Distributed Graph Processing

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from online materials:
http://www.slideserve.com/fynn/pregel-a-system-for-large-scale-graph-processing

Learning objectives

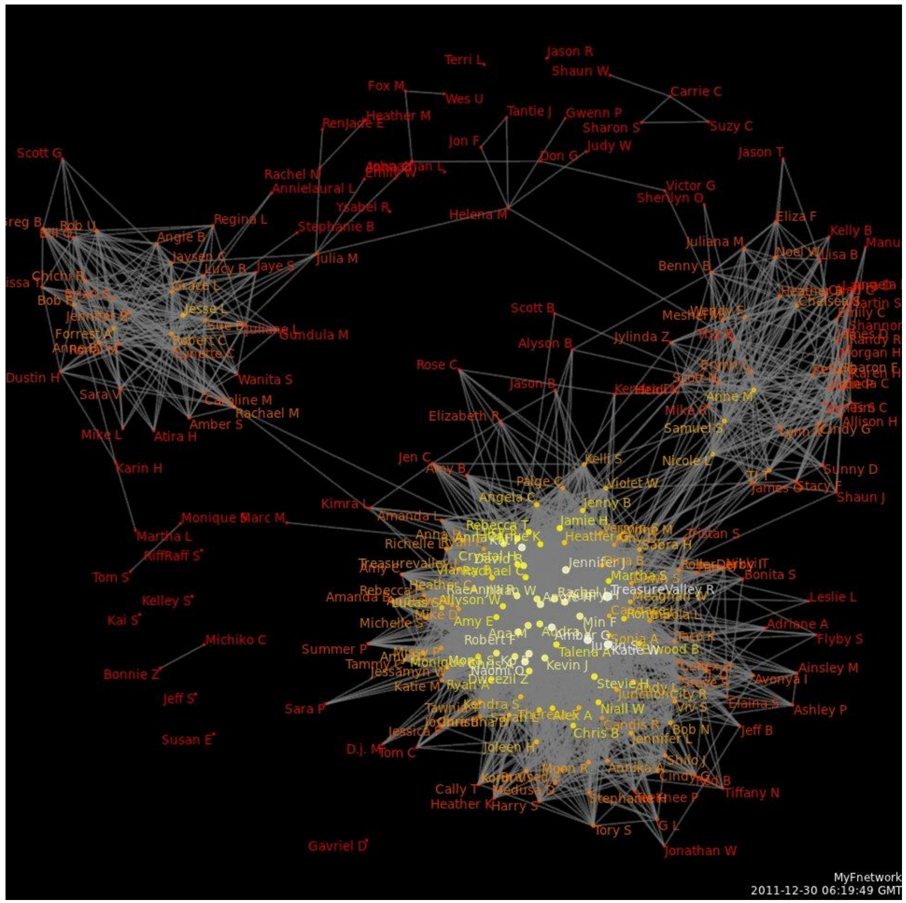
Introduction to distributed graph processing:

- graph data sets
- the Google Pregel model
- Pregel examples: max, PageRank, SSSP

Motivation

- Many important data sets look like graphs: web hyperlinks, social networks, protein interaction networks, transportation networks (roads, highways, flights, etc.)
- Some graph data sets are very large (e.g., 1+ billion vertices in Facebook social network graph), which makes graph computations very time-consuming.
- Single-machine solutions offer multi-core scalability but cannot harness together hundreds of commodity servers like MapReduce.
- MapReduce itself is not very good at graph algorithms, which generally require multiple MR stages.
- Apache Spark is more comfortable at PageRank, but its basic programming model is not a natural fit for graphs.

