right choice if you need reliable hardware that gives you the power to isolate variables in performance testing.

All of this is also related to the topic of scaling. As mentioned before, scaling to what would be true "big data" scenarios would require distributing the dataset over the cluster,

rather than reading it from the master node. While that put a limit on scaling the datasets, there also was the limit on my basic account on Amazon as to how large I could scale one of my created clusters. Even with these constraints on both data and the cluster in play, I still intended to do some small tests on clusters of varying sizes. Specifically, I wanted to try three setups: 1 `c3.xlarge` instance, 2 `c3.xlarge` instances (the basic setup used in all the above experimentation), and 3 `c3.xlarge` instances. Unlike the previous tests, scaling the cluster would require me to terminate and create a new cluster to test a new size. This is where the variance in the cluster performance really was amplified. For example, in scaling from 2 instances to 3 instances, I at first saw a large performance boost, which was promising. Later, I tried the same tests and the performance was *worse* than the data I had collected from 2 instances. In addition, I found that creating a cluster of the *same size* would even yield different results than what I had collected earlier. For this reason, I could not justify including data from scaling the cluster; it simply loses too much reliability in relative performance. Keep in mind that *as long as I remained on the same cluster*, the relative results seemed to be fairly consistent. It was only when I terminated a cluster and created a new one that the cluster performance was a roll of the dice. This certainly isn't reason to blame Amazon; it makes sense for their service to provide the fastest instances available at any given time, even if that means variance in performance. But all of these observations of Amazon's cluster services are certainly things to still keep in mind, especially if you intend to be measuring benchmark performance.

CHAPTER 6
POSSIBLE FUTURE WORK

Given the scope of this thesis, there were many areas that show opportunity for exploration and analysis that were simply not possible to include here. In this section, I detail some of directions that I believe this work could be extended and continued.

6.1 Scaling and Verification of Approach

One obvious theme that has come up throughout the experimentation is the lack of ability to scale to a level that constitutes "big data."[15] While the primary goal of this project was to simply implement a solution to the Maximum Flow Problem in GraphX based on the Push-Relabel algorithm, a necessary next step in the validation would be to measure its performance at a large-scale, and more realistic level. Until that kind of testing is done, this particular implementation can be neither verified as effective, nor ruled out as an ineffective option. At a conceptual level, and throughout testing, the parallelization utilized in the algorithm is very evident. The number of active messages clearly grows as the algorithm progresses, with each active message representing an operation that can be executed in parallel on the graph. In addition, factors like caching and the comparison between the highly parallel datasets `parallel-5-5` and `parallel-12-5` seem to indicate that the benefits of the distributed approach will be magnified as the size of the problem is scaled.

As discussed in section 5.4, measuring cloud computing performance is not a trivial task. The approach to scaling the testing would most likely require a more stable cluster setup than what Amazon can provide. In addition, it would be helpful to remove the non-deterministic nature of the implementation. While it is not key to the algorithm itself that

[15] To put the concept of big data in perspective, the largest dataset used in this thesis for experimentation included 271440 edges (`parallel-12-5`). In comparison, the dataset used to validate GraphX on PageRank as specified on their website contained 3.7 billion edges.

push operations are selected using a consistent convention, it would certainly be helpful in benchmark testing to be able to run on the same dataset and maintain an identical execution path. Note that the additions necessary to make the implementation deterministic, perhaps by using an ordered Map, would likely have a negative effect on overall performance. Still, that may be offset by the benefit of being able to more easily isolate variables and test the implementation.

6.2 Algorithm Optimization

In addition to testing the performance of the current implementation, there is still a lot of room to try and optimize the algorithm. Listed here are some different approaches that I have considered for improving performance.

- Look into combining the *Surveying* and *Execution* steps into a single MapReduce step. I know I have made the argument that this was not possible due to how messages are sent and the need to update the graph, which is certainly true on the first iteration. But perhaps there is a way on subsequent iterations to include the *Surveying* step for the next iteration at the end of the MapReduce cycle in the *Execution* step. It is possible that this is not feasible, but it is worth exploring as eliminating a full MapReduce cycle on each iteration would have a huge impact.

