Data Management in the Cloud

PREGEL AND GIRAPH

Thanks to Kristin Tufte

1

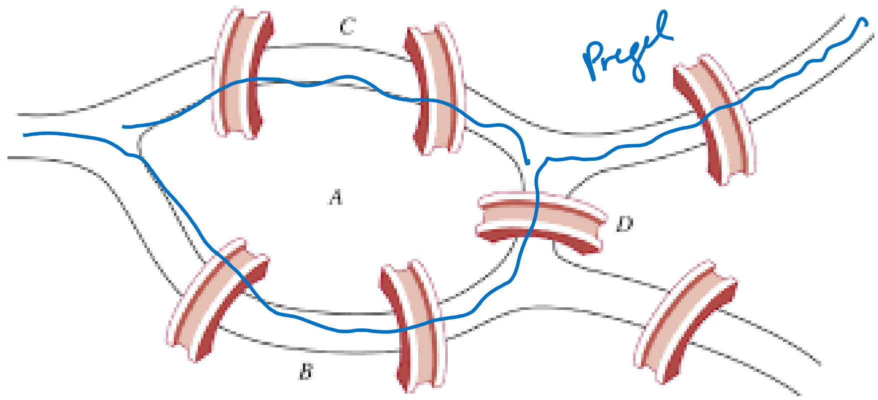
Why Pregel?

- Processing large graph problems is challenging
- Options
  - Custom distributed infrastructure
  - Existing distributed computing platform (MapReduce is ill-suited to some graph processing)
  - Single-computer graph algorithm (not scalable)
  - Existing parallel graph system (not fault-tolerant)
- Pregel…
  - Vertex-centric superstep framework
  - Focuses on independent local actions
    - Well-suited for distributed implementation
  - Fault-tolerant
  - C++ API

2

Why "Pregel"?

Seven bridges of Königsberg
Can you find a path that crosses each bridge exactly once?



3

Efficient Processing of Graphs is Challenging

- Poor locality of memory access
- Little work per vertex
- Changing degree of parallelism over the course of execution
  "Long tail" of graph searches

4

How is Pregel Different from Neo4j?

- Neo4j is a graph storage system
  Has some support for running graph searches
- Pregel holds its graph in main memory
  Focus on parallel graph algorithms

5

Model of Computation

- Input: Directed graph, each vertex has a unique id
- Computation: BSP Bulk Synchronous Processing
  - Input/initialization
  - Sequence of Supersteps (global synch between Supersteps)
  - Termination/output

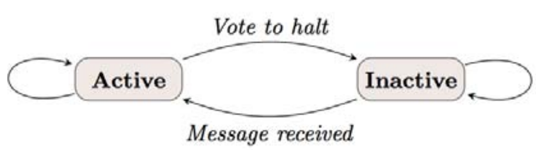


Figure 1: Vertex State Machine

6

Model of Computation II

State:

- Set of vertexes and directed edges
- Each vertex stores
  - A (possibly complex) value
  - A list of outgoing edges plus (optional) state for each

- Example: Facebook users
  - One vertex per user, value is **`<userID>`**
  - One out edge to each friend:
    Edge value is **`<#likes, #comments>`**

7

Model of Computation II

- Computation proceeds in Supersteps…
  - In a Superstep vertexes compute in parallel, execute the same user-defined function
- In a Superstep, a vertex can:
  - Receive messages sent to it in previous Superstep
  - Modify the value of the vertex
  - Modify values of outgoing edges
  - Send messages to other vertexes to be received in next Superstep
  - Add/remove vertexes and edges
- Note: edges may have value, but have no associated computation
- Note: pure message-passing model, no remote reads or shared memory assumptions

8

Example: Maximum Vertex Value

- Each vertex v has a LocalVal (an integer) and GlobalMax (initally = LocalVal)
- **For each** incoming message msg:m
```
    GlobalMax ←
        max(GlobalMax, m)
```
- **If** GlobalMax changed **then**
```
    for each outgoing edge (v, u)
        send msg:GlobalMax to u
```