Example: subset of Facebook's social network



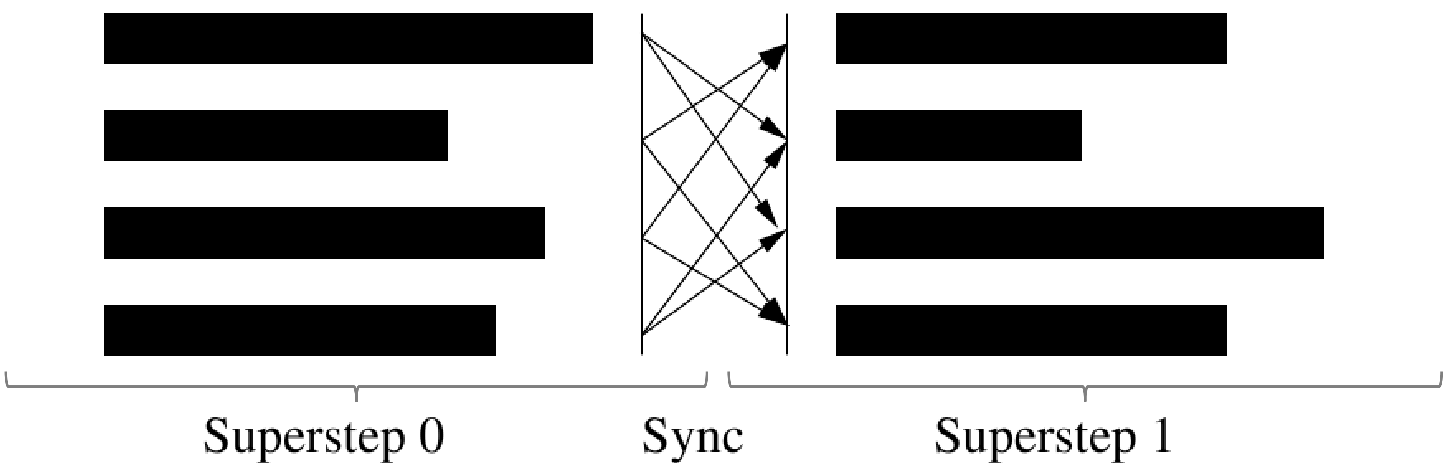
source:
https://upload.wikimedia.org/wikipedia/commons/9/90/Kencf0618FacebookNetwork.jpg

Google's solution: Pregel

- Pronounced Pregel like bagel, not "pre-gel" like hair gel.
- **Master/worker** model similar to Hadoop and Spark.
- Each worker is responsible for a **vertex partition**, which is a subset of a directed graph's vertices.
- Model of computation is **vertex-centric** ("think like a vertex").
- The framework maintains the following state for each vertex:
  - a problem-specific value (e.g., the PageRank of a vertex)
  - a list of messages sent to the vertex
  - a list of outgoing edges
  - a binary active/inactive state

Google's solution: Pregel (cont.)

Pregel exemplifies the Bulk Synchronous Parallel (BSP) model proposed by Leslie Valiant. The computation is organized into synchronous rounds or iterations, called **supersteps**, driven by the master. Workers compute asynchronously within each superstep, and communicate only between supersteps.



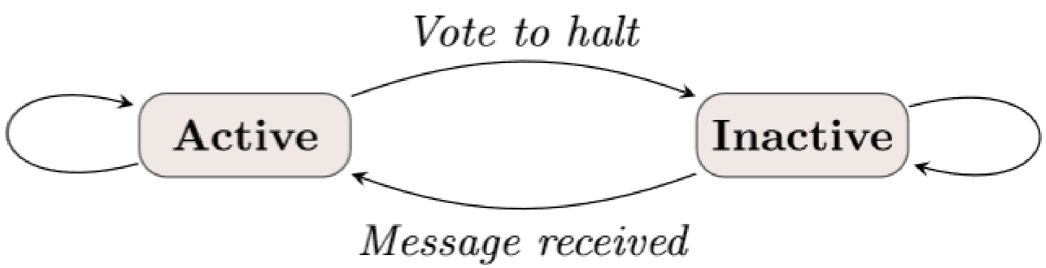
source: http://www.multicorebsp.com/images/algorithm.gif

Google's solution: Pregel (cont.)