- Consider other data structures for the message format. The reasoning for using a Map is because it allowed constant time access to the push amount when selecting pushes, rather than having to iterate over the entire data structure to find the matching value. That said, perhaps a simpler array would actually yield better results. The GraphX programming guide specifies that the `aggregateMessages` method performs optimally when the messages are constant sized, in that they add to a single value on reduction rather than concatenating into a list [16]. The current implementation with a Map certainly does not follow this recommendation, but there may be no way around that given the need to accumulate an unknown amount of possible push operations. What-

ever the case, it may be worth more analysis since any optimization in the messaging will be multiplied over the huge quantity of messages that are generated in the duration of a run.

- See if an uncaching strategy can improve performance. The fastest implementation that I tested involved caching RDDs, but never manually uncaching them. There may be an opportunity to optimize the garbage collection if the RDDs are also unpersisted properly.

REFERENCES CITED

[1] Computers are becoming faster and faster, but their speed is still limited by the physical restrictions of an electron moving through matter. what technologies are emerging to break through this speed barrier? http://www.scientificamerican.com/article/computers-are-becoming-fa/. Accessed: 2015-05-24.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[3] Apache hadoop. https://hadoop.apache.org/.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972.

[5] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[7] Powered by spark. https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark.

[8] Ibm announces major commitment to advance apache spark, calling it potentially the most significant open source project of the next decade. https://www-03.ibm.com/press/us/en/pressrelease/47107.wss.

[9] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2188-4. doi: 10.1145/2484425.2484427. URL http://doi.acm.org/10.1145/2484425.2484427.

[10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184.

[11] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[12] Apache spark. https://spark.apache.org/.

[13] Scala programming language. http://www.scala-lang.org/.

[14] Apache spark graphx. https://spark.apache.org/graphx/, .

[15] Graph analytics with graphx. http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html, .

[16] Graphx programming guide. https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api, .

[17] Wikimedia Commons. Max flow.svg, 2012. URL https://http://commons.wikimedia.org/wiki/File:Max_flow.svg.

[18] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988. ISSN 0004-5411. doi: 10.1145/48014.61051. URL http://doi.acm.org/10.1145/48014.61051.

[19] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404. URL http://www.rand.org/pubs/papers/P605/.

[20] V.Sachdeva D.A. Bader. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[21] Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1025–1033, 2011. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156.

[22] Amazon emr. https://aws.amazon.com/elasticmapreduce/, .

[23] The maximum flow project - benchmark. http://www.cs.tau.ac.il/~sagihed/ibfs/benchmark.html.

[24] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560.

[25] Benchmarking amazon ec2: The wacky world of cloud performance. http://www.infoworld.com/article/2613784/cloud-computing/benchmarking-amazon-ec2--the-wacky-world-of-cloud-performance.html, .

APPENDIX A - IMPLEMENTATION CODE

The implementation used for the experimentation is provided here in its entirety. Note that this is the version of the code that includes caching; the non-caching version would be identical except with all of the calls to `cache` removed.

CacheMaxFlow.scala

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

import org.apache.log4j.Logger
import org.apache.log4j.Level

import scala.io.Source
import java.io._
import java.util.Calendar
import scala.util.control.Breaks._

type VertexPushMap = Map[(VertexId, Boolean), Int]
type EdgeData = (Int, Int)
type VertexData = (Int, Int, VertexPushMap)
type SurveyMessage = (VertexPushMap, Int)

val CHECKPOINT_ITERATION_COUNT = 25

// Turn console logging off
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

// Create new logging file
val timestamp = Calendar.getInstance().getTime()
val logFileName = "log.txt"
val createdFile = new FileWriter(new File(logFileName))

