CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 8, 4/22/2015. Scribed by Orren Karniol-Tambour, Hao Yi Ong, Swaroop Ramaswamy, and William Song.

8.1 Introduction

This lecture covers the basics of GraphX, Spark's graph-processing API, which extends the popular Pregel data flow paradigm to Spark. It also features an introduction to matrix computations using Spark.

8.2 How does Pregel work?

Pregel (a portmanteu of the words Parallel, Graph, and Google) is a data flow paradigm and system for large-scale graph processing created at Google to solve problems that are hard or expensive to solve using only the MapReduce framework. While the system remains proprietary at Google, the computational paradigm was adopted by many graph-processing systems, and many popular graph algorithms have been converted to the Pregel framework.

Pregel is essentially a message-passing interface constrained to the edges of a graph. The idea is to "think like a vertex" - algorithms within the Pregel framework are algorithms in which the computation of state for a given node depends only on the states of its neighbours. Figure 1 shows the Pregel paradigm's data flow model. A Pregel computation takes a graph and a corresponding set of vertex states as its inputs. At each iteration, referred to as a superstep, each vertex can send a message to its neighbors, process messages it received in a previous superstep, and update its state. Thus, each superstep consists of a round of messages being passed between neighbors and an update of the global vertex state. A few examples of Pregel implementations of graph algorithms will help clarify how the paradigm works.

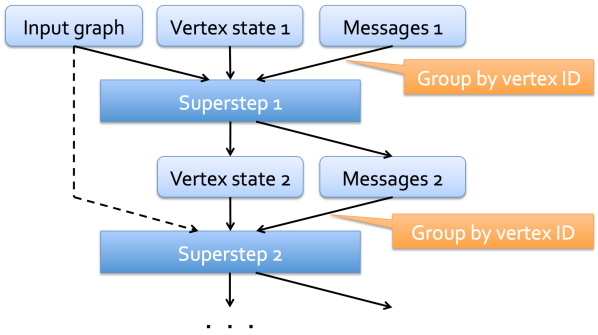

Figure 1: The Pregel paradigm's data flow model.

8.3 Examples of Graph Algorithm Implementations using Pregel

8.3.1 PageRank

PageRank is very intuitively implemented in the Pregel paradigm. At each superstep, each vertex updates its state with a weighted sum of PageRanks from all of its neighbors, processing the set of previous incoming messages. It then sends out an equal share of its new PageRank to each of its neighbors, sending out a set of outgoing messages. This continues until convergence.

Algorithm 1 PageRank

input: $G : Graph[V, E])$
while $err \geq \epsilon$ **do**
  for vertex i **do**
    $R[i] = 0.15 + 0.85 \sum_{j \in N_{\text{in}}(i)} M[j]$
    $M[i] = R[i]/|N_{\text{out}}|$
    Send $M[i]$ to all $N_{\text{out}}(i)$
  end for
  $err = |R - previousR|$
end while

8.3.2 Connected Components

To compute connected components in a graph using the Pregel paradigm, we can imagine letting each vertex 'infect' its neighbors. Each vertex is initialized with a unique ID, and at each iteration, each vertex sends its ID to its neighbors. Each vertex then overwrites its own ID with the max (or alternatively, min) ID it receives from its neighbors at each superstep. This continues until convergence. For computing the weakly connected components of an undirected graph, we assume $N_{in} = N_{out}$.

One caveat regarding the algorithm given above is that it may take $O(n)$ iterations to converge for some types of graphs. For example, consider the lollipop graph shown in Figure 2. If you start at one of the end nodes on the right, then at each iteration, you will only infect one node. For graphs which have n in the order of millions or billions, this may be unacceptable. However, in most social networks and web graphs, this does not really happen, so this is not a problem for most practical applications.

Figure 2: An example of a lollipop graph.