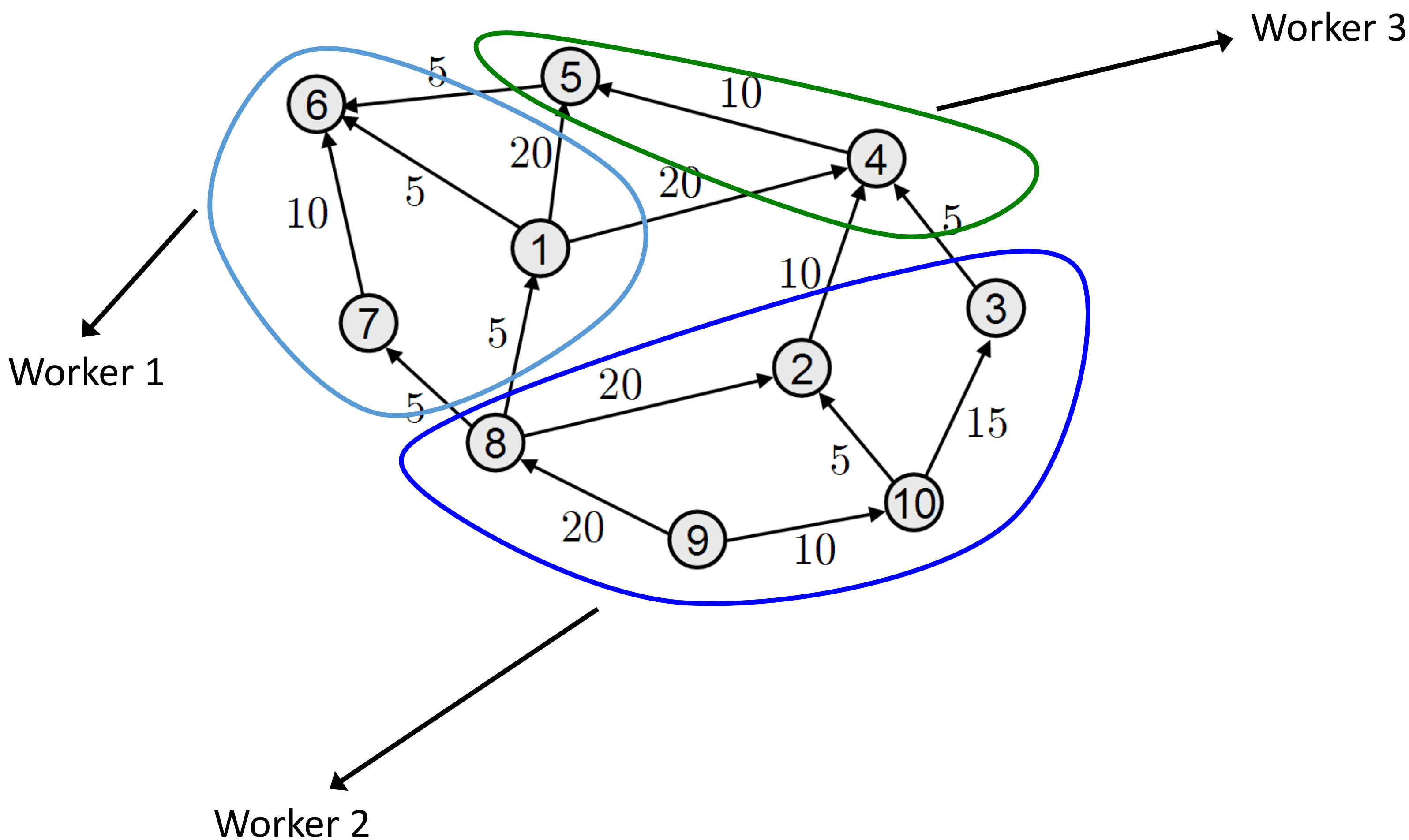
In each superstep the following actions happen:

- workers asynchronously execute a user-defined function on each vertex
- vertices can receive messages sent in the previous superstep
- vertices can modify their value, modify the values of their edges, as well as add/remove edges
- vertices can send messages to be received in the next superstep
- each vertex may also deactivate itself (i.e., vote to halt)
- an inactive vertex is reactivated when it receives a message

The distributed execution stops when **all vertices are inactive** and there are **no more messages to process**.



