Distributed Systems

21. Graph Computing Frameworks

Paul Krzyzanowski

Rutgers University

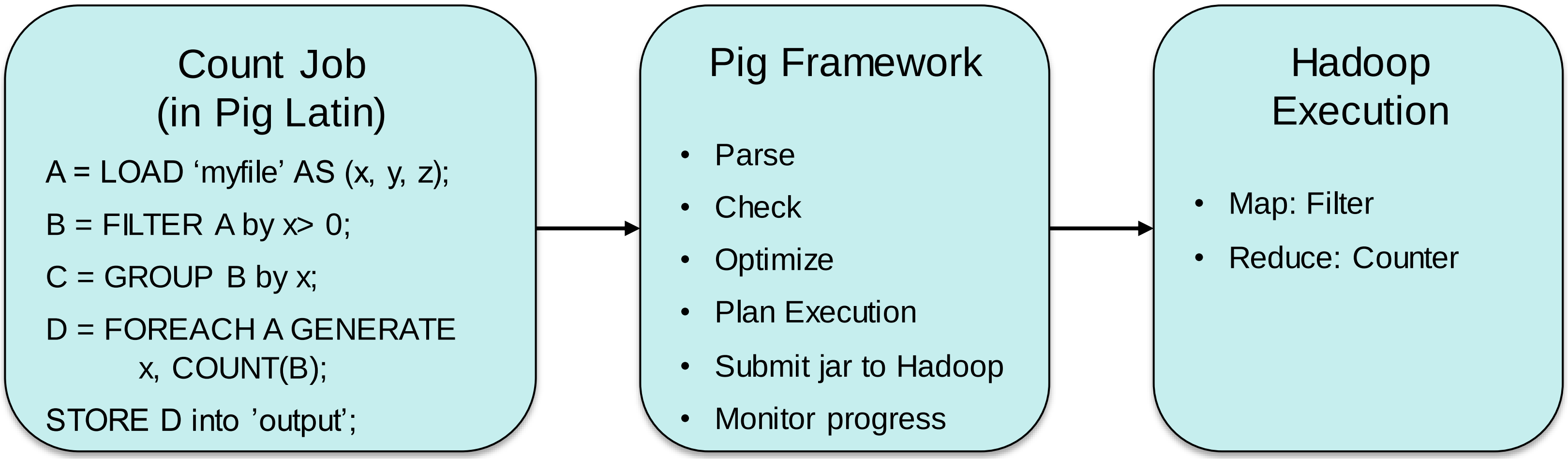
Fall 2016

Can we make MapReduce easier?

Apache Pig

- Why?
  - Make it easy to use MapReduce via scripting instead of Java
  - Make it easy to use multiple MapReduce stages
  - Built-in common operations for join, group, filter, etc.
- How to use?
  - Use Grunt – the pig shell
  - Submit a script directly to pig
  - Use the PigServer Java class
  - PigPen – Eclipse plugin
- Pig compiles to several Hadoop MapReduce jobs

Apache Pig


Count Job
(in Pig Latin)
A = LOAD ‘myfile’ AS (x, y, z);
B = FILTER A by x> 0;
C = GROUP B by x;
D = FOREACH A GENERATE
x, COUNT(B);
STORE D into ’output’;
Pig Framework
• Parse
• Check
• Optimize
• Plan Execution
• Submit jar to Hadoop
• Monitor progress
Hadoop
Execution
• Map: Filter
• Reduce: Counter

Pig: Loading Data

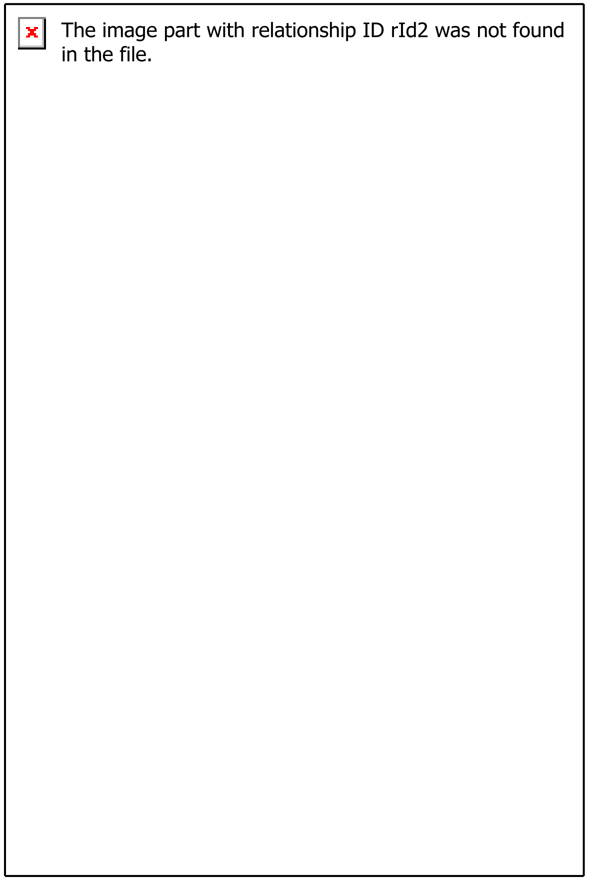
Load/store relations in the following formats:

- PigStorage: field-delimited text
- BinStorage: binary files
- BinaryStorage: single-field tuples with a value of *bytearray*
- TextLoader: plain-text
- PigDump: stores using toString() on tuples, one per line

Example

```
log = LOAD 'test.log' AS (user, timestamp, query);
grpd = GROUP log by user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
fltrd = FILTER cntd BY cnt > 50;
srtd = ORDER fltrd BY cnt;
STORE srtd INTO 'output';
```

- Each statement defines a new dataset
  - Datasets can be given aliases to be used later
- FOREACH iterates over the members of a ”bag”
  - Input is grpd: list of log entries grouped by user
  - Output is group, COUNT(log): list of {user, count}
- FILTER applies conditional filtering
- ORDER applies sorting



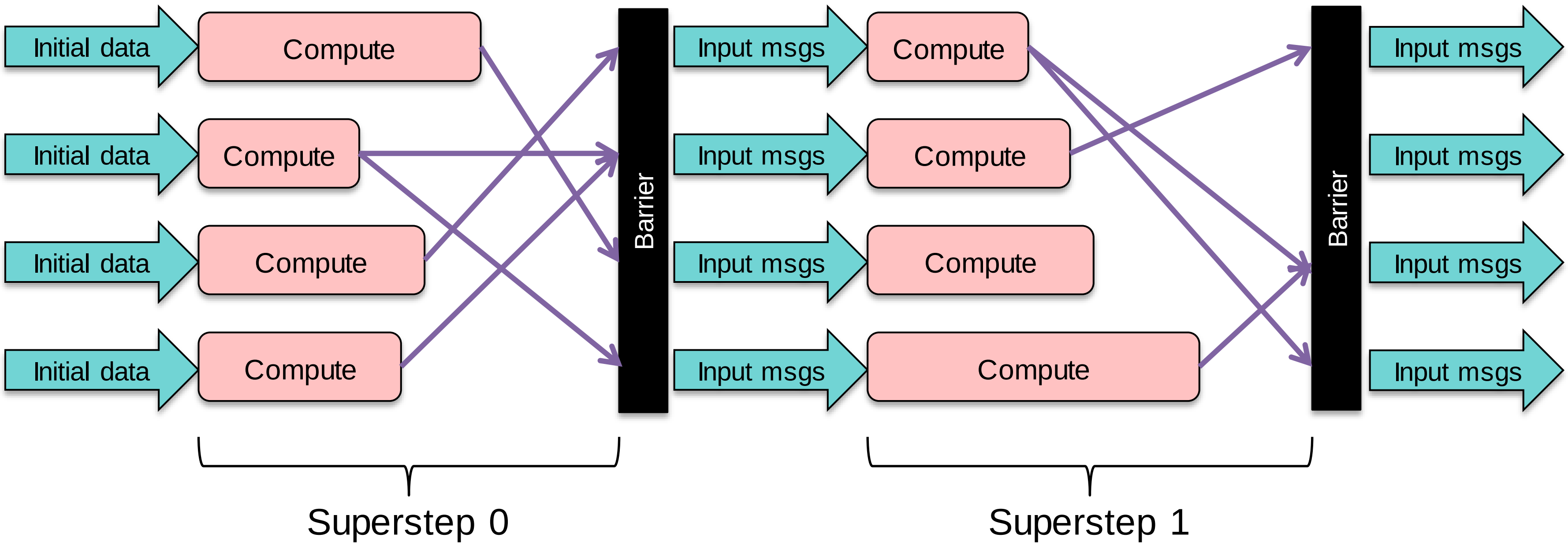
See pig.apache.org
for full documentation

MapReduce isn't always the answer

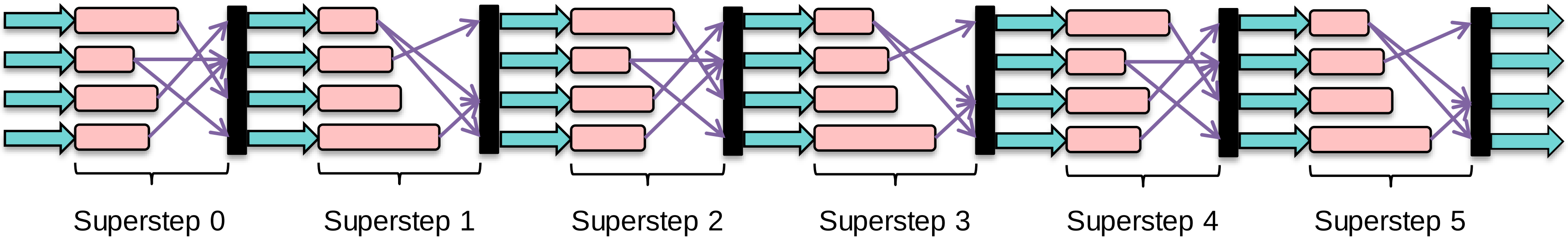
- MapReduce works well for certain problems
  - Provides automatic parallelization
  - Automatic job distribution
- For others
  - May require many iterations
  - Data locality usually not preserved between Map and Reduce
    - Lots of communication between *map* and *reduce* workers