9

Maximum Vertex Value

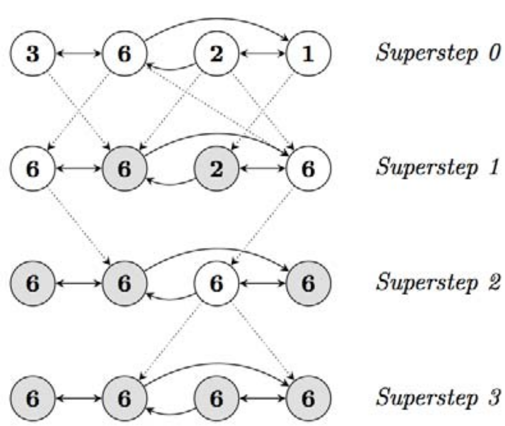


Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

Figure credit: Pregel: A System for Large-Scale Graph Processing

10

Termination

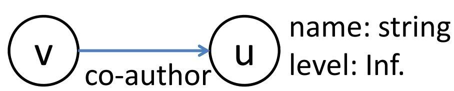
At the end of a Superstep, a vertex can vote to halt

If all vertexes vote to halt, then stop

Example: If no change in `GlobalMax`, vote to halt

11

Level from Vardi to Hellerstein



- `Set Vardi’s level to 0`
- `[`*`Process incoming messages`*`]`
- `[`*`Send outgoing messages`*`]`
- ```
  If level not Inf
     and name = “Hellerstein”
     then output level
  ```