Algorithm 2 Strongly Connected Components

input: $G : Graph[V, E]$)
$stale = 0$
while $stale \neq |V|$ **do**
  $stale = 0$
  for vertex i **do**
    if $ID[i] = \max_{j \in N_{in}(i)} ID[j]$ **then**
      $stale = stale + 1$
    else
      $ID[i] = \max_{j \in N_{out}(i)} ID[j]$
      Send $ID[i]$ to all N_{out}
    end if
  end for
end while

In many real-world graphs, few nodes have very high degree neighborhoods - for example, a node representing the followers of Katy Perry on Twitter will have tens of millions of neighbors. Critically, the Pregel framework does not require that we able to fit the incoming messages from the neighborhood of a vertex in a single machine. The Pregel paradigm allows us to distribute the computation for high degree nodes, as we'll see in GraphX's notion of 'vertex cutting'.

8.4 GraphX

GraphX is Spark's graph processing and computation API, which implements the Pregel paradigm within Spark using RDDs. To implement Pregel, separate RDDs are created to represent the graph, the global vertex state, and the messages from each vertex to its neighbors. Note that because RDDs are immutable objects in Spark, new RDDs representing the vertex state and outgoing messages are created at each superstep of the computation. A `groupByKey` operation is used to perform each superstep in the computation. We could use a `FlatMap` and a `reduceByKey` but this can be very expensive when only 1-2 vertices send messages in a superstep. Since Pregel algorithms perform supersteps until convergence, later supersteps typically involve updates on only a few vertices. If we used a `FlatMap` and a `reduceByKey`, we would have to shuffle the whole data just to update a few vertices. This is very inefficient when we consider algorithms where very few vertices send messages at each superstep, so we use `groupByKey` instead.

In GraphX, every vertex has an ID and a property associated with it, which can take the form of any tuple. Similarly, each edge is associated with an ID for the source vertex, ID of the destination vertex and a property.

8.4.1 Map-Reduce Triplets

A triplet contains a source vertex, a destination vertex and the edge connecting these two. As it turns out, computing triplets, or joining edges and vertices, is a basic computation used in many useful graph algorithms. However, if done naively, computing triplets would require two joins on the vertex RDD and the edge RDD, which is very expensive. GraphX implements an optimized triplets computation and provides us with a triplets operator for joining vertices and edges in its API. The Triplets operator is composed of two inputs, a map and a reduce function. The Map function takes as its input a triplet and returns an object. The Reduce function performs a reduce operation on these objects.

8.4.2 Triplets Example : Oldest Follower

In the oldest follower problem, we try to to find the oldest follower for each node in a directed graph. We can compute this using MapReduce triplets as follows to accomplish this

```
val oldestFollowerAge = graph.mrTriplets(
    e=> (e.dst.id, e.src.age),// Map
    (a,b)=> max(a, b) // Reduce
).vertices
```

8.4.3 Other Operations

Other operations natively available in the GraphX API include PageRank, strongly connected components, triangle counting etc. GraphX also provides a Mask operator, which given a graph, returns a sub-graph with specified vertices masked. For a complete list of GraphX operations, refer to the GraphX programming guide.

8.4.4 Optimization

Graphs derived from natural phenomena (e.g., social networks) tend to follow skewed power-law distributions. In general, partitioning algorithms based on edge cutting typically fare poorly on such graphs due to high-degree vertices present in such graphs. In comparison, vertex cut partitioning schemes have been observed to perform well on many large natural graphs [4]. Thus, to distribute computation workload evenly GraphX partitions the graph with vertex-cut. This is different from, say, Giraph, which uses edge-cut partitioning.

In brief, vertex-cut splits high-degree vertices across partitions and evenly assigns edges to a machine in a way that minimizes the number of times each vertex is cut. As explained in lecture, GraphX represents a graph using three RDDs: an edge collection and two vertex collections. Being separate constructs, these RDDs do not need to be stored on the same machine. Each vertex partition contains a routing table RDD and a datatable RDD. The routing table is a logical map from a vertex id to the set of edge partitions that contains adjacent edges. The datatable RDD simply stores vertex data in the form of vertex (id, data) pairs. The edge RDD stores the adjacency structure and edge data. Each edge is represented as a tuple consisting of the source vertex id,

destination vertex id, and user-defined data as well as a virtual partition identifier (pid). Note that the edge table contains only the vertex ids and not the vertex data. The edge table is partitioned by the pid. This technique incurs some overhead due to the joins and aggregation needed to coordinate vertex properties across partitions containing adjacent edges.

