Data Intensive Computing – Review Questions 5

Adriano Mundo, Riccardo Colella (Group 29)

1. What are the differences between vertex-centric and edge-centric graph processing models?

Both models are implementations of the scatter-gather programming model for graph computation. The main difference between them is that *vertex-centric* iterates over vertex, while *edge-centric* iterates over edges.

- The *vertex-centric model*:
  - during the scatter phase, for all vertices that need to scatter updates, it sends updates over outgoing edges of the considered vertex.
  - during the gather phase for all vertices that have updates, it applies updates from inbound edges of the considered vertex
  - *vertex-centric* scatter-gather trade-off: $\frac{EdgeData}{RandomAccessBandwith}$

This approach represents a trade-off between sequential and random access, favoring a small number of random access through the edge index over streaming a large number of potentially unrelated edges.

- The *edge-centric model*:
  - during the scatter phase, for all the edges, it sends update over the considered edge
  - during the gather phase, for all the edges that have updates, it applies the update to the destination of the considered edge
  - *edge-centric* scatter-gather trade-off: $\frac{ScattersxEdgeData}{SequentialAccessBandwith}$

This approach allows to stream the set of edges from storage and to avoid random accesses into it, that is often advantageous since a common property for graphs is to have the edge set much larger than the vertex set and therefore the access to edges and updates dominates the processing cost. Doing so comes, however, at the cost of random access into the set of vertices, that can be mitigated using streaming partitions over it, so that each partition fits in high-speed memory. This is what X-Stream does, partitioning also the set of edges such that they appear in the same partition as their source vertex and there is no need to sort the edge list.

2. Explain briefly when we should use pure graph-processing platforms (such as GraphLab), and when we need to use platforms such as GraphX?

GraphLab is a high-performance framework exclusively built for graph processing. It should be used for PageRank and community detection algorithms on graphs with billions of vertices and edges. It allows iterative asynchronous computation. It is the preferred choice when high performances are required. On the other side, graphs are not the only part of the larger analytics process which often combines graphs with unstructured and tabular data. That's why *GraphX* is very useful.

GraphX is a graph processing framework built on top of Spark that unifies the advantages of *GraphLab* with those provided by general purpose data-flow frameworks such as Spark, hence enabling a single system to address the entire analytic pipeline. Differently from existing graph processing systems, it enables the composition of graphs with unstructured and tabular data allowing the same physical data to be viewed both as a graph and as collections without data movement or duplication. Moreover, it introduces a range of optimizations in how graphs are encoded as collections as well as in the execution of the common data-flow operators. Moreover, it achieves low cost fault tolerance by leveraging logical partitioning and lineage. In conclusion, *GraphX* should be used on Natural graphs (those derived from natural phenomena), since it can achieve performance parity with specialized graph processing systems while preserving the advantages of a general-purpose dataflow framework.

3. Explain briefly how the command `pregel` works in GraphX?

It is a method to implement iterative computation. The declaration of the command is the following: `graph.pregel(list1)(list2).`

The first list contains configuration parameters such as the initial message, the maximum number of iterations and the edge direction in which to send messages.
The second list contains the user defined functions which are Gather (`mergeMsg`), Apply (`vprog`) and Scatter (`sendMsg`).

This method implements the GAS decomposition by iteratively composing the join and group-by stages with data-parallel map stages. Each iteration begins by executing the join stage to bind active vertices with their outbound edges. Using the triplets view, messages are computed along each triplet in a map stage and then aggregated at their destination vertex in a group-by stage. Finally, the messages are received by the vertex programs in a map stage over the vertices. The dataflow embedding of the Pregel abstraction demonstrates that graph-parallel computation can be expressed in terms of a simple sequence of join and group-by dataflow operators. Thus, having expressed message computation as an edge-parallel map followed by associative aggregation it is possible to mitigate the cost of high-degree vertices.

4. Assume we have a graph and each vertex in the graph stores an integer value. Write three pseudo-codes, in Pregel, GraphLab, and PowerGraph to find the maximum value in the graph.

Pregel (edge-cut partitioning): the applications run in a sequence of iterations, called supersteps, during a vertex can: reads a message received in the previous superstep, sends messages to other vertex and modifies its state. Hence, communication happens through sending messages. The idea of the algorithm is to receive the messages from the previous iteration, go through the messages and update if the value is greater than the current. Then, if the value is update vote to halt, otherwise send messages to the neighbors.

```
Pregel_maxValue(curr, messages):
    temp = curr

    for each message m in messages:
           if (m > curr): curr = m

    if (temp == curr): vote_to_halt()
    else:
           for each neighbor n in neighbors:
                  send_message(n, curr)
```

GraphLab (edge-cut partitioning): it allows asynchronous iterative execution. A vertex can read and modify any of the data in its scope (shared memory). The idea is to obtain data from all the neighbors and then to signal the neighbors if there is a change. The algorithms signal the neighbors with an outgoing edge from the current one.

```
GraphLab_maxValue(curr):
   temp = curr

   foreach(inc in inc_neighbors(curr)):
          if (inc > curr): curr = inc

   if (temp != curr):
          foreach (out in out_neighobors(curr)):
                 signal(o, curr)
```

PowerGraph (vertex-cut partitioning): it factorizes local vertices functions into gather, apply and scatter phases (GAS) asynchronously or synchronously. The main idea is to accumulate information from neighborhood, then the sum accumulates the partial results and in the Apply phase the value from the Gather (biggest) is applied to the current node, center vertex. Finally, it notifies adjacent nodes.

```
PowerGraph_maxValue(i):
   temp = curr

   Gather(j -> curr): return j

   Sum(a,b):
          if (a > b): return a
          else: return b

   Apply(curr, biggest):
          if (biggest > curr): curr = biggest

   Scatter(curr -> j):
          if (temp != curr): activate(j)
```