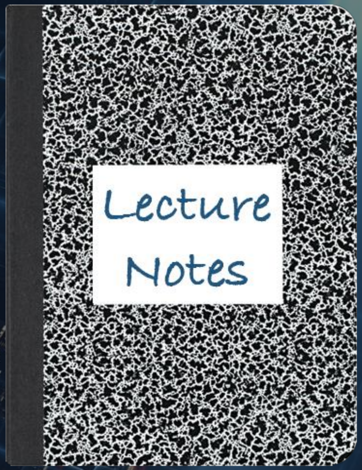
CS 417 – DISTRIBUTED SYSTEMS



Week 10: Large-Scale Data Processing

Part 2: Bulk Synchronous Parallel & Pregel

Paul Krzyzanowski

MapReduce isn't always the answer

MapReduce works well for certain problems

- The framework provides
  - Automatic parallelization
  - Automatic job distribution

For others:

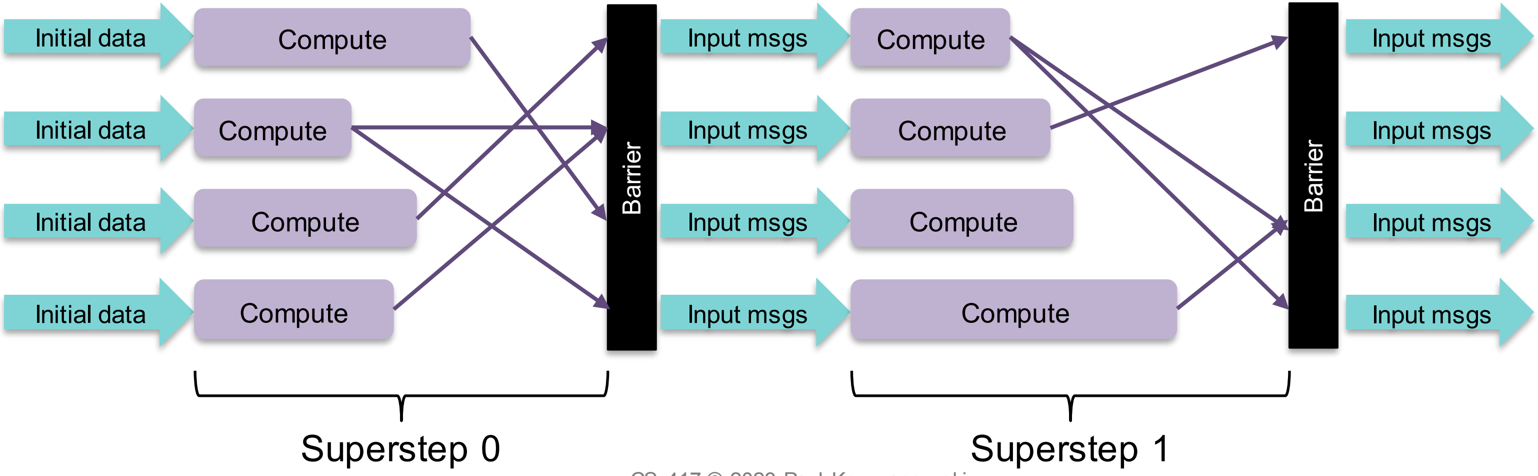
- May require many iterations of MapReduce
- Data locality usually not preserved between Map and Reduce
  - Lots of communication between *map* and *reduce* workers

Bulk Synchronous Parallel (BSP)

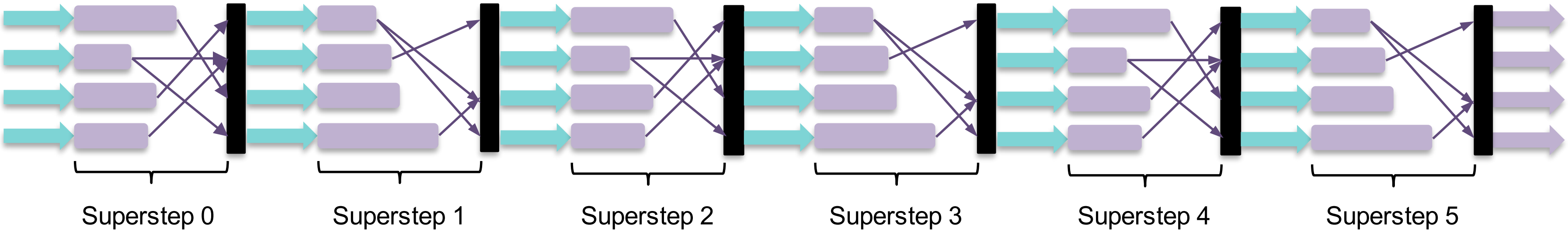
Created as a computing model for parallel computation

Execution is a series of supersteps

1. Concurrent computation
2. Communication
3. Barrier synchronization



Bulk Synchronous Parallel (BSP)