12
  ```

Model of Computation - Comments

- Vertexes and edges kept on machine that performs computation
- Chained map-reduce would require passing the entire state of the graph from one stage to the next
- There is also provision for aggregation across all vertexes
  - Compute current error
  - Collect results

13

C++ API

- Implement by subclassing the Vertex Class
- Functions possible:
  - Compute()
  - GetValue() / MutableValue()
  - Inspect and modify values of output edges
- Note: no data races, each vertex can only modify its outgoing edges
  - Note: limited state (values associated with a vertex and its edges) – simplifies computation, graph distribution and failure recovery
    Checkpoint: Local state of vertexes plus current rounf of messages

14

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

Figure 3: The Vertex API foundations.

Figure credit: Pregel: A System for Large-Scale Graph Processing

15

Message Passing

- Messages sent from vertex to vertex
  - Consist of: message value and ID of destination vertex
- Vertex can send as many messages as it wants in a Superstep

  Normally, messages sent to neighbor vertexes (but can send to any vertex)
- Messages sent to V in Superstep S are visible when Compute method is called in Superstep S+1
- No guaranteed order of messages, but guaranteed they will be delivered

16

Combiners & Aggregators

- Combiners
  - Suppose we know a vertex will always take the max value of all incoming values
  - Can reduce overhead by doing local maxs over all messages to same vertex
  - User can write a Combiner function that combines several messages intended for vertex V into a single message
  - No guarantees about what messages will or will not be combined
  - Fourfold reduction for single-source shortest path
- Aggregators
  - Global aggregations
  - Each vertex provides a value to the aggregator in each superstep (S)
  - Aggregate value made available to all vertices in step S+1
  - Sum applied to local out-degrees can count edges in the graph

17

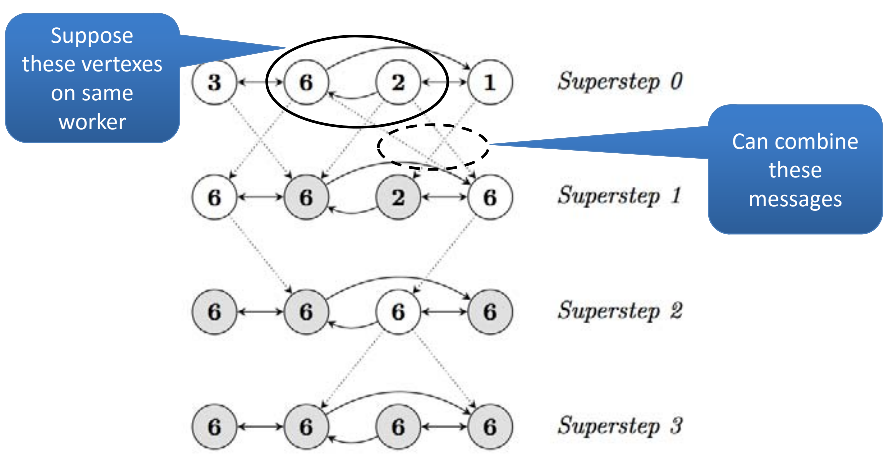


Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

Figure credit: Pregel: A System for Large-Scale Graph Processing

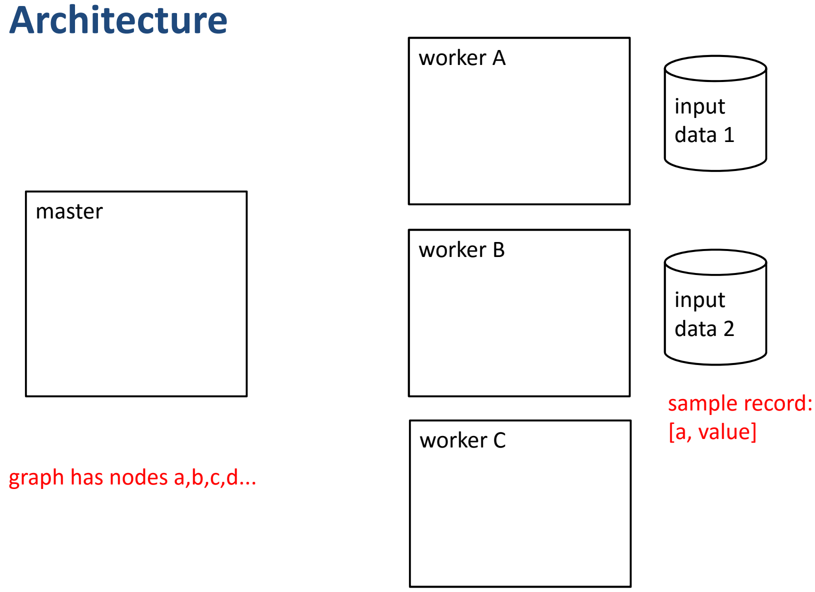
18

Architecture

- Designed for Google cluster architecture
- Graph is partitioned – sets of vertices and their outgoing edges
- Default partitioning is hash(vertexid) mod N, but can be customized (i.e. colocate vertices representing pages of same site)
- Execution stages:
  1. Initialization: Program copies begin executing – one is master – workers discover master's location and register with master
  2. Graph partitioning: Master determines # of partitions and assigns partitions to worker machines
  3. Load: master assigns a portion of input to each worker (independent of partitions); worker reads vertex and loads or forwards
  4. Master tells each worker to complete a Superstep; repeat until all vertexes vote halt
  5. [Master may instruct workers to checkpoint their portion of the graph]

19

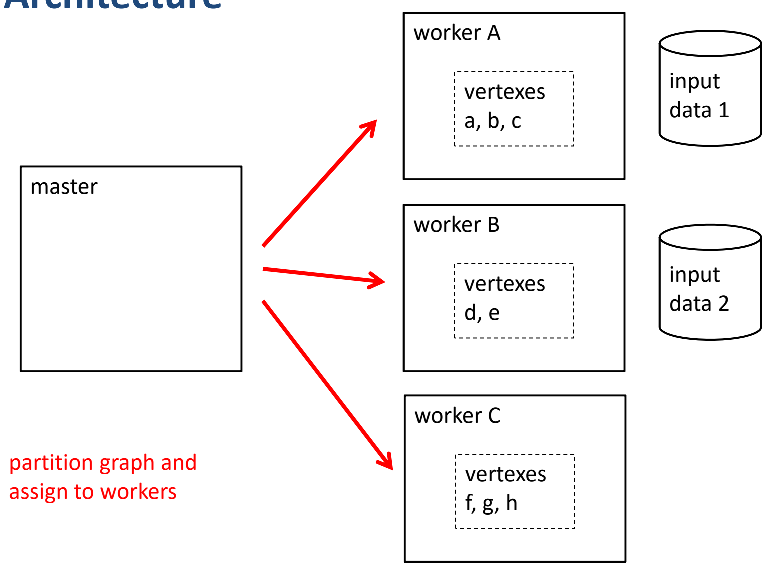
Architecture



Architecture slides credit Hector Garcia-Molina

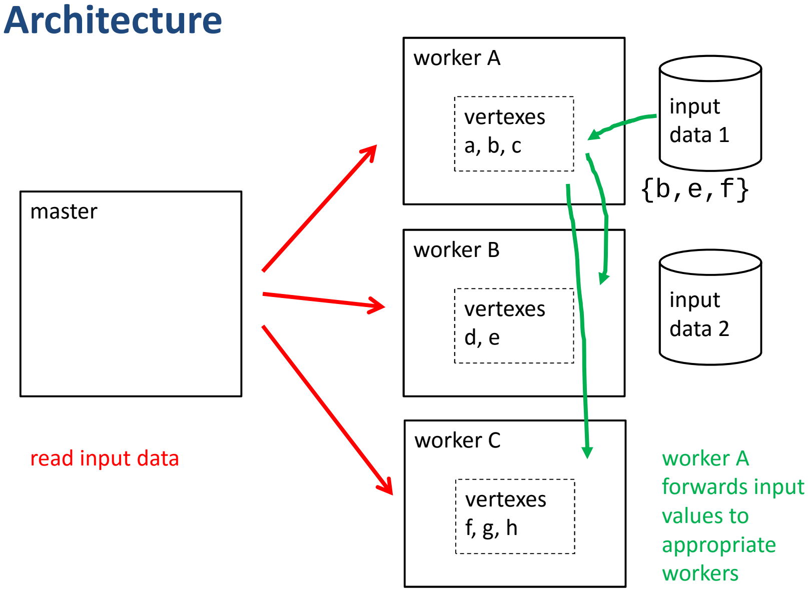
20

Architecture



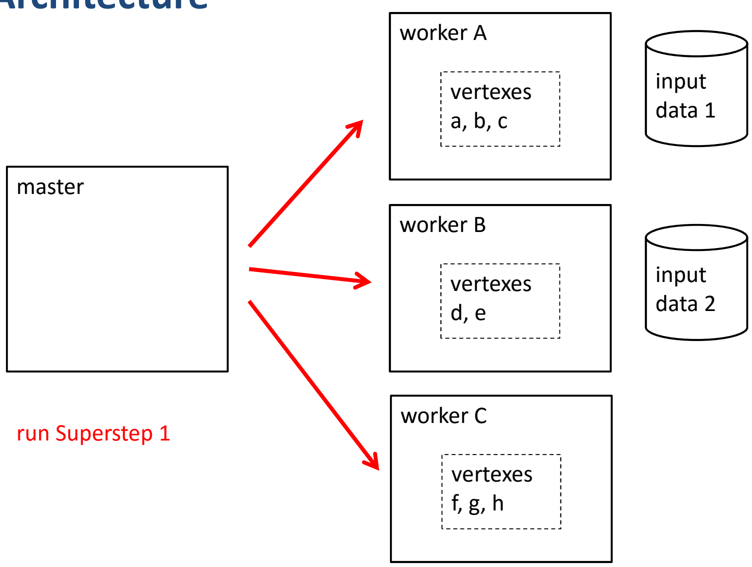
Architecture slides credit Hector Garcia-Molina 21

Architecture



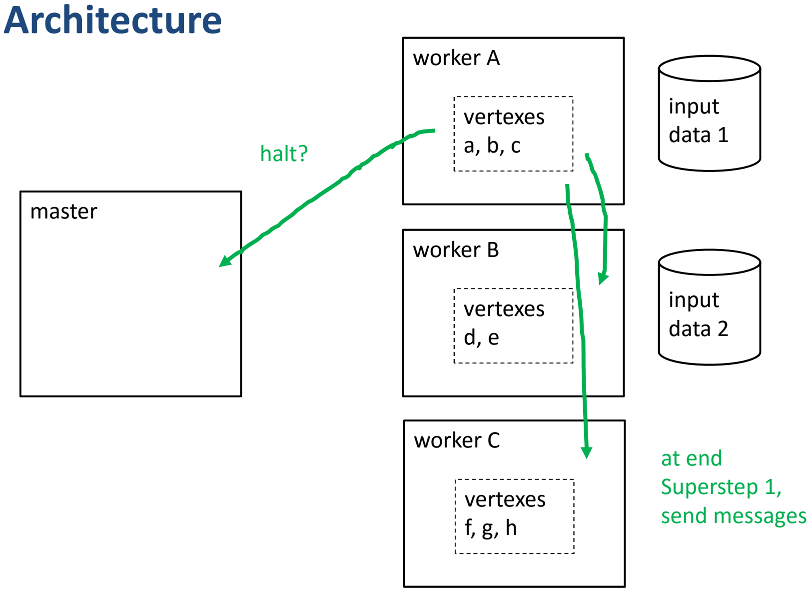
Architecture slides credit Hector Garcia-Molina 22

Architecture



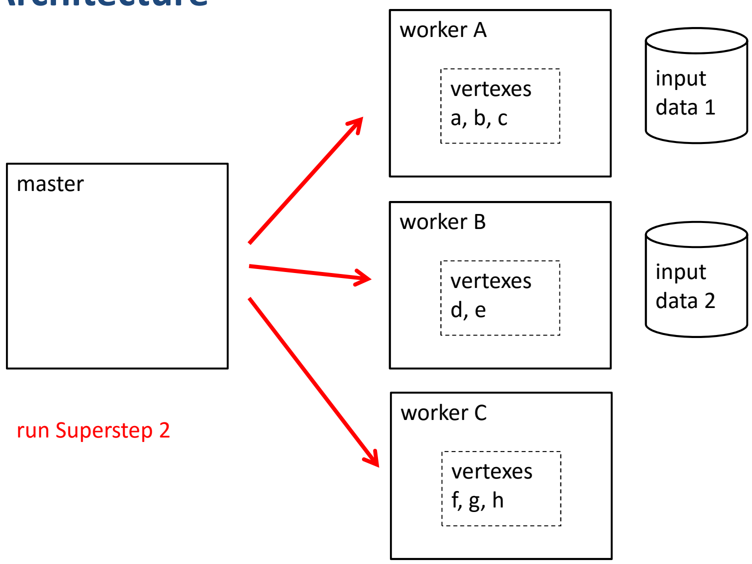
Architecture slides credit Hector Garcia-Molina 23

Architecture



Architecture slides credit Hector Garcia-Molina 24

Architecture

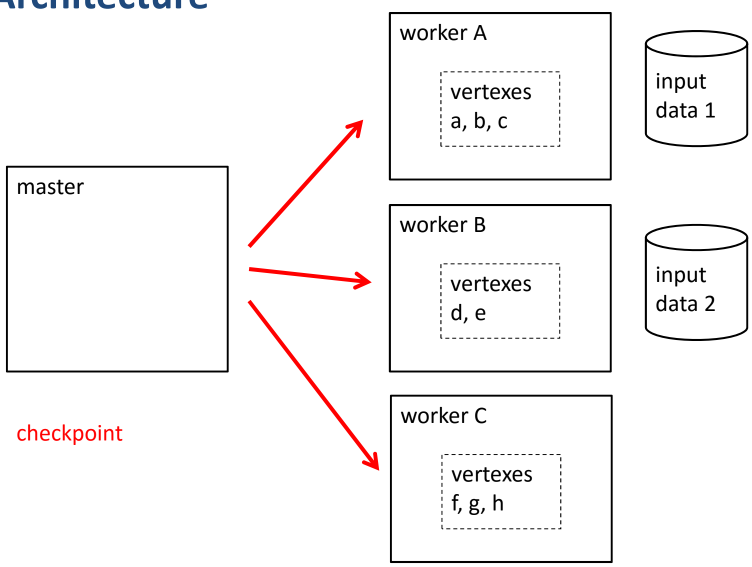


Architecture slides credit Hector Garcia-Molina

Fault Tolerance

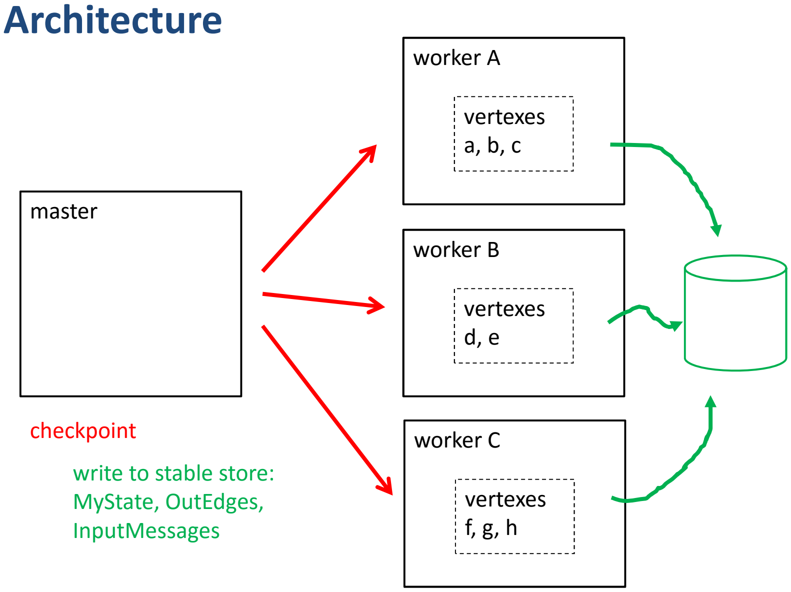