Bulk Synchronous Parallel (BSP)

- Computing model for parallel computation
- Series of supersteps
  1. Concurrent computation
  2. Communication
  3. Barrier synchronization



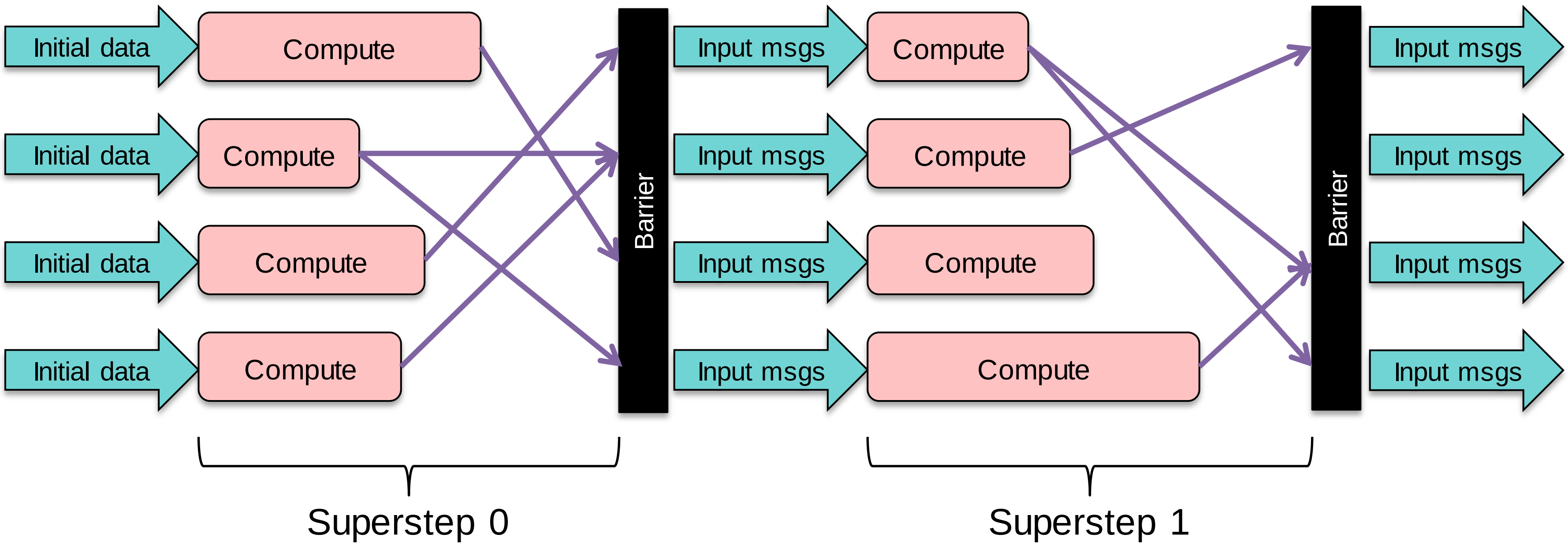
Bulk Synchronous Parallel (BSP)



Bulk Synchronous Parallel (BSP)

- Series of supersteps
  1. Concurrent computation
  2. Communication
  3. Barrier synchronization