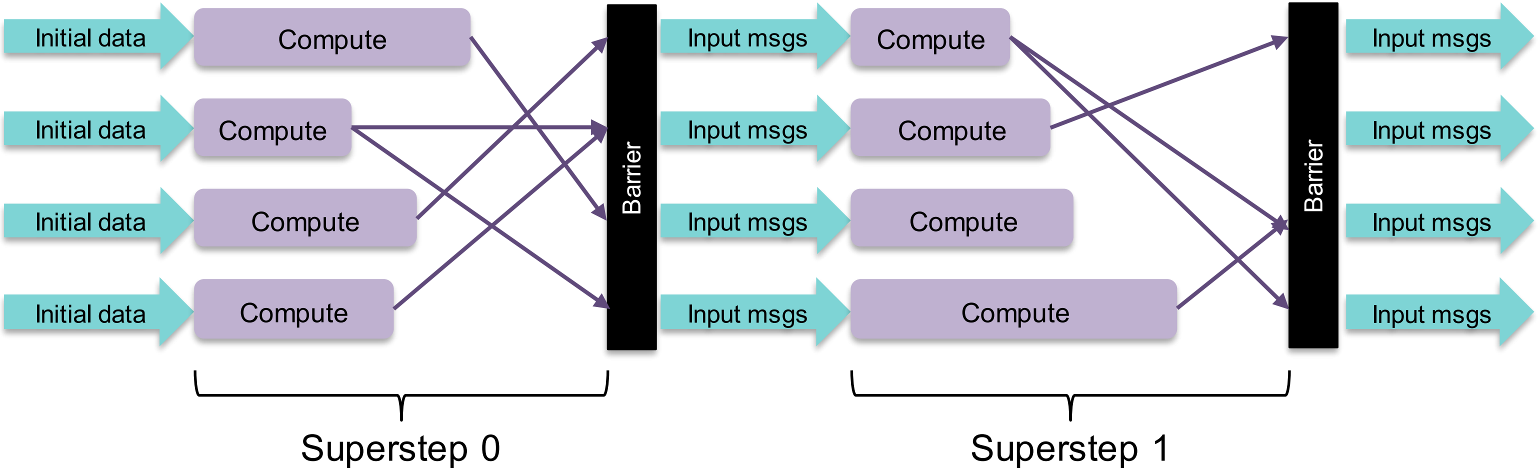


Bulk Synchronous Parallel (BSP)

Series of supersteps

1. Concurrent computation
2. Communication
3. Barrier synchronization

- Processes (workers) are randomly assigned to processors
- Each process uses only local data
- Each computation is asynchronous of other concurrent computation
- Computation time may vary

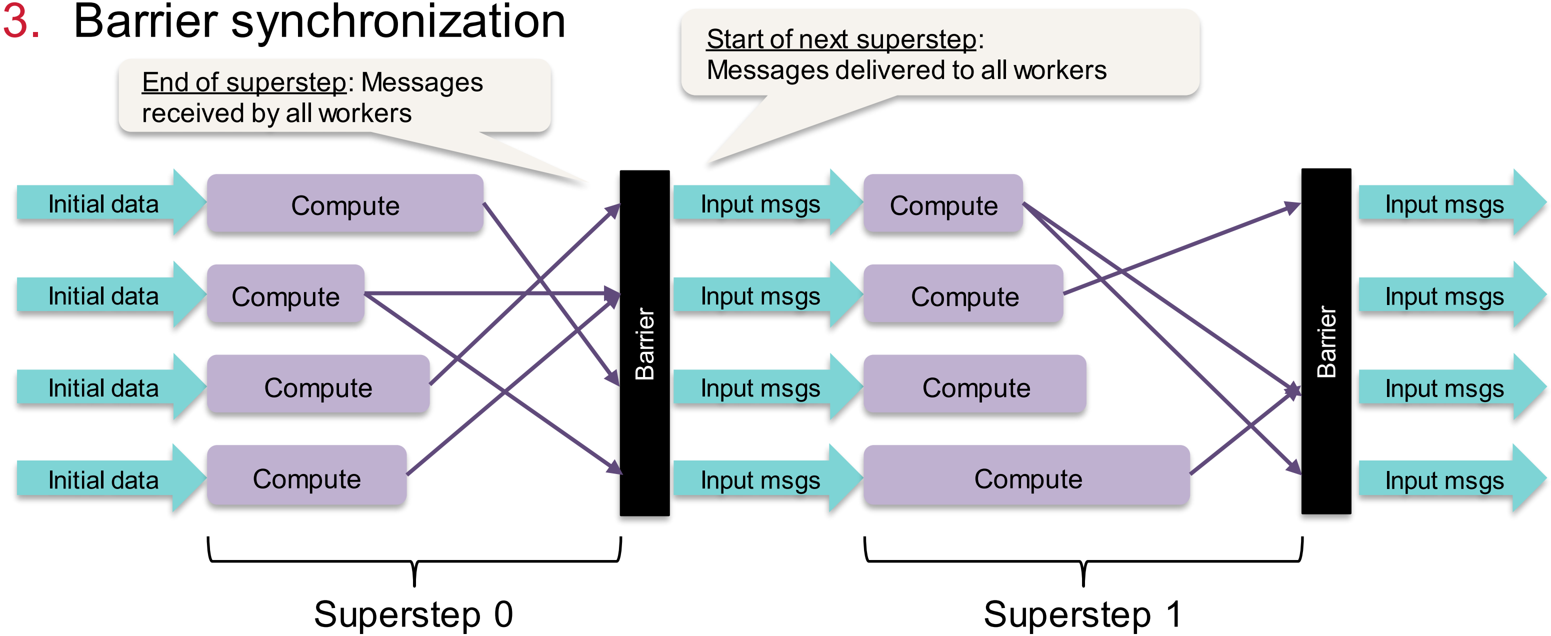


Bulk Synchronous Parallel (BSP)