- Fault tolerance handled with checkpointing
- At start of a superstep (but not every superstep), master instructs workers to save state
  - Vertex values
  - Edge values
  - Incoming messages
  - Master saves aggregator values
- Worker fails (doesn't respond to ping)
  - Reassigns "lost" partitions
  - *Everyone* restarts from most recent checkpoint
- Confined recovery was under development
  - Recovery confined to only lost partitions

Architecture



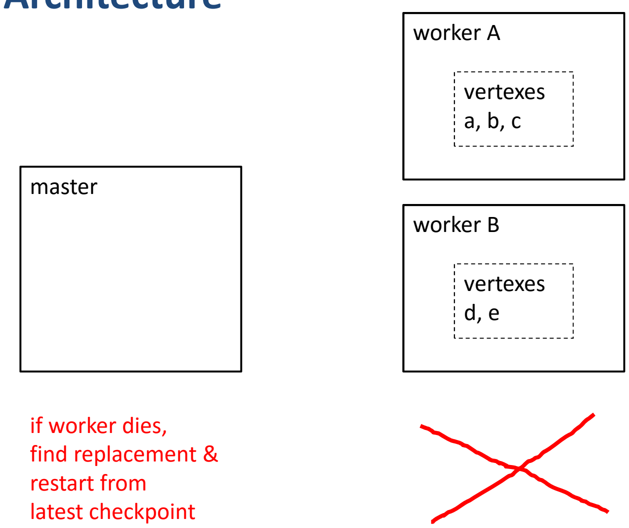
Architecture slides credit Hector Garcia-Molina

Architecture



Architecture slides credit Hector Garcia-Molina

Architecture



Architecture slides credit Hector Garcia-Molina

29

Implementation Details

- Worker maintains state of its portion of graph in memory
- Worker loops through all vertexes - each vertex Compute() function receives:
  - Vertex's current value
  - Iterator to incoming messages
  - Iterator to outoing edges
- Messages
  - Worker determines if messages are for a local or remote vertex
  - Remotes are buffered until threshold reached, then flushed
  - Combiners are applied when messages are:
    - added to outgoing message queue (reduces space and network transmission)
    - received at incoming message queue (reduces space only)

30