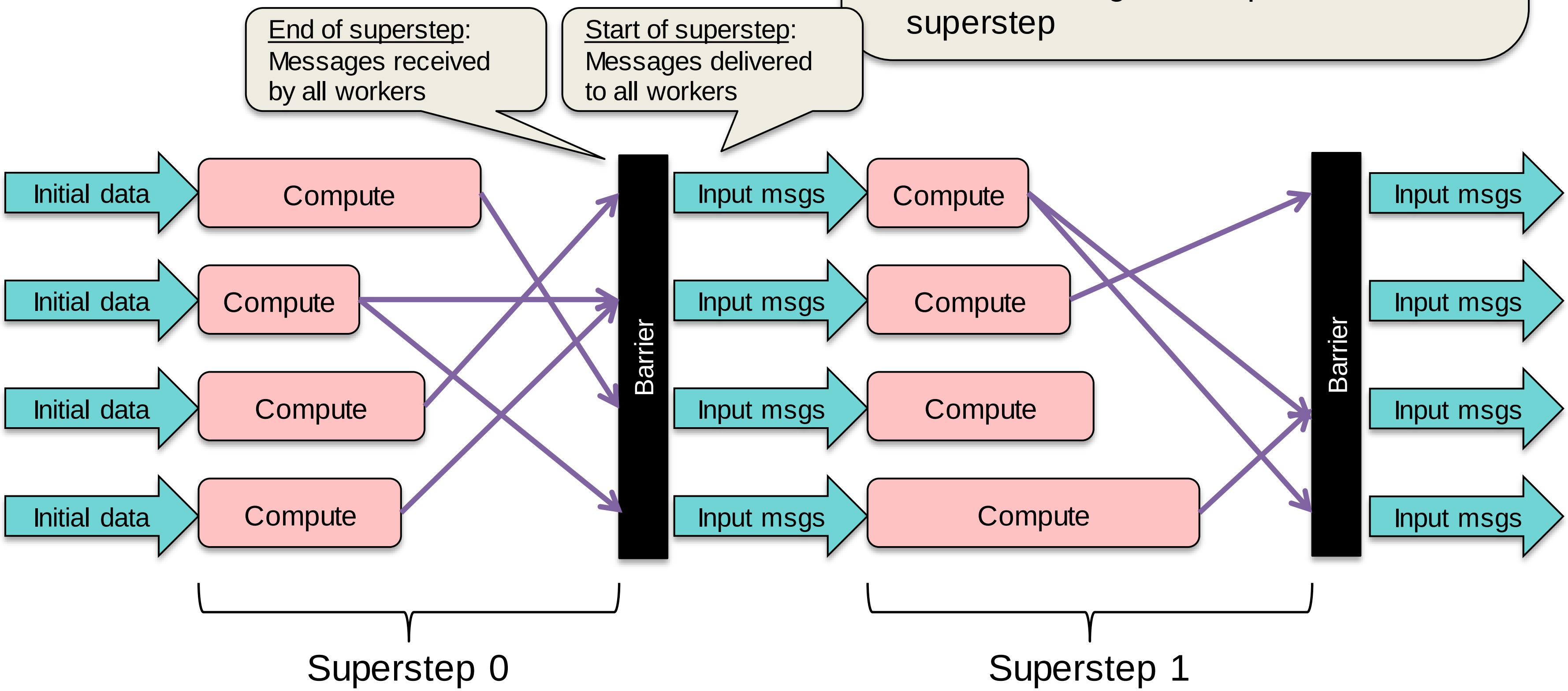
- Processes (workers) are randomly assigned to processors
- Each process uses only local data
- Each computation is asynchronous of other concurrent computation
- Computation time may vary



Bulk Synchronous Parallel (BSP)

- Series of supersteps
  1. Concurrent computation
  2. Communication
  3. Barrier synchronization

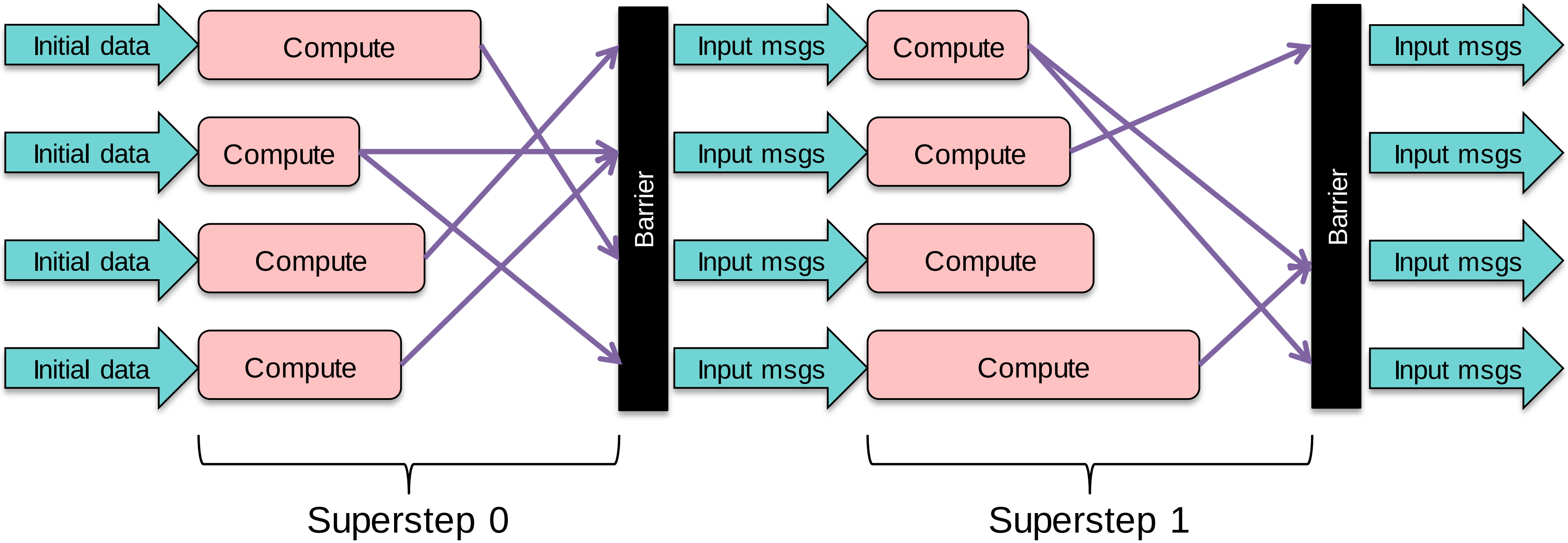
- Messaging is restricted to the end of a computation superstep
- Each worker sends a message to 0 or more workers
- These messages are inputs for the next superstep