Series of supersteps

1. Concurrent computation
2. Communication
3. Barrier synchronization

- Incoming messages are received at the start of a superstep
- Messaging are sent by a process during a superstep
- Each process may send a message to 0 or more processes
- These messages become inputs for the next superstep

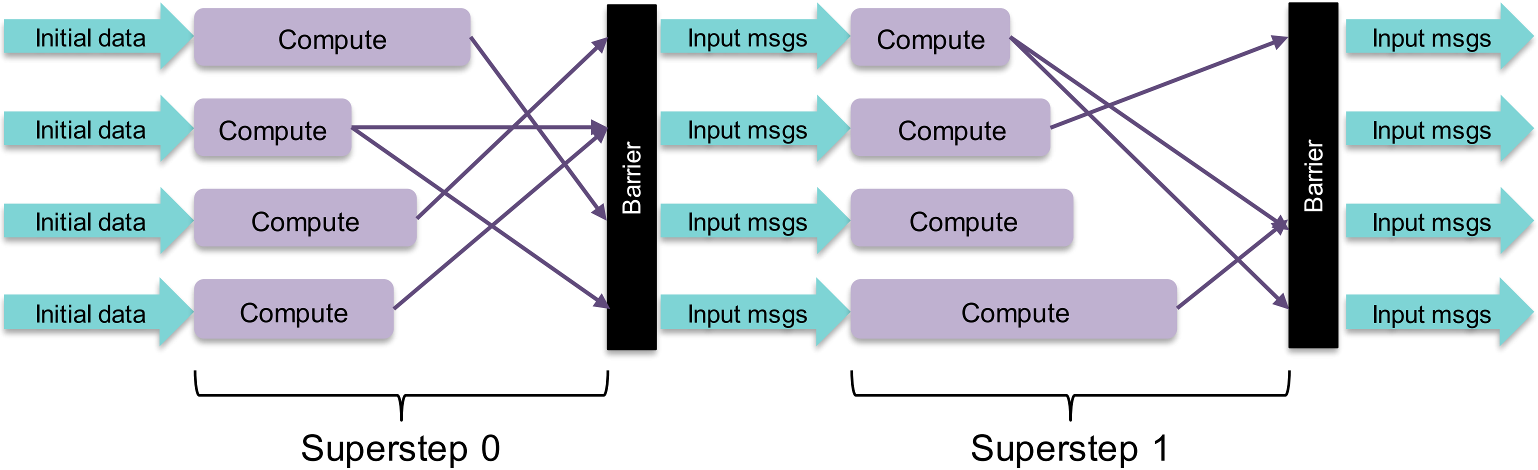


Bulk Synchronous Parallel (BSP)

Series of supersteps

1. Concurrent computation
2. Communication
3. Barrier synchronization

- The next superstep does not begin until **all** messages have been received
- Barriers ensure no deadlock: no circular dependency can be created
- Provide an opportunity to **checkpoint** results for fault tolerance
  - If there's a failure, restart computation from the last superstep



BSP Implementation: Apache Hama

- **Hama**: BSP framework on top of HDFS
  - Provides automatic parallelization & distribution
  - Uses Hadoop RPC
    - Data is serialized with Google Protocol Buffers
  - Zookeeper for coordination (Apache version of Google's Chubby)
    - Handles notifications for Barrier Sync

- Good for applications with data locality
  - Matrices and graphs
  - Algorithms that require a lot of iterations

hama.apache.org

Hama programming (high-level)

- Pre-processing
  - Define the number of peers for the job
  - Split initial inputs for each of the peers to run their supersteps
  - Framework assigns a unique ID to each worker (peer)

- Superstep: the worker function is a superstep
  - ***getCurrentMessage()*** – input messages from previous superstep
  - Compute – your code
  - ***send(peer, msg)*** – send messages to a peer
  - ***sync()*** – synchronize with other peers (barrier)

- File I/O
  - Key/value model used by Hadoop MapReduce & HBase *Google Bigtable*
  - ***readNext(key, value)***
  - ***write(key, value)***

For more information

- Architecture, examples, API
- Take a look at:
  - Apache Hama project page
    - http://hama.apache.org
  - Hama BSP tutorial
    - https://hama.apache.org/hama_bsp_tutorial.html
  - Apache Hama Programming document
    - http://bit.ly/1aiFbXS
      http://people.apache.org/~tjungblut/downloads/hamadocs/ApacheHamaBSPProgrammingmodel_06.pdf

Graph computing

Graphs are common in computing

- Social links
  - Friends
  - Academic citations
  - Music
  - Movies
- Web pages
- Network connectivity
- Roads
- Disease outbreaks