```
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();
    }
  }
};
```

Update my own page rank. 0.85 is “damping factor”

Distribute my value evenly among the pages I point to

Figure 4: PageRank implemented in Pregel.

Figure credit: Pregel: A System for Large-Scale Graph Processing

31

```
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();
  }
};
```

Initialization

Figure out my shortest path to source

Send out known shortest path to that vertex

Figure 5: Single-source shortest paths.

Figure credit: Pregel: A System for Large-Scale Graph Processing

32

```
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);
  }
};
```

Figure 6: Combiner that takes minimum of message values.

Figure credit: Pregel: A System for Large-Scale Graph Processing

33

Performance

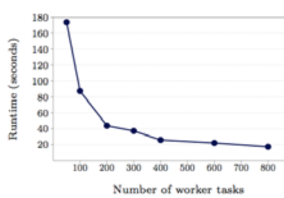


Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines

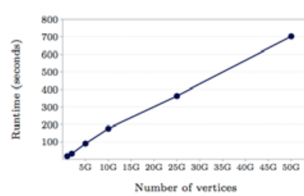


Figure 8: SSSP—binary trees: varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

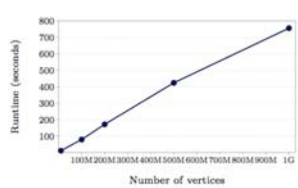


Figure 9: SSSP—log-normal random graphs, mean out-degree 127.1 (thus over 127 billion edges in the largest case): varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

34

Pregel Graph Processing System

- Vertex-based
- Distributed (message passing only)
- Parallel
- Fault-tolerant
- Master/Worker architecture

- To be continued …

Giraph: Billions → Trillions of Edges

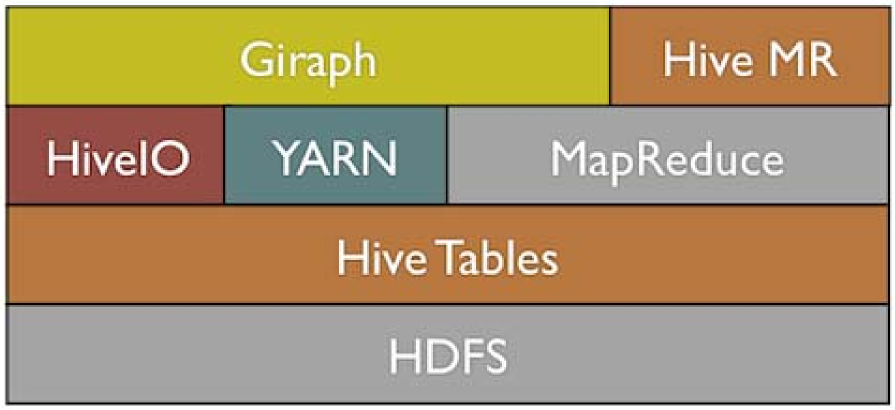
- Apache Giraph started as an open-source version of Pregel
- Facebook is a main contributor

Challenge: Scale graph-processing framework such as Pregel to 100s of billions of edges

Facebook graph (2014):
- 1.4B active users
- 600B edges

- Will organize material as issues + solutions.

Facebook Stack



From Ching, et al.: One Trillion Edges: Graph Processing at Facebook-Scale

37

Issue 1: Input Organization

Input needs to be organized as vertex plus outgoing edges

Might want to take edges from a different place than vertexes

- Vertexes: FB users
- Edges 1: User A likes posts of User B
- Edges 2: User A messages to User B

Solution: Allow edges to be supplied separately and distributed.

Often drawn from Hive tables

38

Issue 2: Better Pallelism

Were generally running one worker per machine.

Wasn't giving optimum parallelism

For example, slowest-worker problem

Solution: More parallelism options

- Coarse-grain: multiple workers per machine
- Fine-grain: multiple threads (hence cores) per worker