// Define method to print both to console and logging file
```

```
def logMessage = { (str: String) =>
  println(str)
  val fw = new FileWriter(logFileName, true)
  try {
    fw.write(str)
    fw.write(System.getProperty("line.separator"))
  }
  finally fw.close
}

// Set checkpointing directory
sc.setCheckpointDir("hdfs://")

// val filename = "rmf-wide.n2.3141.shuf.bk"
val filename = "graph-singleline-500.bk"
// val filename = "graph-veryparallel-5-5.bk"
// val filename = "graph-veryparallel-12-5.bk"

logMessage("Filename: " + filename)
logMessage("Timestamp: " + timestamp)

// Initialize source and sink
var sourceId = -1L
var sinkId = -1L

/* READ IN .bk FILE */

// Loop line by line
var vertexBuffer = scala.collection.mutable.ArrayBuffer[(Long, VertexData)]()
var edgeBuffer = scala.collection.mutable.ArrayBuffer[Edge[EdgeData]]()

for (line <- Source.fromFile(filename).getLines()) {
  if (line.length() > 0) {
    if (line.charAt(0) == 'p') {
      // Create source and sink with ids based on total number of nodes
      val values = line.split(" ")
      sourceId = values(1).toLong
      sinkId = values(1).toLong + 1L
      vertexBuffer += ((sourceId, (0, values(1).toInt + 1, Map[(VertexId, Boolean), Int]())))
      vertexBuffer += ((sinkId, (0, 0, Map[(VertexId, Boolean), Int]())))
    } else if (line.charAt(0) == 'n') {
```

```
      // Create node and add any edges to source or sink
      val values = line.split(" ")
      if (values(2).toInt != 0) {
        vertexBuffer += ((values(1).toLong, (values(2).toInt, 0, Map[(VertexId, Boolean), Int]())))
        edgeBuffer += Edge(sourceId, values(1).toLong, (values(2).toInt, values(2).toInt))
      } else {
        vertexBuffer += ((values(1).toLong, (0, 0, Map[(VertexId, Boolean), Int]())))
      }
      if (values(3).toInt != 0) {
        edgeBuffer += Edge(values(1).toLong, sinkId, (0, values(3).toInt))
      }
    } else if (line.charAt(0) == 'a') {
      val values = line.split(" ")
      if (values(3).toInt != 0) {
        edgeBuffer += Edge(values(1).toLong, values(2).toLong, (0, values(3).toInt))
      }
      if (values.length > 4) {
        if (values(4).toInt != 0) {
          edgeBuffer += Edge(values(2).toLong, values(1).toLong, (0, values(4).toInt))
        }
      }
    }
  }
}
/* END READ IN .bk FILE */

// Add graph information to the log
logMessage("Number of Vertices: " + vertexBuffer.length)
logMessage("Number of Edges: " + edgeBuffer.length)
logMessage("")

// 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).cache()
```

```
val startTime = System.currentTimeMillis

// Iterate until no vertices push or relabel
while (activeMessages > 0) {

  val iterationStartTime = System.currentTimeMillis

  // "Surveying" MapReduce step -> (Map[dstId, pushAmount], dstHeight)
  val eligiblePushesRDD = graph.aggregateMessages[SurveyMessage] (
    // Map: Send message if push is possible
    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))
            }
          }
        }
      }

      // Make sure not to push from sink or source
      if (edgeContext.dstId != sinkId && edgeContext.dstId != sourceId) {
        // If a residual edge exists from destination to source
        if (edgeContext.attr._1 > 0) {
          // If destination has an excess
          if (edgeContext.dstAttr._1 > 0) {
            // If destination has height one greater than source
            if (edgeContext.dstAttr._2 == (edgeContext.srcAttr._2 + 1)) {
              // Push is possible, send message to source containing push information
              val pushAmount = math.min(edgeContext.attr._1, edgeContext.dstAttr._1)
```

```
            edgeContext.sendToDst((Map((edgeContext.srcId, false) -> pushAmount), edgeContext.srcAttr._2))
          } else {
            edgeContext.sendToDst((Map(), edgeContext.srcAttr._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))
  }
).cache()