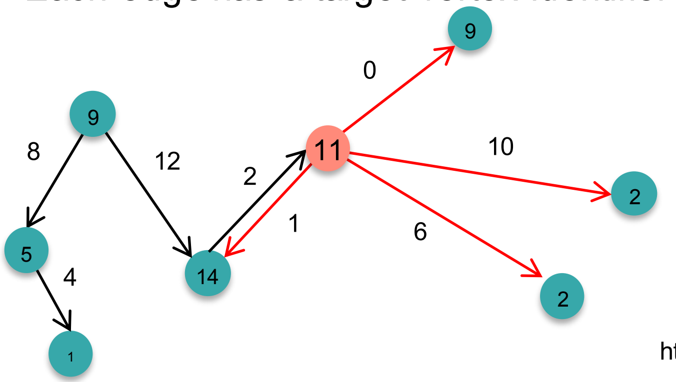
Processing graphs on a large scale is hard

- Computation with graphs
  - Poor locality of memory access
  - Little work per vertex
- Distribution across machines
  - Communication complexity
  - Failure concerns
- Solutions
  - Application-specific, custom solutions
  - MapReduce or databases
    - The <key,value> view of the world isn't the most natural for graphs
    - But require many iterations (and a lot of data movement)
  - Single-computer libraries: limits scale
  - Parallel libraries: do not address fault tolerance
  - BSP: ***close*** but too general

Pregel: a vertex-centric BSP

Input: directed graph

- A vertex is an object
  - Each vertex uniquely identified with a name
  - Each vertex has a modifiable value
- Directed edges: links to other objects
  - Associated with source vertex
  - Each edge has a modifiable value
  - Each edge has a target vertex identifier



Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

ABSTRACT

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution-related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive and easy to program.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Design, Algorithms

Keywords

Distributed computing, graph algorithms

1. INTRODUCTION

The Internet made the Web graph a popular object of analysis and research. Web 2.0 fueled interest in social networks. Other large graphs—for example induced by transportation routes, similarity of newspaper articles, paths of disease outbreaks, or citation relationships among published scientific work—have been processed for decades. Frequently applied algorithms include shortest paths computations, different flavors of clustering, and variations on the page rank theme. There are many other graph computing problems of practical value, e.g., minimum cut and connected components.

Efficient processing of large graphs is challenging. Graph algorithms often exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution [31, 39]. Distribution over many machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose system for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment.

Implementing an algorithm to process a large graph typically means choosing among the following options:

1. Crafting a custom distributed infrastructure, typically requiring a substantial implementation effort that must be repeated for each new algorithm or graph representation.
2. Relying on an existing distributed computing platform, often ill-suited for graph processing. MapReduce [14], for example, is a very good fit for a wide array of large-scale computing problems. It is sometimes used to mine large graphs [11, 30], but this can lead to suboptimal performance and usability issues. The basic models for processing data have been extended to facilitate aggregation [41] and SQL-like queries [40, 47], but these extensions are usually not ideal for graph algorithms that often better fit a message passing model.
3. Using a single-computer graph algorithm library, such as BGL [43], LEDA [35], NetworkX [25], JDSL [20], Stanford GraphBase [29], or FGL [16], limiting the scale of problems that can be addressed.
4. Using an existing parallel graph system. The Parallel BGL [22] and CGMgraph [8] libraries address parallel graph algorithms, but do not address fault tolerance or other issues that are important for very large scale distributed systems.

None of these alternatives fit our purposes. To address distributed processing of large scale graphs, we built a scalable

135

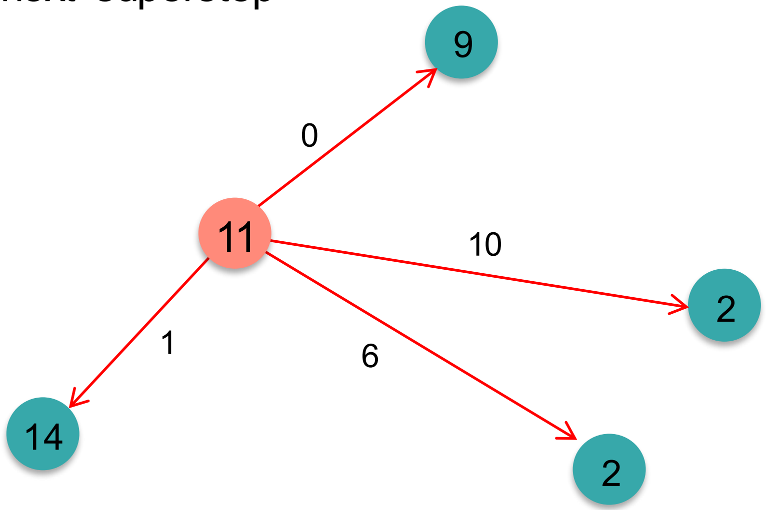
http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html