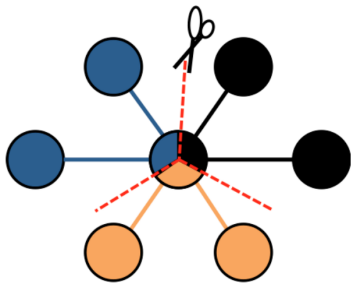

Figure 3: An illustration of Vertex Cut.

During graph computations, we often need to assemble an edge with the data associated on both vertices. GraphX uses a 3-way relational join to bring together the source vertex data, edge data, and target vertex data:

```
VertexDataTable v
JOIN VertexMap vm ON (v.id=vm.id)
RIGHT OUTER JOIN EdgeTable e
   ON (e.pid=vm.pid AND (e.src=v.id OR e.dst=v.id))
   WITH PARTITIONER edgeTable.partitioner ON pid
```

The joins to obtain triplets are fairly straightforward and use a partitioner. As the edge table is often much larger than the vertex data table, the partitioner is used to ensure the join site would be local to the edge table. This allows GraphX to shuffle only the vertex data and avoid moving any of the edge data. To minimize communication, GraphX co-partitions the two tables so the first join can be done locally. The resulting table from the 3-way join presents an edge-centric view of the graph, with each tuple containing the edge data, source vertex data, and the target vertex data. More information can be found in [5].

8.5 Computations on Matrices with Spark

8.5.1 Distributed Matrices

In Spark, matrices are typically stored broken up for storage in three different ways:

- By entries (CoordinateMatrix): stored as a list of $(i, j, value)$ tuples
- By rows (RowMatrix): each row is stored separately (e.g. Pagerank)

- By blocks (BlockMatrix): by storing submatrices of a matrix as dense matrices, block matrices can take advantage of low-level linear algebra library for operations like multiplications.

8.5.2 RowMatrix × LocalMatrix

When multiplying a RowMatrix with a small local matrix, we broadcast the entire small matrix to each machine that contains different parts of the RowMatrix and perform multiplications on each machine. Currently, Spark uses BLAS level 1 optimization, which optimizes for vector-vector multiplications during the multiplication.

$$\text{rows are distributed} \left\{ \begin{bmatrix} — & r_1^T & — \\ — & r_2^T & — \\ & \vdots & \\ — & r_n^T & — \end{bmatrix} \right. \begin{bmatrix} | & | & & | \\ l_1 & l_2 & \dots & l_m \\ | & | & & | \end{bmatrix}$$

8.5.3 CoordinateMatrix × CoordinateMatrix

CoordinateMatrix is used to represent sparse matrices, and is stored as a list of (row, column, value) entries. To perform matrix multiplication of two Coordinate Matrix elements $C = AB$, one can summarize the procedure as follows:

Algorithm 3 CoordinateMatrix Multiplication

input: $A : \{(i, j, A_{ij}) | A_{ij} \neq 0\}, B : \{(i, j, B_{ij}) | B_{ij} \neq 0\}$
$J \leftarrow$ Join A, B on $a.j$ and $b.i$ for $a \in A, b \in B$
$M \leftarrow$ For each $j \in J$, map it to (key, value) where key=$(a.i, b.j)$ and value=$a.val \times b.val$
$C \leftarrow$ Reduce M with "+"

Unfortunately, Spark does not have CoordinateMatrix multiplication implemented in the current library. One possible implementation with Scala, when assuming the matrices are stored as `RDD[MatrixEntry(i, j, value)]` is shown below

```
import org.apache.spark.mllib.linalg.distributed._

val n = 10          // one dimension of the matrix
val range = sc.parallelize({1 to n * n})
// Generate two random sparse matrices of size nxn
val A = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))
val B = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))

// Perform multiplication
val C = A.map(e => (e.j, e)).join(B.map(e => (e.i, e)))
         .map(p => ((p._2._1.i, p._2._2.j), p._2._1.value * p._2._2.value))
         .reduceByKey(_ + _).map(p => MatrixEntry(p._1._1, p._1._2, p._2))
```

Effectively, for each $1 \leq i \leq n$, we first join all entries of matrix A on the i-th column with the entries of matrix B on the i-th row, which would create a Cartesian product of two sets of entries for each i. The resulting set contains all possible pairs of entries that would've been multiplied together during a normal matrix multiplication, and each pair of entries is keyed by their shared dimension during the dot product operation. We then remap each element in this set by its position in the result matrix and change its value to the product of the two entries and then reduce each result position with the addition operator. This effectively simulates the dot product operation. Finally, we remap the result to the desired format.