// Store results in graph, only keeping as much as excess can support
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)
    }
  }
}.cache()

// Update edge values based on selected pushes
graph = graph.mapTriplets[EdgeData](
  (edgeTriplet: EdgeTriplet[VertexData, EdgeData]) => {
    if (edgeTriplet.srcAttr._3.contains((edgeTriplet.dstId, true))) {
      // Push from source to destination
      val pushAmount: Int = edgeTriplet.srcAttr._3((edgeTriplet.dstId, true))
      (edgeTriplet.attr._1 + pushAmount, edgeTriplet.attr._2)
    } else if (edgeTriplet.dstAttr._3.contains((edgeTriplet.srcId, false))) {
      // Push from destination to source
      val pushAmount: Int = edgeTriplet.dstAttr._3((edgeTriplet.srcId, false))
      (edgeTriplet.attr._1 - pushAmount, edgeTriplet.attr._2)
    } else {
      // No push
      edgeTriplet.attr
    }
  }
).cache()

// "Execution" MapReduce step
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
    }
  ).cache()

  // Store results in graph, only keeping as much as excess can support
  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)
    }
  }.cache()

  // Checkpointing
  if (iteration % CHECKPOINT_ITERATION_COUNT == 0) {
    graph.checkpoint()
    logMessage("---------- CHECKPOINTED ----------")
    val totaltime = math.floor(System.currentTimeMillis - startTime)/1000.0
    logMessage(s"Current Run Time: $totaltime")
  }

  // Store the number of messages to determine if more iterations are needed
  activeMessages = eligiblePushesRDD.count().toInt

  // Log iteration info and increment counter
  val iterationRunTime = math.floor(System.currentTimeMillis - iterationStartTime)/1000.0
  logMessage(s"Iteration: $iteration    Active Messages: $activeMessages    Iteration Time: $iterationRunTime")
  iteration += 1
}