Pregel: computation

Computation: series of supersteps

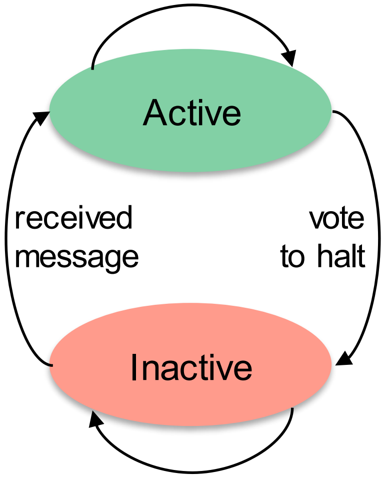
- Same user-defined function **runs on each vertex**
  - Receives messages sent from the previous superstep
  - May modify the state of the vertex or of its outgoing edges
  - Sends messages that will be received in the next superstep
    - Typically to outgoing edges
    - But can be sent to any known vertex
  - May modify the graph topology

Each superstep ends with a **barrier** (synchronization point)



Pregel: termination

- Initially, every vertex is in an *active* state
  - Active vertices compute during a superstep
- Each vertex may choose to deactivate itself by **voting to halt**
  - The vertex has no more work to do
  - Will not be executed by Pregel
  - UNLESS the vertex receives a message
    - Then it is reactivated
    - Will stay active until it votes to halt again
- Algorithm terminates when all vertices are inactive and there are no messages in transit



Vertex State Machine

Pregel: output

Output is the set of values output by the vertices

- Often a directed graph
  - May be non-isomorphic to original since edges & vertices can be added or deleted
- Or may be summary data

Examples of graph computations

- **Shortest path to a node**
  - Each iteration, a node sends the shortest distance received to all neighbors
- **Cluster identification**
  - Each iteration: get info about clusters from neighbors
  - Add myself
  - Pass useful clusters to neighbors (e.g., within a certain depth or size)
    - May combine related vertices
    - Output is a smaller set of disconnected vertices representing clusters of interest
- **Graph mining**
  - Traverse a graph and accumulate global statistics
- **PageRank**
  - Each iteration: update web page ranks based on messages from incoming links

Simple example: find the maximum value

Each vertex contains a value – we want to find the largest one

- In the first superstep:
  - A vertex sends its value to its neighbors
- In each successive superstep:
  - If a vertex learned of a larger value from its incoming messages, it sends it to its neighbors
  - Otherwise, it votes to halt
- Eventually, all vertices get the largest value
- When no vertices change in a superstep, the algorithm terminates

Simple example: find the maximum value

1. vertex value type;
2. edge value type (none!)
3. message value type

Semi-pseudocode:

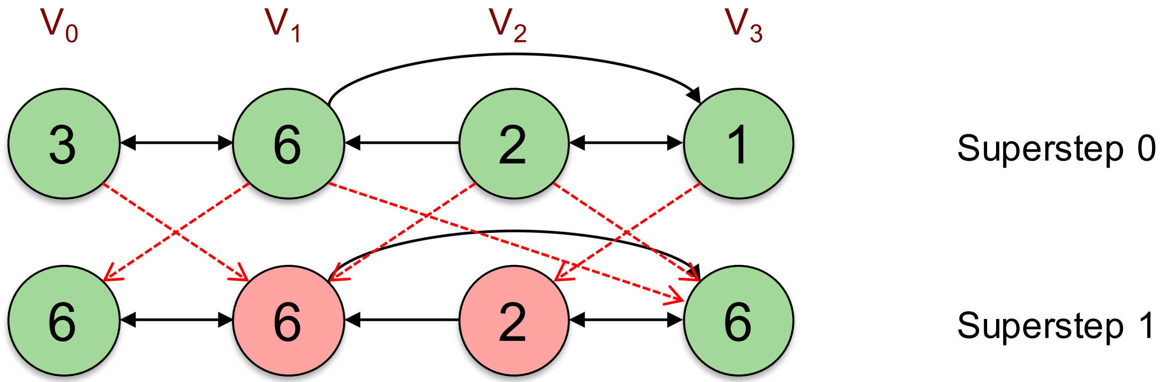
```
class MaxValueVertex
     : public Vertex<int, void, int> {
   void Compute(MessageIterator *msgs) {
      int maxv = GetValue();
      for (; !msgs->Done(); msgs->Next())
          maxv = max(msgs.Value(), maxv);

      if (maxv > GetValue()) || (step == 0)) {
          *MutableValue() = maxv;
          OutEdgeIterator out = GetOutEdgeIterator();
          for (; !out.Done(); out.Next())
               sendMessageTo(out.Target(), maxv)
      } else
          VoteToHalt();
      }
   }
};
```