Second option works better: fewer TCP connections, bigger messages batches

39

Issue 3: Size of State

Vertex value, edge values, message payloads were all Java objects

Out-of-memory errors, lots of garbage collection

Solution: Serialize edge info for a vertex into a byte array

- In one example, reduced space by 6x
- Don't compress vertex data. Why?

40

Issue 4: Using Zookeeper for Aggregation

- Workers write partial aggregates to Zookeeper
- Master computer final aggregate and puts back in ZK
  - Limit of 1 MB per "znode", could have 10s of GB from each worker
  - Master doing all the work

Solution: Sharded aggregators – assign a different worker to each aggregate, communicate directly

Note that workers aren't busy between Supersteps

41

Solution 4: Sharded Aggregators

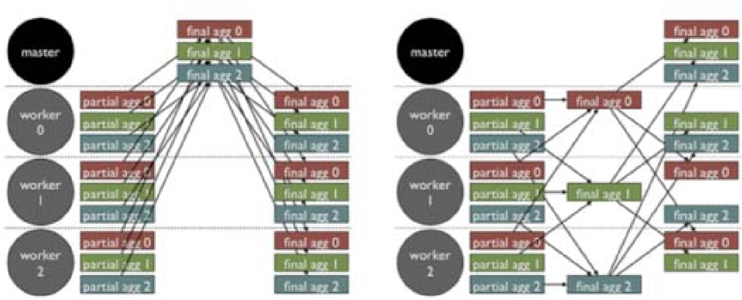


Figure 3: After sharding aggregators, aggregated communication is distributed across workers.

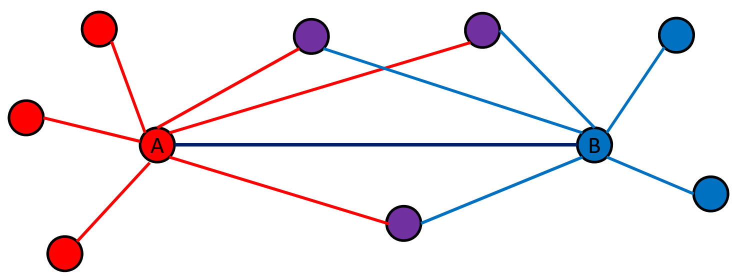
From Ching, et al.: One Trillion Edges: Graph Processing at Facebook-Scale

42

Issue 5: Different Computations

Might need to run different computations at different Supersteps.

Example: Friend of a Friend (FOAF)
- Which friends do you have the most friends in common with?
- Note: This is a simple form of triangles



Mutual Friends

- Superstep 1: Each vertex sends its friend list to its neighbors (friends)
- Superstep 2: Each vertex compares incoming friend lists with its own set of friends, finds largest intersection

Solution: Separate Computation from Vertex
- Define multiple possible computations for the vertexes
- Master says which one to use at the beginning of the Superstep.

Issue 6: Too Much Message State

Not enough room for all the incoming messages

- Consider mutual friends
  - Maximum of 5000 friends on FB
  - How much message state can one vertex receive, potentially?

45

Solution 6: "Sub-Supersteps"

Divide message-sending and processing into rounds

- For example:
  - Round 1: Send messages to even # vertexes
  - Round 2: Send messages to odd # vertexes

Choose the number of rounds such that one round of messages fits in main memory of a worker.

46

Remaining Issues

- Graph partitioning
- Asynchronous messaging option

47

Other Graph-Processing Frameworks

- GraphX: Combining graphs and tables, uses Spark
- GraphLab: Asynchronous messaging

48

References

- Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (SIGMOD '10).
- Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (August 2015), 1804-1815.