```
Distributed Programming Models -- Graph-based

Models that we've learnt so far
- data flow (mapreduce, dryad, spark)
  * typed key-value collections
  * high-level operators: map, reduce, join
  * immutable data (create new collections)
  * computation represented as a dataflow-graph of operators, each of which comp
osed of many parallel tasks
  * fine-grained failure recovery of individual tasks

  + performance cost is explicit (reduce/join typically require shuffling)
  + fine-grained recovery is good for big cluster
  + can deal with both in-core and out-of-core comutation
  - immutability --> not the most efficient
  - somewhat unintuitive to program
  - (cannot deal with asynchronous computation) to be added later

- distributed shared memory (threadmarks)
  * un-typed global state
  * read/write mutable state (think how you program with pthread)
  * requires synchronization primitives: barrier, locks etc.
  * coarse-grained failure recovery (periodic checkpointing of state and re-exec
ute)

  + intuitive to program
  + if all goes well, can be very efficient
  - performance cost is hidden, easy to write super slow programs
  - coarse-grained recovery
  - only good for in-core computation


Today: graph-based programming model
   A completely new programming model or a specialization?

Why graph-based model:
* many computations work on graphs.
  Example graphs: web graph, follow graph, friendship graph, user-like-movies gr
aph
  Example computation: PageRank, connected-component, shortest path

The Push-based graph model (Pregel):
  - User-defined vertex program (function) takes as input a "combined" message
    it can "SendMsg" to its out-neighbors:
  - User-defined combiner to aggregate multiple messages to the same vertex

        class VertexProgram {
                vector<node_id> outNbrs() {
                        //return list of my outgoing nbrs
                }
                void sendMsg(node_id nbr, Message msg) {
                        //...
                }
                virtual doVertex(Message msg) {
                        //user-defined vertex function
                }
                virtual Message combine(Message accum, Message m1) {
                }
                node_id my_id_;
        }

        class AppVertexProgram : VertexProgram {
                doVertex()
                combine(..., ...)

                //one can add other app state
        }

Execution model:
```



```
* Bulk synchronous execution
  while (some node is active) {
        foreach vertex in Graph {
                if vertex.active {
                        doVertex()
                }
        }
        global_barrier;
  }
* Asynchronous execution
  At each machine:
    while (some node is active) {
                lock outgoing neighbors
                doVertex
                unlock
        }
  //no global barrier


Exercise: write two programs in this interface
   PageRank
   ConnectedComponent

Performance:
    (draw pic of Figure 4, PowerGraph)
    Partition-by-vertex: randomly assign each vertex to one of the p machines
        Communication overhead: # of edges cut across machines
        + straightforward to implement
        - load-imbalance (nodes w/ large in-degrees or more out-degrees)
        - most edges got cut across machines --> lots of communication

        PageRank/CC communication cost: Num_edges per iteration

PowerGraph's idea: partition-by-edge

How to implement this feature?

* First, a node is not aware of all its outgoing neighbors, thus,
        - users cannot manually loop over outgoing neighbors in doVertex
        solution: factor out message sending in a separate user-defined function

        - virtual Message scatter(node_id nbr) {
          }
   //
        while (some node is active) {
                parallel foreach vertex v in Graph {
                if v.active {
                        v.doVertex()
                        foreach u of vertex.out_nbrs
                                v.scatter(u)
                }
        }
        global_barrier;
  }

* Second, since scatter for the same (source) vertex needs to run on different m
achines  ==> mirror vertex state
        //
        while (some node is active) {
                parallel foreach vertex v in Graph {
                if v.active {
                        v.doVertex()
                        foreach u of vertex.out_nbrs
                                v.scatter(u) //messages sent to master of each v
ertex replica
                }
        }
        global_barrier;
        parallel foreach mirror of a vertex {
```

```
                synchronize mirror with master
        }
        global_barrier;
  }

  Communication overhead:  # of total vertex replicas

* The claim: Communication of random partition-by-edge < Communication of random
 partition-by-vertex for power-law graphs
  In the paper, they proposed a greedy assignment strategy to minimize vertex re
plication

  Look at Figure (9) Fan-in versus fan-out:
  The intuition why vertex cut is better (for large fan-out):
  A node with lots of out-degrees to randomly chosen other nodes
  Under partition-by-vertex, communication cost \approx out-degree
  Under partition-by-edge, essentially, send this node's state to all other mach
ines, run scatter at those machines
        communication cost \approx #_of_machines

Question: Is Graph-model a fundamental abstraction or one that can be built on
top another fundamental abstraction?

Build Pregel on the following data flow operator?
Spark-like operators: map, filter, groupBy, reduce, join, count
Join is different from that in the Spark paper, more similar to Dryad:

    T_new = join T and T' on T.f1 = T'.f2

V: a collection of <id, halt, msg, p>  V is some user-defined state, msg is the
combined message destined for a node
E: a collection of <src, dst> msg is the message flowing along some edge

while (1) {
        active_v = V.filter(isActive())
        if (active_v.count() > 0) {
                EV = join active_v and E on active_v.id = E.src (EV.src, EV.dst,
 EV.src_state, EV.msg)
                combined_msgs = EV.map(scatter()).groupBy(EV.dst).reduce(combine
())
                vertexes = (join active_v and combined_msgs).map(doVertex)
        }
}

Certain optimizations are needed for the data-flow performance to be competitive
:
   Do not always materialize EV

Are all the optimizations for PowerGraph applicable for this version?
1. Vertex-cut (i.e. edge partition)
   Depends on how join is implemented for v and E
   If E is partitioned by src_id, then corresponds to vertex partition
   If E is partitioned by src_id and dst_id, then corresponds to edge partition
   The value of V must be multi-casted to all machines which contains that parti
cular vertex

2. asynchronous execution??
```