find maximum value

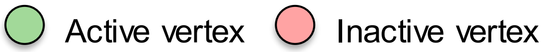
send maximum value to all edges

Simple example: find the maximum value

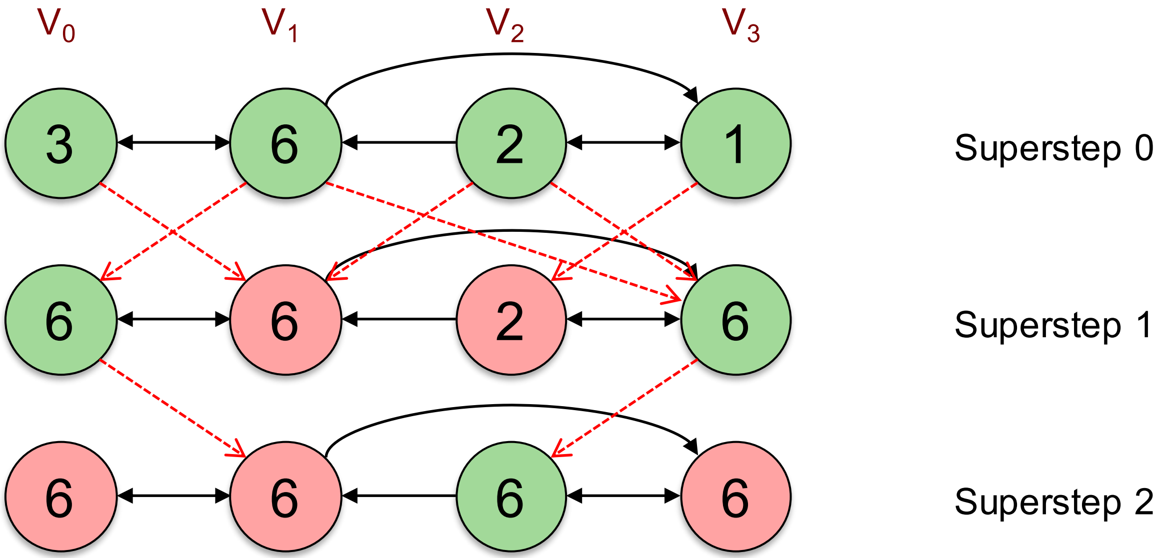


Superstep 0: Each vertex propagates its own value to connected vertices

Superstep 1: V_0 updates its value: 6 > 3
V_3 updates its value: 6 > 1
V_1 and V_2 do not update so vote to halt



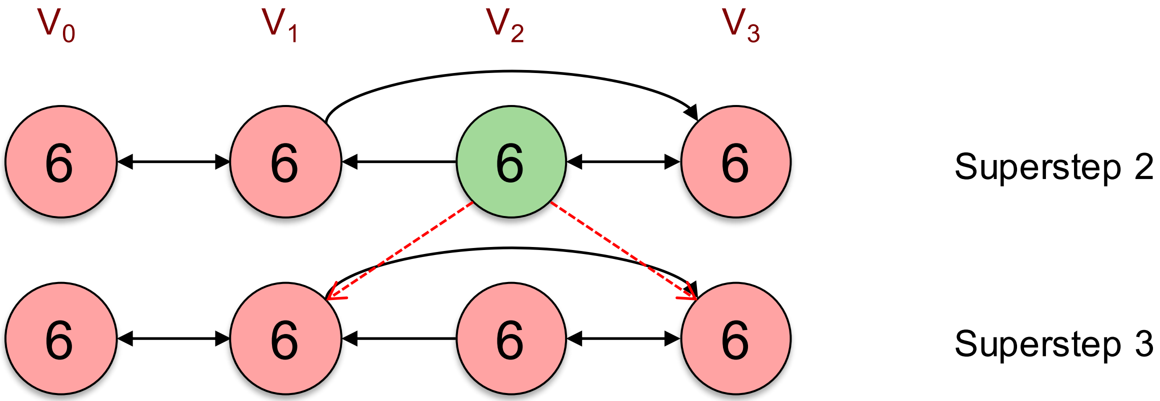
Simple example: find the maximum value



Superstep 2: V_1 receives a message – becomes active
V_3 updates its value: 6 > 2
V_1, V_2, and V_3 do not update so vote to halt