val runTime = math.floor(System.currentTimeMillis - startTime)/1000.0
logMessage("")
logMessage(s"Total Run Time: $runTime s")
```

Additionally, included here is the code used when attempting to limit to an active set, as discussed in section 4.3.

ActiveSetMaxFlow.scala

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.graphx.impl.GraphImpl
import org.apache.spark.rdd.RDD

import org.apache.log4j.Logger
import org.apache.log4j.Level

import scala.io.Source
import java.io._
import java.util.Calendar
import scala.util.control.Breaks._

type VertexPushMap = Map[(VertexId, Boolean), Int]
type EdgeData = (Int, Int)
type VertexData = (Int, Int, VertexPushMap)
type SurveyMessage = (VertexPushMap, Int)

val CHECKPOINT_ITERATION_COUNT = 50

// Turn console logging off
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

// Create new logging file
val timestamp = Calendar.getInstance().getTime()
val logFileName = "log.txt"
val createdFile = new FileWriter(new File(logFileName))

// Define method to print both to console and logging file
def logMessage = { (str: String) =>
  println(str)
  val fw = new FileWriter(logFileName, true)
  try {
    fw.write(str)
```

```
    fw.write(System.getProperty("line.separator"))
  }
  finally fw.close
}

// Set checkpointing directory
sc.setCheckpointDir("hdfs://")

// val filename = "rmf-wide.n2.3141.shuf.bk"
// val filename = "graph-singleline-500.bk"
// val filename = "graph-veryparallel-5-5.bk"
val filename = "graph-veryparallel-12-5.bk"

logMessage("Filename: " + filename)
logMessage("Timestamp: " + timestamp)

// Initialize source and sink
var sourceId = -1L
var sinkId = -1L

/* READ IN .bk FILE */

// Loop line by line
var vertexBuffer = scala.collection.mutable.ArrayBuffer[(Long, VertexData)]()
var edgeBuffer = scala.collection.mutable.ArrayBuffer[Edge[EdgeData]]()

for (line <- Source.fromFile(filename).getLines()) {
  if (line.length() > 0) {
    if (line.charAt(0) == 'p') {
      // Create source and sink with ids based on total number of nodes
      val values = line.split(" ")
      sourceId = values(1).toLong
      sinkId = values(1).toLong + 1L
      vertexBuffer += ((sourceId, (0, values(1).toInt + 1, Map[(VertexId, Boolean), Int]())))
      vertexBuffer += ((sinkId, (0, 0, Map[(VertexId, Boolean), Int]())))
    } else if (line.charAt(0) == 'n') {
      // Create node and add any edges to source or sink
      val values = line.split(" ")
      if (values(2).toInt != 0) {
        vertexBuffer += ((values(1).toLong, (values(2).toInt, 0, Map[(VertexId, Boolean), Int]())))
        edgeBuffer += Edge(sourceId, values(1).toLong, (values(2).toInt, values(2).toInt))
```

```
      } else {
        vertexBuffer += ((values(1).toLong, (0, 0, Map[(VertexId, Boolean), Int]())))
      }
      if (values(3).toInt != 0) {
        edgeBuffer += Edge(values(1).toLong, sinkId, (0, values(3).toInt))
      }
    } else if (line.charAt(0) == 'a') {
      val values = line.split(" ")
      if (values(3).toInt != 0) {
        edgeBuffer += Edge(values(1).toLong, values(2).toLong, (0, values(3).toInt))
      }
      if (values.length > 4) {
        if (values(4).toInt != 0) {
          edgeBuffer += Edge(values(2).toLong, values(1).toLong, (0, values(4).toInt))
        }
      }
    }
  }
}

// Add graph information to the log
logMessage("Number of Vertices: " + vertexBuffer.length)
logMessage("Number of Edges: " + edgeBuffer.length)
logMessage("")

// 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)

// Set initial messages to be vertices connected to the sink (needed to establish active set)
var eligiblePushesRDD = graph.aggregateMessages[SurveyMessage] (
  // Map: Send push information to vertices that received flow
  edgeContext => {
    if (edgeContext.srcId == sourceId) {
```

```
      edgeContext.sendToDst((Map[(VertexId, Boolean), Int](), 0))
    }
  },
  // Reduce: Combine all incoming flow into a single total
  (a, b) => {
    (Map[(VertexId, Boolean), Int](), 0)
  }
)

val startTime = System.currentTimeMillis

// Iterate until no vertices push or relabel
while (activeMessages > 0) {

  var iterationStartTime = System.currentTimeMillis

  // "Surveying" MapReduce step -> (Map[dstId, pushAmount], dstHeight)
  eligiblePushesRDD = graph.asInstanceOf[GraphImpl[VertexData, EdgeData]].aggregateMessagesWithActiveSet[SurveyMessage] (
    // Map: Send message if push is possible
    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))
            }
          }
        }
      }

      // Make sure not to push from sink or source
      if (edgeContext.dstId != sinkId && edgeContext.dstId != sourceId) {
```

```
      // If a residual edge exists from destination to source
      if (edgeContext.attr._1 > 0) {
        // If destination has an excess
        if (edgeContext.dstAttr._1 > 0) {
          // If destination has height one greater than source
          if (edgeContext.dstAttr._2 == (edgeContext.srcAttr._2 + 1)) {
            // Push is possible, send message to source containing push information
            val pushAmount = math.min(edgeContext.attr._1, edgeContext.dstAttr._1)
            edgeContext.sendToDst((Map((edgeContext.srcId, false) -> pushAmount), edgeContext.srcAttr._2))
          } else {
            edgeContext.sendToDst((Map(), edgeContext.srcAttr._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))
  },
  // Specify triplet fields
  TripletFields.All,
  Some(eligiblePushesRDD, EdgeDirection.Either)
).cache()

graph = graph.joinVertices(eligiblePushesRDD) (
  (id: VertexId, data: VertexData, msg: SurveyMessage) => {
    if (msg._2 >= data._2) {
      // Eligible for relabel
      (data._1, msg._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._1.foreach(pushData => {
          val dstId = pushData._1._1
          val forwardPush = pushData._1._2
```