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 failure, restart computation from 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 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 ← *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

Graphs are common in computing

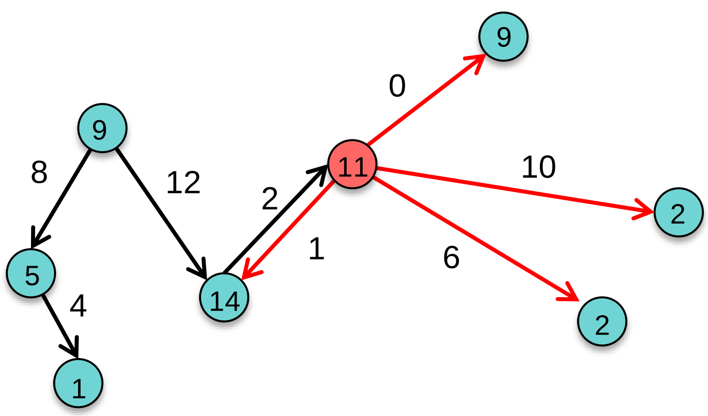
- Social links
  - Friends
  - Academic citations
  - Music
  - Movies
- Web pages
- Network connectivity
- Roads
- Disease outbreaks

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
    - 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

Categories and Subject Descriptors

General Terms

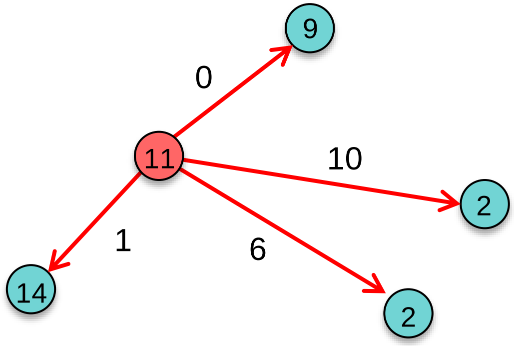
Keywords

1. INTRODUCTION

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

Pregel: computation

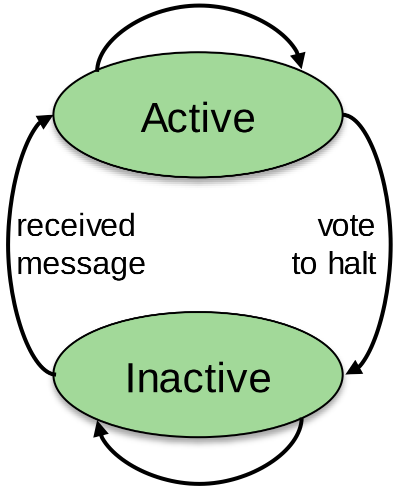
- Computation: series of supersteps
  - Same user-defined function **runs on each vertex**
    - Receives messages sent from the previous superstep
    - May modify the state of the vertex or of its outgoing edges
    - Sends messages that will be received in the next superstep
      - Typically to outgoing edges
      - But can be sent to any known vertex
    - May modify the graph topology
- Each superstep end with a **barrier** (synchronization point)



Pregel: termination

Pregel terminates when every vertex votes to halt

- 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 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
- Page rank
  - Each iteration: update web page ranks based on messages from incoming links.