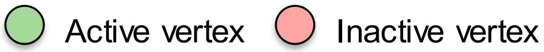
Active vertex Inactive vertex

Simple example: find the maximum value

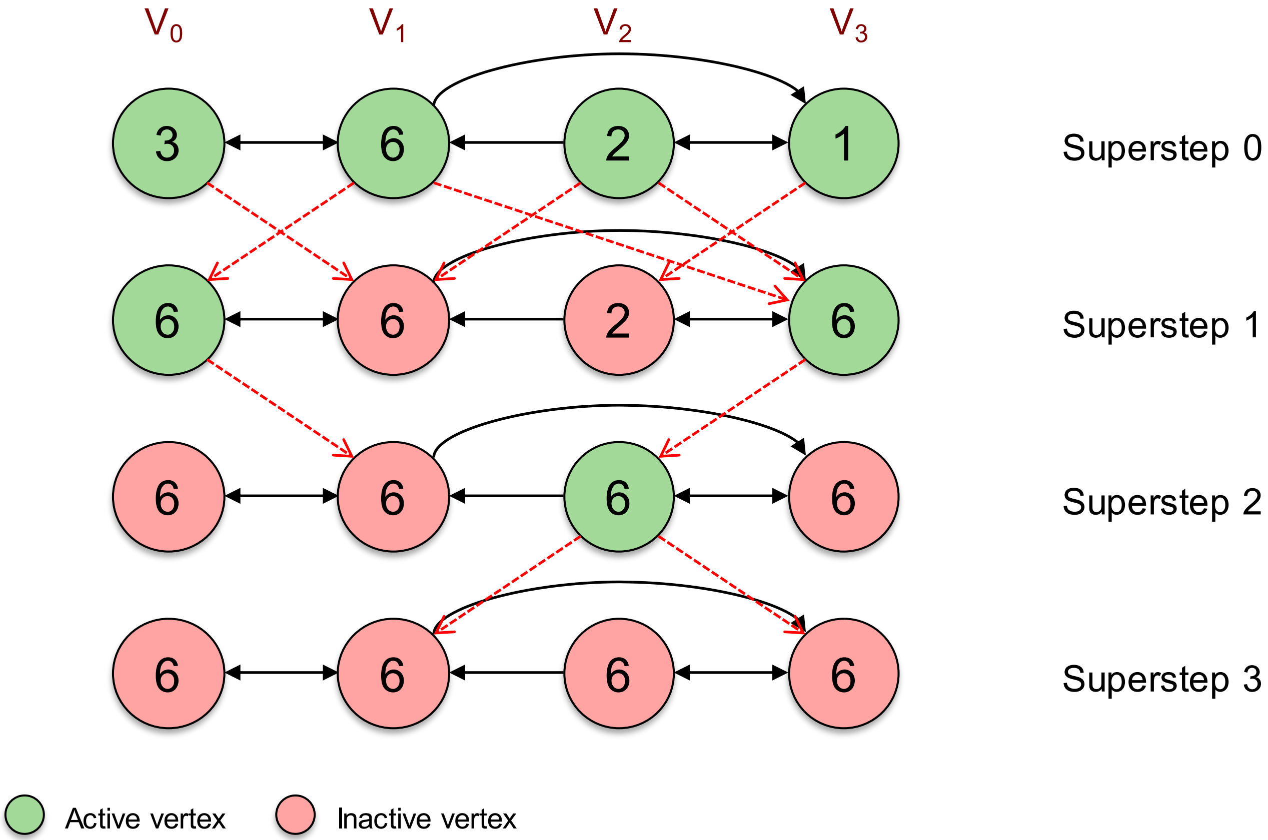


Superstep 3: V_1 receives a message – becomes active
V_3 receives a message – becomes active
No vertices update their value – **all vote to halt**

Done!



Summary: find the maximum value



Locality

- Vertices and edges remain on the machine that does the computation

- To run the same algorithm in MapReduce
  - Requires chaining multiple MapReduce operations
  - Entire graph state must be passed from *Map* to *Reduce*

  … and again as input to the next *Map*

Pregel API: Basic operations

A user subclasses a `Vertex` class

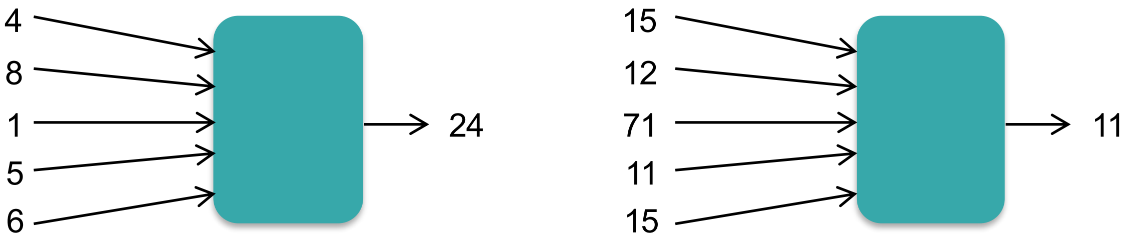
Methods:

- `Compute(MessageIterator*)`: Executed per active vertex in each superstep
  - MessageIterator identifies incoming messages from the previous superstep
- `GetValue()`: Get the current value of the vertex
- `MutableValue()`: Set the value of the vertex
- `GetOutEdgeIterator()`: Get a list of outgoing edges
  - `.Target()`: identify target vertex on an edge
  - `.GetValue()`: get the value of the edge
  - `.MutableValue()`: set the value of the edge
- `SendMessageTo()`: send a message to a vertex
  - Any number of messages can be sent
  - Ordering among messages is not guaranteed
  - A message can be sent to *any* vertex (but our vertex needs to have its ID)

Pregel API: Special operations

Combiners

- Each message has an overhead – let's reduce # of messages
  - Many vertices are processed per worker (multi-threaded)
  - Pregel can combine messages targeted to one vertex into one message
- Combiners are application specific
  - Programmer subclasses a Combiner class and overrides Combine() method
- No guarantee on which messages will be combined



Combiner
Sums input messages

Combiner
Minimum value

Pregel API: Special operations

Aggregators

- **Handle global data**
- A vertex can provide a value to an aggregator during a superstep
  - Aggregator combines received values to one value
  - Value is available to all vertices in the next superstep
- User subclasses an Aggregator class
- Examples
  - Keep track of total edges in a graph
  - Generate histograms of graph statistics
  - Global flags: execute until some global condition is satisfied
  - Election: find the minimum or maximum vertex