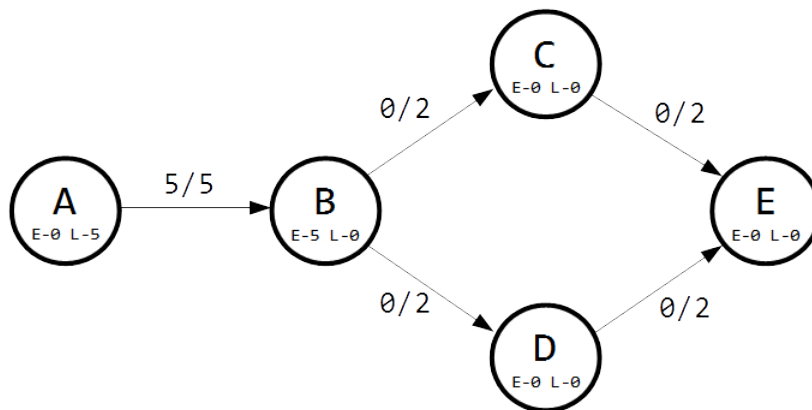
```
          val pushAmount = pushData._2
          if (excess > 0) {
            val selectedPushAmount = math.min(pushAmount, excess)
            excess -= selectedPushAmount
            selectedPushes((dstId, forwardPush)) = selectedPushAmount
          } else {
            break
          }
        })
      }

      (excess, data._2, selectedPushes.toMap)
    }
  }
).cache()

// Update edge values based on selected pushes
graph = graph.mapTriplets[EdgeData](
  (edgeTriplet: EdgeTriplet[VertexData, EdgeData]) => {
    if (edgeTriplet.srcAttr._3.contains((edgeTriplet.dstId, true))) {
      // Push from source to destination
      val pushAmount: Int = edgeTriplet.srcAttr._3((edgeTriplet.dstId, true))
      (edgeTriplet.attr._1 + pushAmount, edgeTriplet.attr._2)
    } else if (edgeTriplet.dstAttr._3.contains((edgeTriplet.srcId, false))) {
      // Push from destination to source
      val pushAmount: Int = edgeTriplet.dstAttr._3((edgeTriplet.srcId, false))
      (edgeTriplet.attr._1 - pushAmount, edgeTriplet.attr._2)
    } else {
      // No push
      edgeTriplet.attr
    }
  }
).cache()

// "Execution" MapReduce step
var executedPushesRDD = graph.asInstanceOf[GraphImpl[VertexData, EdgeData]].aggregateMessagesWithActiveSet[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
  },
  TripletFields.All,
  Some(eligiblePushesRDD, EdgeDirection.Either)
).cache()

// Store results in graph, only keeping as much as excess can support
graph = graph.joinVertices(executedPushesRDD) (
  (id: VertexId, data: VertexData, msg: Int) => {
    val test = scala.collection.mutable.Map[(VertexId, Boolean), Int]()
    // Add pushed flow to vertex
    (data._1 + msg, data._2, data._3)
  }
).cache()

// Checkpointing
if (iteration % CHECKPOINT_ITERATION_COUNT == 0) {
  graph.checkpoint()
  logMessage("---------- CHECKPOINTED ----------")
  val totaltime = math.floor(System.currentTimeMillis - startTime)/1000.0
  logMessage(s"Current Run Time: $totaltime")
}

// Store the number of messages to determine if more iterations are needed
activeMessages = eligiblePushesRDD.count().toInt

// Log iteration info and increment counter
val iterationRunTime = math.floor(System.currentTimeMillis - iterationStartTime)/1000.0
```

```
  logMessage(s"Iteration: $iteration    Active Messages: $activeMessages    Iteration Time: $iterationRunTime")
  iteration += 1
}

val runTime = math.floor(System.currentTimeMillis - startTime)/1000.0
logMessage("")
logMessage(s"Total Run Time: $runTime s")
```