8.5.4 BlockMatrix × BlockMatrix

In some cases, we'd like to multiply two dense matrices for which the rows and columns may themselves be too large to fit in memory on a single machine. By partitioning our matrices into blocks that do fit on a single machine - encoding each one as a BlockMatrix - and performing multiplication on their partitions, we can manage to perform matrix computation on the larger matrices. Using BlockMatrix, we also have the ability to push down the smaller block matrix multiplications to the CPU/GPU directly using Basic Linear Algebra Subprograms (BLAS) routines - as mentioned above, Spark currently uses BLAS level 1 for matrix multiplication. To perform block matrix multiplication, we partition both matrices appropriately so their blocks are equally sized within each matrix, aligned in size across matrices, and so that a single block from each matrix fits together on a single machine. Following partitioning, block multiplication proceeds similarly to coordinate multiplication. Each matrix is flatmapped to produce a list of blocks for multiplication - each block in the first matrix A is effectively copied as many times as the number of columns in the second matrix B, and each block in B is effectively copied as many times as the number of rows in A. Following the flatmap, a cogroup is used to send pairs of complementary blocks that will need to be multiplied to an individual machine, where the multiplication is pushed down to the CPU/GPU level using BLAS. Finally, results of the individual block multiplications corresponding to each block entry in the resulting matrix are sent to the same machine with ReduceByKey and summed up. A simplified version of the Spark code for block matrix multiplication is presented below:

```
def multiply(other: BlockMatrix): BlockMatrix = {

      // Get partitions
      val resultPartitioner = GridPartitioner(numRowBlocks, other.numColBlocks,
        math.max(blocks.partitions.length, other.blocks.partitions.length))

      // Each block of A must be multiplied with the corresponding blocks
      // in each column of B.
      val flatA = blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(other.numColBlocks)
```

```
            (j => ((blockRowIndex, j, blockColIndex), block))
      }

      // Each block of B must be multiplied with the corresponding blocks
      // in each row of A.
      val flatB = other.blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(numRowBlocks)
                (i => ((i, blockColIndex, blockRowIndex), block))
      }

      // Cogroup and multiply block pairs
      val newBlocks: RDD[MatrixBlock] = flatA.cogroup(flatB, resultPartitioner)
        .flatMap { case ((blockRowIndex, blockColIndex, _), (a, b)) =>
          if (a.nonEmpty && b.nonEmpty) {
            val C = b.head match {
              case dense: DenseMatrix => a.head.multiply(dense) // Uses BLAS 1
              case sparse: SparseMatrix => a.head.multiply(sparse.toDense)
            }
            Iterator(((blockRowIndex, blockColIndex), C.toBreeze))
          } else {
            Iterator()
          }
      }
      // Sum up matrices for each block entry of C
      .reduceByKey(resultPartitioner, (a, b) => a + b)
        .mapValues(Matrices.fromBreeze)
}
```

References

[1] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[2] G. Malewicz and M. Austern and A. Bik and J. Dehnert and I. Horn and N. Leiser and G. Czajkowski. *Pregel: a System for Large-Scale Graph Processing*. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010.

[3] J. Gonzalez and R. Xin and A. Dave and D. Crankshaw and M. Franklin and I. Stoica. *Graphx: Graph Processing in a Distributed Dataflow Framework*. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.

[4] J. E. Gonzalez et al. *Powergraph: Distributed graph-parallel computation on natural graphs.*. OSDI'12, USENIX Association, pp. 1730.

[5] R. S. Xin et al. *GraphX: A resilient distributed graph system on Spark.* Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, New York, USA.