Pregel API: Special operations

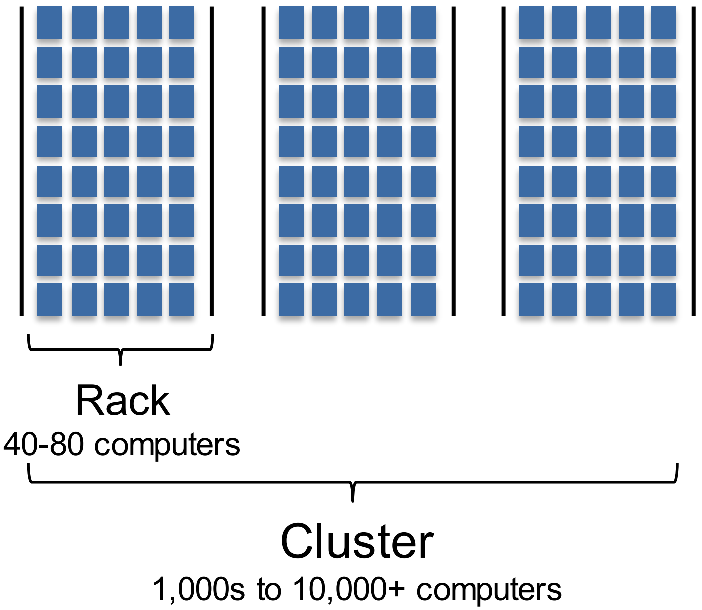
Topology modification

- Examples
  - If we're computing a spanning tree: remove unneeded edges
  - If we're clustering: combine vertices into one vertex
- Add/remove edges/vertices
- Modifications visible in the next superstep

Pregel Design

Execution environment

- Many copies of the program are started on a cluster of machines
- One copy becomes the **master**
  - Will not be assigned a portion of the graph
  - Responsible for coordination
  - The rest will be **workers**
- **Chubby** is used as a name server for the cluster
  - Master registers itself with the name service
  - Workers contact the name service to find the master



Partition assignment

- Master
  - Determines # partitions in graph
  - One or more partitions assigned to each worker
    - Partition = set of vertices
    - Default for N partitions: hash(vertex ID) mod $N \Rightarrow$ worker

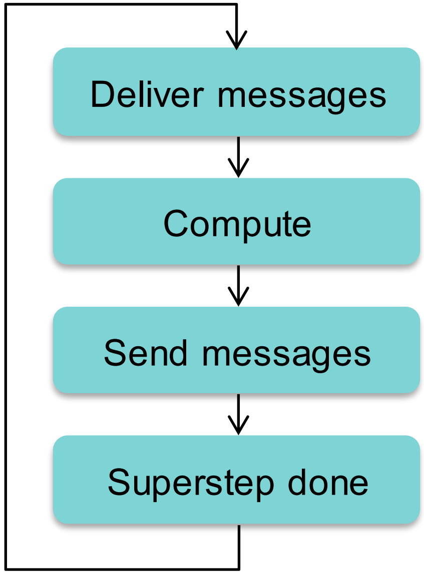
      May deviate: e.g., place vertices representing the same web site in one partition
    - Multiple partitions are assigned per worker: this improves load balancing
- Worker
  - Responsible for its section(s) of the graph
  - Each worker knows the vertex assignments of other workers

Input assignment

- Master assigns parts of the input to each worker
  - Data usually sits in GFS or Bigtable
- Input = set of records
  - Record = vertex data and edges
  - Assignment based on file boundaries
- Worker reads input
  - If it belongs to vertices it manages, local data structures are updated
  - Else worker sends messages to remote workers
- After data is loaded, all vertices are active

Computation

- Master tells each worker to perform a superstep
- Worker:
  - Iterates through vertices (one thread per partition)
  - Calls *Compute()* method for each active vertex
  - Delivers messages from the previous superstep
  - Outgoing messages
    - Sent asynchronously
    - Delivered before the end of the superstep
- When done
  - worker tells master how many vertices will be active in the next superstep
- Computation done when no more active vertices in the cluster
  - Master may instruct workers to save their portion of the graph



Handling failure

- **Checkpointing**
  - Controlled by master … every *N* supersteps
  - Master asks a worker to checkpoint at the start of a superstep
    - Save state of partitions to persistent storage
      - Vertex values, Edge values, Incoming messages
  - Master is responsible for saving aggregator values
- Failure detection: master sends *ping* messages to workers
  - If worker does not receive a ping within a time period ⇒ *Worker terminates*
  - If the master does not hear from a worker ⇒ *Master marks worker as failed*
- Restart: when failure is detected
  - Master reassigns partitions to the current set of workers
  - **All** workers reload partition state from most recent checkpoint

Pregel outside of Google

Apache Giraph

– Initially created at Yahoo

– Used at LinkedIn & Facebook to analyze the social graphs of users
  - Facebook is the main contributor to Giraph
  - Facebook analyzed 1 trillion edges via 200 machines in 4 minutes

– Runs under Hadoop MapReduce framework
  - Runs as a *Map*-only job
  - Adds fault-tolerance to the master by using ZooKeeper for coordination
  - Uses Java instead of C++

= Chubby

https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

Conclusion

Vertex-centric approach to BSP

- Computation = set of supersteps
  - Compute() called on each vertex per superstep
  - Communication between supersteps: barrier synchronization
- Hides distribution from the programmer
  - Framework creates lots of workers
  - Distributes partitions among workers
  - Reads graph input
  - Handles message sending, receipt, and synchronization
  - A programmer just has to think from the viewpoint of a vertex
- Checkpoint-based fault tolerance

The End