Simple example: find the maximum value

- Each vertex contains a value
- 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

Semi-pseudocode:

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

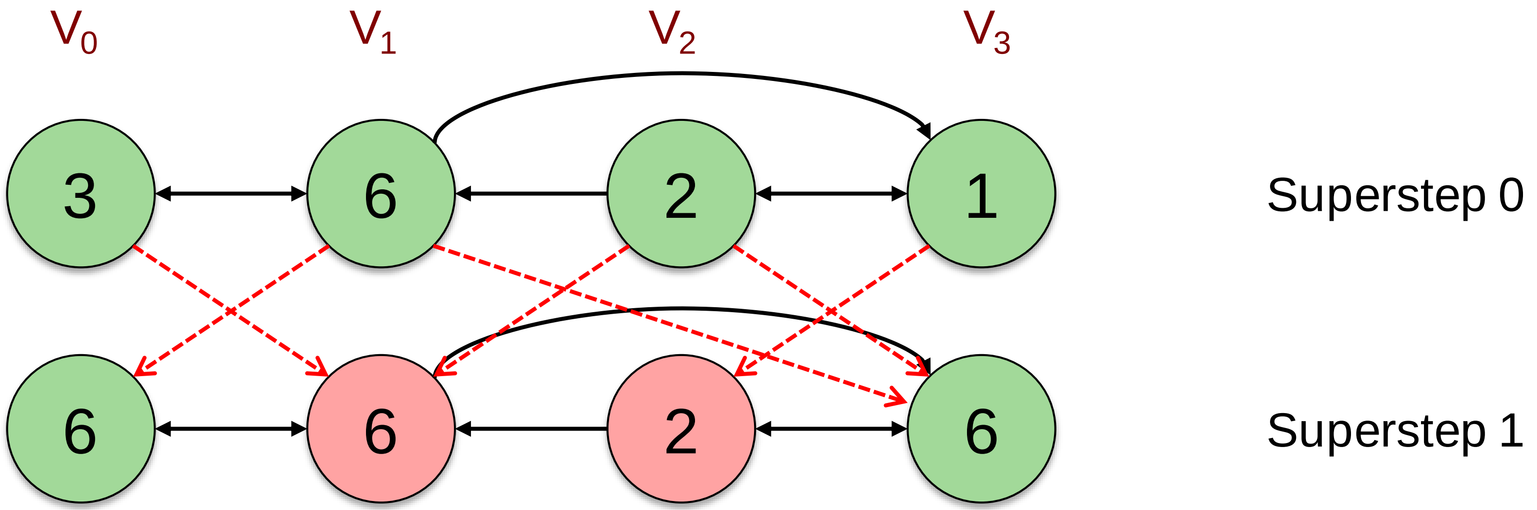
```
class MaxValueVertex
    : public Vertex<int, void, int> {
  void Compute(MessageIterator *msgs) {
    int maxv = GetValue();
    for (; !msgs->Done(); msgs->Next())
        maxv = max(msgs.Value(), maxv);

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

find maximum value

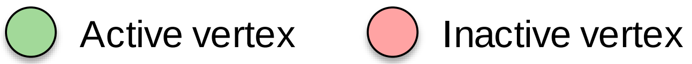
send maximum value to all edges

Simple example: find the maximum value

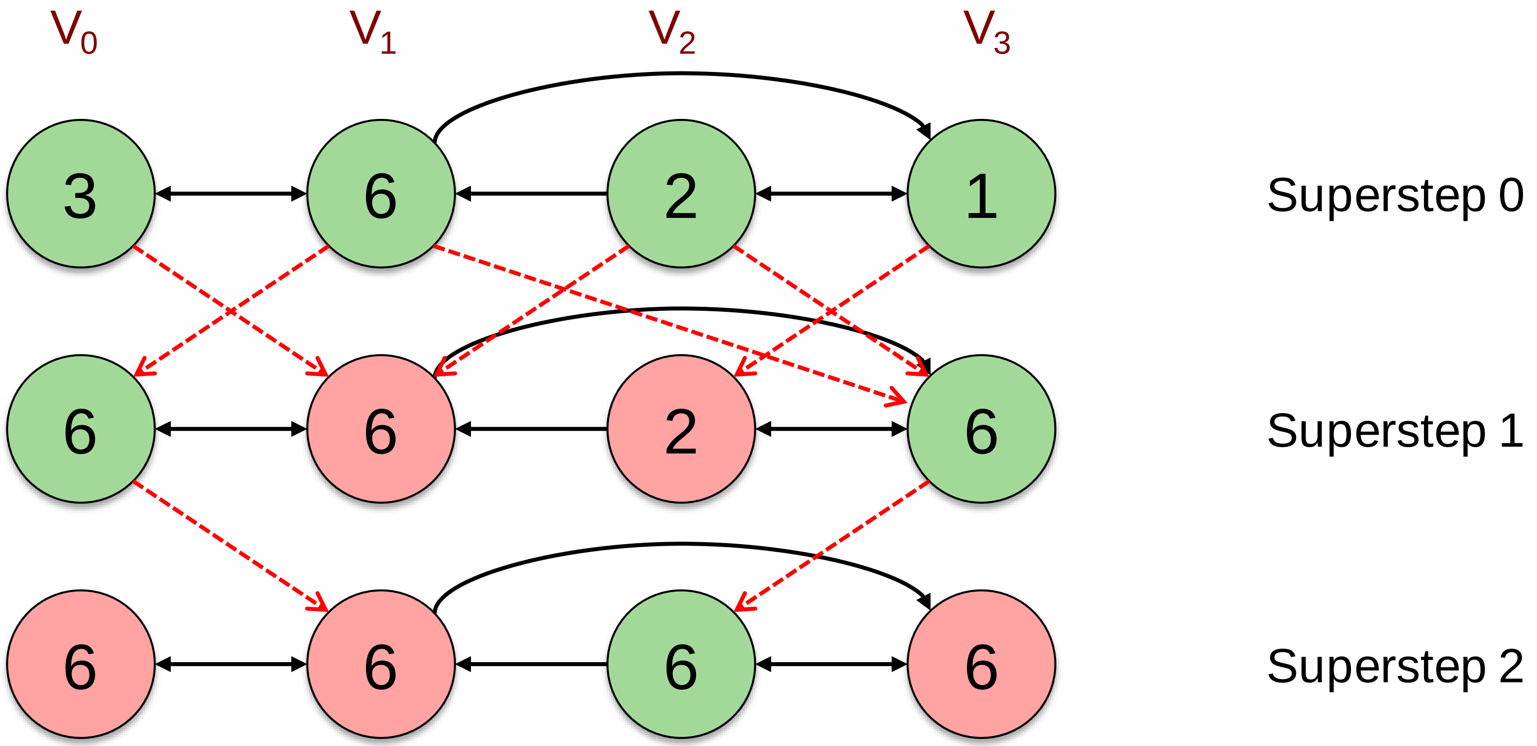


Superstep 0: Each vertex propagates its own value to connected vertices

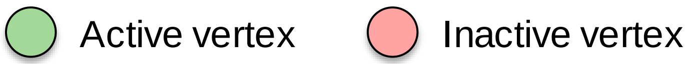
Superstep 1: V_0 updates its value: 6 > 3
V_3 updates its value: 6 > 1
V_1 and V_2 do not update so vote to halt



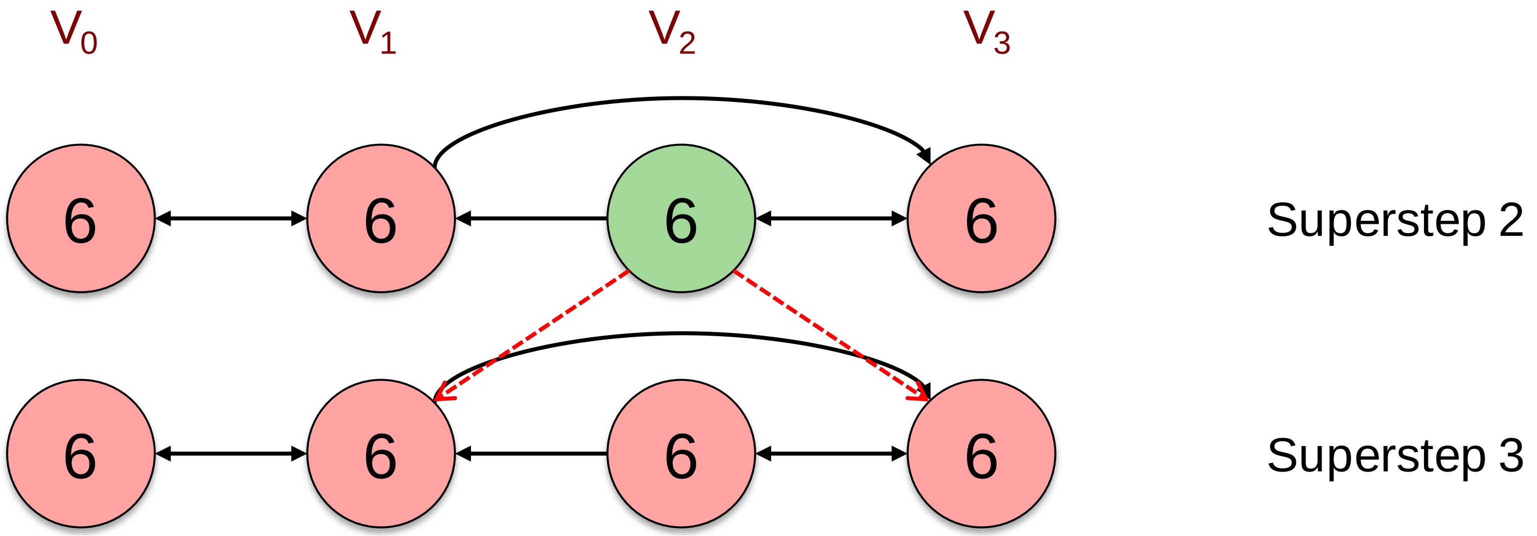
Simple example: find the maximum value