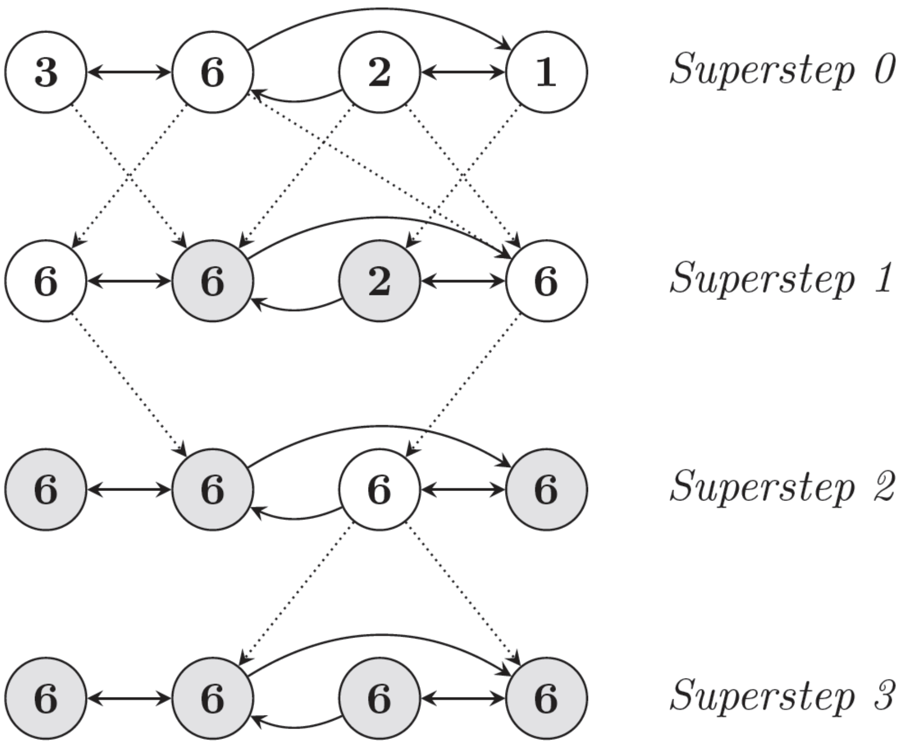
Example of vertex partitions



Initialization

- The master assigns a section of the input to each worker, similarly to the way input splits are determined in Hadoop.
- Vertex ownership is determined not by the input split but by a partitioner, which by default is a simple hash function over vertices. This ensures a fairly even distribution of data (i.e., vertices) across workers, but does not always balance the computation, which is edge-dependent.
- Each worker reads its section of the input, stores vertices that belong to it, and forwards the remaining vertices to the appropriate workers.
- The user can override the default partitioning scheme to exploit locality (e.g., by co-locating graph components or dense subgraphs).

Example: find max vertex label




```
Superstep 0:
  for each neighbor v
    send_message(v, val)

Superstep ≥ 1:
  i_val := val
  for each message m
    if m > val then val := m
  if i_val == val then
    vote_to_halt
  else
    for each neighbor v
      send_message(v, val)
```

Example: PageRank
(without convergence test)

```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Combiners and aggregators

- Pregel supports **combiners**, similarly to Hadoop, which reduce the amount of data exchanged over the network and the number of messages. This trades CPU cycles against network I/O.
- Combiners in general are applicable when the function applied at each vertex is **commutative** and **associative** (e.g., min, max, sum). They are user-defined and must be enabled explicitly.
- Pregel **aggregators** are used to compute aggregate statistics from vertex-reported values. Workers aggregate values from their vertices during each superstep. At the end of each superstep, the values from the workers are aggregated in a tree structure, and the value from the root of the tree is sent to the master. The master shares the value with all vertices in the next superstep.
- Aggregators can be used to evaluate a convergence criterion in an iterative computation, such as PageRank.

Example: single source shortest paths (SSSP)

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

Example: single source shortest paths (SSSP)

```
class MinIntCombiner : public Combiner<int> {
  virtual void Combine(MessageIterator* msgs) {
    int mindist = INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    Output("combined_source", mindist);
  }
};
```

Fault tolerance

- The fault tolerance mechanism is similar to checkpointing in a database. The master tells the workers to save their state to persistent storage at the start of a superstep. This state includes: vertex values, edge values, and a list of incoming messages. The master also saves any aggregator values, if applicable. The frequency of checkpoints is user-determined.
- When the master detects one or more worker failures, it rolls back all workers to the most recent checkpoint, and the computation is repeated from that checkpoint.
- More efficient recovery mechanisms are possible in which only the failed workers revert to the checkpoint. This requires <u>deterministic</u> replay of any messages sent to those workers at each superstep since the checkpoint.

Summary

- Computations on large graphs are expensive and benefit from parallelization.
- Pregel is a model for scalable distributed graph computation.
- Pregel-like APIs are supported in open-source frameworks such as Apache Giraph and Apache Spark.
- The relative merits of centralized versus distributed graph computation are debatable. For data sets that fit into the main memory of a single machine, centralized solutions (e.g., GraphChi) tend to be much more efficient than distributed frameworks.

Appendix: Spark Pregel API

```
import org.apache.spark.graphx._
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// A graph with edge attributes containing distances
val graph: Graph[Long, Double] =
    GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
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
    )
println(sssp.vertices.collect.mkString("\n"))
```

source: http://spark.apache.org/docs/latest/graphx-programming-guide.html#map-reduce-triplets-transition-guide-legacy