APPENDIX B - DATASET FILE TYPE (.BK)

For all testing, the graphs were read in from external files in the `.bk` file format. Here this file format is described, along with a brief example. The first line in the file specifies the total number of both vertices and edges in the graph. Then each following line that is prefaced with the letter **n** defines a vertex in the graph. Three values are provided with the vertex: the ID, the capacity of its connection to the source, and the capacity of its connection to the sink. Note that the file does *not* actually define the source and sink explicitly. Rather, each vertex simply contains a value that represents its capacity to each of the abstracted vertices. Once all the vertices have been defined, Then the edges are defined, each line prefaced with the letter **a**. These lines include four values: the ID of the source vertex, the ID of the destination vertex, the capacity in the direction of the edge, and the capacity in the reverse direction of the edge. Once all the edges have been defined, the `.bk` file is complete. Consider the small graph used as an example in section 3.4:



This graph would be represented in a `.bk` file as the following:

```
p 3 2
n 0 5 0
n 1 0 2
n 2 0 2
a 0 1 2 0
a 0 2 2 0
```

Notice how the source and sink vertices are omitted and adjacent edges are simply included in the vertex defintions. The implementation detailed in appendix A iterates over a file in this format, and converts it into the appropriate arrays/RDDs that GraphX uses for its computation. More example `.bk` files can be found online at http://www.cs.tau.ac.il/~sagihed/ibfs/benchmark.html. Scripts that I created for generating contrived graphs in the `.bk` format are included in appendix C.

APPENDIX C - GRAPH GENERATION SCRIPTS

In order to create contrived graphs that tested the extreme input cases, I created a couple of Python scripts that generate `.bk` files based on a few parameters. The scripts and descriptions of their usage are included here.

The first script is used to generate a graph that is simply a straight line, without any opportunity for parallelization on branching paths. Assuming that python is installed, the script can be called from the command line as `generateSingleLineGraph` <`num_nodes`>. For example, `generateSingleLineGraph 1000` would create a `.bk` file that is simply 1000 vertices chained together from source to sink. The code for this script is provided here:

generateSingleLineGraph.py

```
import sys
import time

if (len(sys.argv) != 2):
        print("Invalid arguments.\ngenerateSingleLineGraph <num_nodes>")
        sys.exit(1)

class Node:
        def __init__(self, nodeId, sourceCapacity, sinkCapacity):
                self.nodeId = nodeId
                self.sourceCapacity = sourceCapacity
                self.sinkCapacity = sinkCapacity

        def __repr__(self):
                return "n %i %i %i" % (self.nodeId, self.sourceCapacity, self.sinkCapacity)

class Edge:
```

```
        def __init__(self, node1, node2, forwardCapacity):
                self.node1 = node1
                self.node2 = node2
                self.forwardCapacity = forwardCapacity
                self.backwardCapacity = 0

        def __repr__(self):
                return "a %i %i %i %i" % (self.node1, self.node2, self.forwardCapacity, self.backwardCapacity)

NUM_NODES = int(sys.argv[1])

nodes = []
edges = []
nodeIndex = 0

newNode = Node(nodeIndex, 1, 0)
nodes.append(newNode)
nodeIndex += 1

for _ in range(0, NUM_NODES - 2):
        newNode = Node(nodeIndex, 0, 0)
        nodes.append(newNode)
        newEdge = Edge(nodeIndex - 1, nodeIndex, 1)
        edges.append(newEdge)
        nodeIndex += 1

newNode = Node(nodeIndex, 0, 1)
nodes.append(newNode)
newEdge = Edge(nodeIndex - 1, nodeIndex, 1)
edges.append(newEdge)
nodeIndex += 1

timestamp = time.strftime("%Y%m%d-%H%M%S")
```

```
output = open("graph-singleline-%i.bk" % NUM_NODES, 'w+')
output.write("p %i %i\n" % (len(nodes), len(edges)))
for node in nodes:
        output.write("n %i %i %i\n" % (node.nodeId, node.sourceCapacity, node.sinkCapacity))
for edge in edges:
        output.write("a %i %i %i %i\n" % (edge.node1, edge.node2, edge.forwardCapacity, edge.backwardCapacity))

print("Graph generated as graph-singleline-%i.bk" % NUM_NODES)
```