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

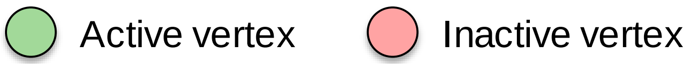


Simple example: find the maximum value



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!



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*

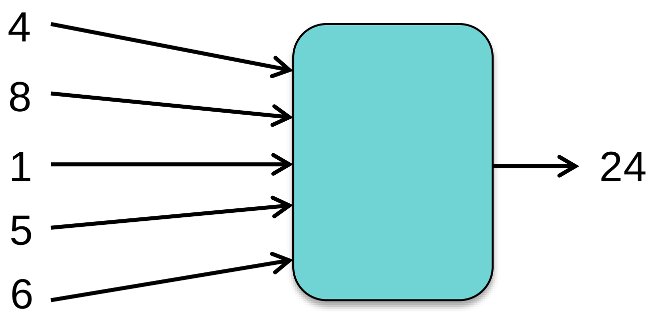
Pregel API: Basic operations

- A user subclasses a Vertex class
- Methods
  - **Compute**(MessageIterator*): Executed per active vertex in each superstep
    - MessageIterator identifies incoming messages from previous supersteps
  - **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)

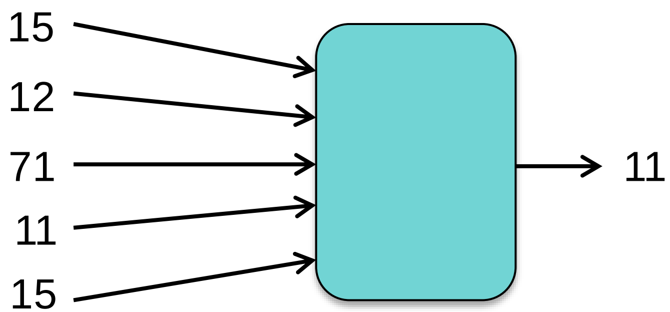
Pregel API: Advanced operations

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 may be combined



Combiner
Sums input messages



Combiner
Minimum value

Pregel API: Advanced 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: Advanced 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

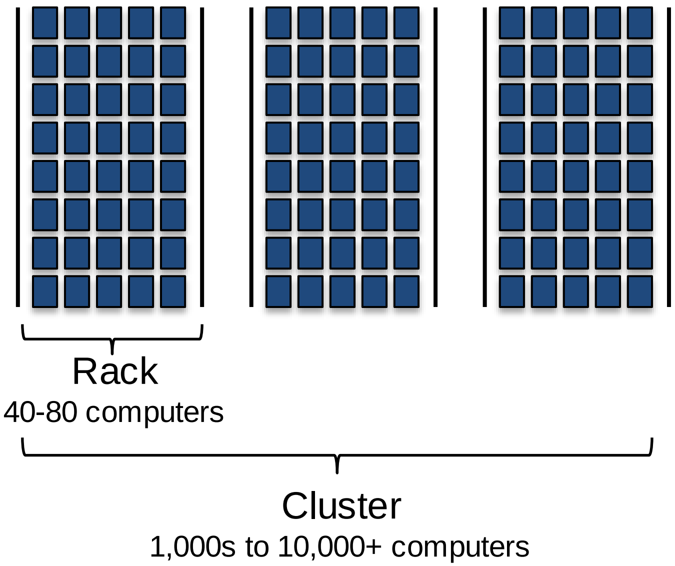
Pregel Design

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