The second script is used to generate a graph that has a high level of branching and opportunity for parallelization. The script can be called from the command line as `generateVeryParallelGraph <branch_factor> <levels>`. The branch factor indicates how many vertices are generated for each vertex on each additional level. For example, `generateVeryParallelGraph 5 3` would create a `.bk` file that has 5 vertices connected to the source vertex, which branch into 25 vertices, which branch into 125 vertices that are all connected to the sink. In other words, the graph will have $\sum_{n=1}^{levels} branch_factor^n$ vertices. The code for this script is provided here:

`generateVeryParallelGraph.py`

```
import sys
import time
from random import randint

if (len(sys.argv) != 3):
        print("Invalid arguments.\ngenerateVeryParallelGraph <branch_factor> <levels>")
        sys.exit(1)

class Node:
        def __init__(self, nodeId, sourceCapacity, sinkCapacity):
                self.nodeId = nodeId
```

```
		self.sourceCapacity = sourceCapacity

		self.sinkCapacity = sinkCapacity


	def __repr__(self):

		return "n %i %i %i" % (self.nodeId, self.sourceCapacity, self.sinkCapacity)


class Edge:

	def __init__(self, node1, node2, forwardCapacity):

		self.node1 = node1

		self.node2 = node2

		self.forwardCapacity = forwardCapacity

		self.backwardCapacity = 0


	def __repr__(self):

		return "a %i %i %i %i" % (self.node1, self.node2, self.forwardCapacity, self.backwardCapacity)


BRANCH_FACTOR = int(sys.argv[1])

LEVELS = int(sys.argv[2])


nodes = []

edges = []

nodeIndex = 0


totalFlow = pow(BRANCH_FACTOR, LEVELS)


for i in range(1, LEVELS+1):

	for j in range(0, pow(BRANCH_FACTOR, i)):

		if (i == 1):

			newNode = Node(nodeIndex, totalFlow / pow(BRANCH_FACTOR, i), 0)

			nodes.append(newNode)

			nodeIndex += 1

		elif (i == LEVELS):

			newNode = Node(nodeIndex, 0, totalFlow / pow(BRANCH_FACTOR, i))
```

```
                        nodes.append(newNode)

                        newEdge = Edge((nodeIndex - 1)/BRANCH_FACTOR , nodeIndex, totalFlow / pow(BRANCH_FACTOR, i))

                        edges.append(newEdge)

                        nodeIndex += 1

                else:

                        newNode = Node(nodeIndex, 0, 0)

                        nodes.append(newNode)

                        newEdge = Edge((nodeIndex - 1)/BRANCH_FACTOR , nodeIndex, totalFlow / pow(BRANCH_FACTOR, i))

                        edges.append(newEdge)

                        nodeIndex += 1


timestamp = time.strftime("%Y%m%d-%H%M%S")


output = open("graph-veryparallel-%i-%i.bk" % (BRANCH_FACTOR, LEVELS), 'w+')

output.write("p %i %i\n" % (len(nodes), len(edges)))

for node in nodes:

        output.write("n %i %i %i\n" % (node.nodeId, node.sourceCapacity, node.sinkCapacity))

for edge in edges:

        output.write("a %i %i %i %i\n" % (edge.node1, edge.node2, edge.forwardCapacity, edge.backwardCapacity))


print("Graph generated as graph-veryparallel-%i-%i.bk" % (BRANCH_FACTOR, LEVELS))
```