- Cluster's name server = chubby
  – 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* ⇒ worker

    May deviate: e.g., place vertices representing the same web site in one partition
  - More than 1 partition per worker: improves load balancing
- Worker
  - Responsible for its section 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

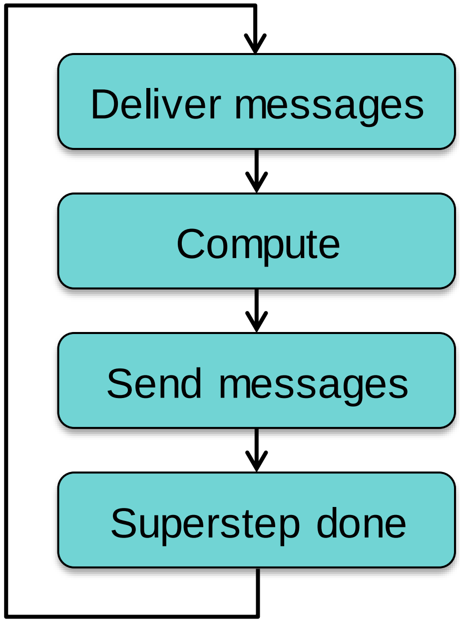
- Input = set of records
  - Record = vertex data and edges
  - Assignment based on file boundaries

- Worker reads input
  - If it belongs to any of the vertices it manages, messages sent locally
  - 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
- 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
- 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 Facebook to analyze the social graph of users
  - 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

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
  - Distributes 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